深入理解并行编程

编辑

Paul E. McKenney
Linux Technology Center
IBM Beaverton
paulmck@linux.vnet.ibm.com

2011-02-12

翻译

谢宝友(xie.baoyou172958@zte.com.cn，scxby@163.com)
鲁阳(luyang.co@gmail.com)
陈渝(wilbur512@gmail.com)

译者简介

谢宝友：毕业于四川省税务学校税收专业，现供职于中兴通讯操作系统团队，对操作系统内核有较强的兴趣。专职于操作系统内核已经有四年时间。目标是利用十年时间，成为一名真正的“内核菜鸟”。个人主页是：http://xiebaoyou.blog.chinaunix.net。

主要工作是对 Linux 进行分析，解决遇到的标准内核故障；并向项目组提出应用程序优化措施。也从头编写过自主知识产权的操作系统。

鲁阳：2009 年硕士毕业于成都电子科技大学，后加入中兴通讯操作系统团队。不玩 Linux 则已，一玩就深度上瘾，“熟读唐诗三百首，不会作诗也会吟”，每日攻读内核不辍，对 Linux 内核内存管理子系统有较深刻的认识。邮箱 luyang.co@gmail.com。

主要工作是分析 Linux 内核代码，为中兴 Embsys Linux 内核开发包括调度、内存管理、AMP 支持和编译框架在内的定制功能，参与过中兴自主知识产权操作系统（非 Linux）的实现。

陈渝：2009 年从成都电子科技大学计算机专业毕业后，加入中兴通讯操作系统团队。主要负责中兴定制 linux 系统的各种 bug shooting，性能调优等。熟悉 mips 架构，对 kexec 有研究，参与过自主产权操作系统的 mips 架构实现。

wilbur512@gmail.com

法律声明

这代表作者的观点，并不一定代表雇主的观点。

IBM、zSeries 和 Power PC 是国际商用机器公司在美国、其他国家（或者兼而有之）的商标或者注册商标。

Linux 是 Linus Torvalds 的注册商标。

i386 是 Intel 公司或者它在美国、其他国家（或者兼而有之）的子公司的注册商标。

其他公司、产品以及服务名称也许是某些公司的商标或者服务标记。

前言

本书的目的是帮助您理解：如何编写基于共享内存的并行程序，而不用经历一次危险的旅程[1]。通过描述以往行之有效的算法及设计，我们希望帮助您避免一些将困扰并行编程的意外错误。但是，您应当把本书作为工作的基础，而不是把它作为解决问题的全部指南。如果您认可的话，您的使命是帮助促进并行编程的发展，最终使得本书显得过时；并行编程将不再被认为是困难的，希望本书对您来说显得更简单。

本书遵照并行编程领域一个革新式的变化。这个变化首先来自于科学、研究领域，以及一个大的工程项目。为展示这个工程项目，本书将审核该项目与并行编程相关的特定工作，将描述它们将如何有效的处理一些特殊情况。

编写本书的目的，是希望展示这个工程项目，它将避免新一代并行编程者重复发明一些旧轮子，从而专注于新的领域。虽然本书主要是为自己学习而准备的，但是我想它可能还有其他用途。希望本书对您也有用，并且并行编程会带给您许多快乐、兴奋，这些快乐、兴奋已经带给作者多年了。

[1]或者可以更准确的说，不用遭受比非并行编程更大的危险。我想起来了，不必说得太多，附录 A 描述了一些重要的并行编程与顺序编程之间的问题。

1. 简介

并行编程已经获得这样一个声誉：它是一个黑客可以参与解决问题的困难领域之一。教科书已经警告过我们关于死锁、活锁、竞争条件、不确定性、实时延迟的危险。这些危险是非常现实的。在如何处理这些问题方面，我们已经累积了多年的经验，也留下了多年的感情伤痕、白头发，这些失去的头发就和我们累积的经验一样多。

新技术已经变得难于使用，但是随着时间的推移，终究会更容易使用。例如，驾驶技术曾经是一项罕见的技能，但是在很多发达国家，这项技术已经很普通了。这个引人注目的变化来自于两个基本原因：（1）汽车变得更便宜、更好用，因此更多的人有机会学习驾驶。（2）由于自动变速箱、自动刹车、自动启动、大幅提高可靠性以及很多其他技术方面的改进等等原因，汽车更易于操控。

同样的，其他技术（包括计算机技术）也有很多改进。编写程序不再是一件困难的事情。电子制表软件允许大多数非程序员从他们的计算机得到结果。数十年前，这需要专家才能胜任。或许更好的例子是 web 冲浪及内容发布。自 2000 年以来，非专业人士就使用各种社交网络，它们变得非常简单了。即使在 1968 年，这样的内容发布还是一个远期研究项目，那时还在用“落在白宫草坪上的 UFO”这样的比喻来描述它们。

因此，如果您想说并行编程在将来仍然象现在这么困难，那么请提供您的证据。请注意几个世纪以来的各种例子。

1.1. 导致并行编程困难的历史原因

如标题所示，本书将采取一种不同的方式。不是抱怨并行编程的困难，而是认真分析并行编程为何困难的原因，然后帮助读者克服这些困难。正如后面将看到的一样，这些困难分为几类，包括：

1. 并行系统曾经的高价格以及相对罕见。
2. 研究人员以及从业人员的稀少。
3. 缺少公开的并行代码。
4. 缺少并行编程的工程经验。
5. 任务间通信代价高昂，即使是共享内存的计算机系统也是如此。

几乎所有这些困难现在都已经被克服了。首先，在过去数十年里，并行系统的价格已经从数倍于房屋减少到一辆二手车的价格。这首先得感谢多核系统的出现。早在 1996 年，就有文章在唤起人们对多核 CPUs 优势的注意。同时，在 2000 年的时候，IBM 在它的高端 POWER 系列中引入了多线程，在 2001 年引入了多

核。2000 年 1 月，Inte 在它的 Pentium 产品线引入超线程。AMD 和 Intel 在 2005 年的时候同时引入了双核 CPUs。2005 年晚些时候，Sun 在 Niagara 中引入了多核/多线程。实际上，在 2008 年的时候，再找一个单核 CPU 已经是一件困难的事情了。单核 CPU 仅仅与上网本及嵌入式设备相关。

其次，廉价实用的多核系统的出现，意味着对几乎所有研究者和从业者来说，并行编程体验已经变得很实在了。实际上，并行系统已经处于学生及业余爱好者的预算范围之内了。因此，我们可以预期：将会增加大量的与并行系统相关的发明创造。

第三，在 20 世纪，高并行系统软件系统几乎总是被作为私有秘密进行保护，在 21 世纪，已经有大量开源（因此公众可用）的并行软件项目出现。其中包含 Linux 内核，数据库系统，消息传递系统。本书将主要描述 Linux 内核，但是将提供一些适合用户层应用程序的资料。

第四，即使 1980 年代和 1990 年代，那些大型并行项目几乎都还是私有的，但是这些项目已经产生很多有产品经验的社区核心开发者。本书的一个主要目的就是表现这些工程经验。

不幸的是，第五个困难，任务间的通信代价高昂的问题，仍然是存在的。虽然在新的千年中会重视这个困难，倡根据 Stephen Hawking 的研究，光的有限速度以及原子特性会限制这个领域的进展。幸运的是，这个困难自 1980 年代以来就存在了，工程技术已经演变，处理这个困难已经有有效的策略。另外，硬件设计师更清楚这些问题，也许在将来，硬件对于并行编程来说会更友好，这将在 2.3 节讨论。

小问题 1.1：加油吧!!!在数十年间，并行编程将会是非常困难的。也许您会觉得它不是那么困难。你还在玩什么游戏？即使并行编程不像通常宣传的那么困难，它也比串行编程有更多的工作需要做。

小问题 1.2：并行编程如何才能变得与串行编程一样简单？

1.2. 并行编程的目标

相对于串行编程来说，并行编程有如下三个主要目标：

- 性能
- 生产率
- 通用性

小问题 1.3：哦，真的吗？？？正确性、可维护性、健壮性等方面如何？

小问题 1.4：如果确性、可维护性、健壮性都没有位于目标列表中，为什么生产率和通用性在目标列表中？

小问题 1.5：倘若并行编程更难于证明其正确性，为什么正确性不在目标列表中？

小问题 1.6：仅仅是为了乐趣？

以上的目标将会在随后的章节中详细说明。

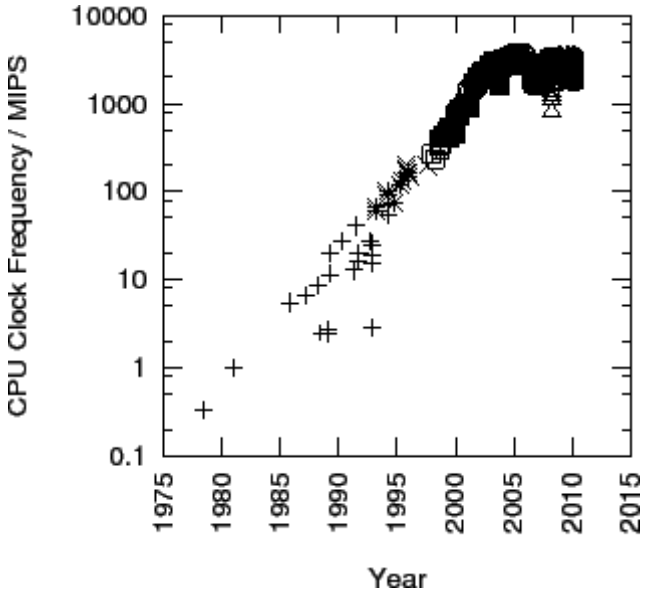


图 1.1 Intel CPUs MIPS/时钟频率趋势

1.2.1. 性能

大多数并行编程的目的主要是性能。毕竟，如果不考虑性能，为什么不编写串行代码，这样不是更 happy 吗？这很可能会更简单，并且您可能会更快的完成任务。

小问题 1.7：没有其他不考虑性能的情况吗？

注意：性能是一个总的概念，包含可扩展性（每 CPU 性能）及效率（如每瓦特性能）。

也就是说，关注焦点已经从硬件转向并行软件。这是由于虽然摩尔定律仍然在晶体管密度方面有效，但是在提高单线程性能方面已经不再有效。这可从图 1.1 看出来。

注：这表明最新 CPUs 时钟频率的理论上的能力，它可以在一个时钟周期执行一个或者多个指令。对以前的 CPUs 来说，即使最简单的指令也需要多个时钟周期。采取这个方案的原因是新 CPUs 执行多个指令的能力受到内存系统性能的限制。

这意味着编写单线程代码并简单的等待 CPUs 一两年时间不再是一个可行的方法。所有主流厂商最近的趋势是朝多核/多线程系统发展，并行是充分利用这些系统性能的好办法。

因此，首要目的是性能而不是可扩展性。特别是提供一个简单的方法达到线性的扩展性能的目标，而不用扩展每 CPU 性能。对于一个 4-CPU 系统，您将如何选择？是编写一个程序，它在一个单 CPU 上提供每秒 100 个事务？还是编写一个程序，在每个 CPU 上提供 10 个事务，并且在多核上扩展它？第一个程序看起来是更好的选择。如果您有权访问一个 32-CPU 系统的话，也许你的答案有所不同。

也就是说，您拥有多核 CPUs，这并不是必然要全部使用它们的原因，特别是最近以来，多核系统的价格已经下降很多。理解这一点的关键是：并行编程主要是性能优化，因为它是众多优化措施中的一种。如果您当前的程序已经足够快了，没有必要再优化，也不必将它并行化。*注：当然，如果您对并行编程有兴趣，就有更多的理由进行并行化而不管是否有必要。*出于相同的原因，如果您的串行程序需要优化为并行程序，您需要在不同并行算法中进行比较。这需要小心一点，太多的书籍在分析并行算法时忽略了使用不同算法的优先顺序。

1.2.2. 生产率

小问题 1.8：为什么有这么多非技术性问题的话？？？并且不仅仅是非技术性问题，还有生产率这个东东？谁会关心它？

数十年来，生产率已经变得更加重要。要明白这点，考虑价值数百万美元的早期计算机时代，一个工程师的年薪只有几千美元。为这样一台机器奉献十个工程师的项目组，将它的性能提高 10%，这相当于他们薪水的很多倍。

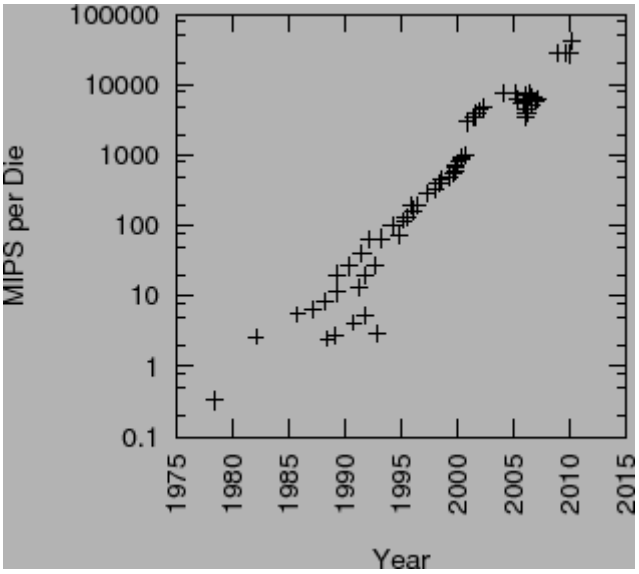


图 1.2 MIPS per Die for Intel CPUs

CSIRAC 就是这样的机器，它是最早的存储程序的计算机，至 1949 年开始运行。这台机器拥有 768 个字的 RAM，可以放心大胆的说，生产率问题对这台

机器来不存在任何问题。因为这台机器在晶体管时代以前生产，它由 2000 个真空管组成，运行频率是 1kHz，每小时消耗 30kW 的电量，重量超过 3 吨。

60 年后（2008 年），想要买到这么小的计算能力的机器是非常困难的。与之最接近的是 8 位嵌入式微处理器 Z80。这个 CPU 有 8500 个晶体管，仍然能够在 2008 年购买到，价格低于 2$。与 CSIRAC 相比，软件开发人员的费用根本微不足道。对 Z80 来说，则并不是这样。

CSIRAC 和 Z80 是长期趋势中的两个点，这可以从图 1.2 看出来。该图标示出过去三十年内计算能力增长的近似值。请注意，由于多核 CPUs 的出现，使得这种增长趋势没有减弱，即使在 2003 年开始遇到了 CPU 频率方面的瓶颈。

硬件价格不断下降的后果之一，就是软件生产率越来越重要。仅仅高效的使用硬件已经不再足够，高效的利用开发者已经变得同样重要。串行硬件时代已经很久了，仅仅在最近一段时期，并行硬件才变成低成本商品。因此，高生产率的创建并行软件仅仅在最近才变得非常重要。

1.2.3. 通用性

为开发并行程序的高费用进行辩护的方式之一是争取更大的通用性。一个更通用的软件能够比一个不怎么通用的软件更能分散费用。

不幸的是，通用性会带来更大的性能损失和生产率损失。要明白这点，考虑下面这些流行的并行编程环境：

C/C++ “锁及线程”：这包含 POSIX 线程（pthreads），Windows 线程，以及很多操作系统内核环境。它们提供了优秀的性能（至少在 SMP 系统上是如此），也提供了良好的通用性。可惜的是生产率较低。

Java：这个编程环境与生俱来就有多线程能力，广泛的认为它比 C 或者 C++更有生产率。它能够自动进行垃圾收集，并且拥有大量的类型库。但是，虽然它的性能在过去十年间有了长足的进步，但是通常被认为低于 C 和 C++。

MPI：这个消息传递接口向大量的科学和技术计算提供能力，它提供无比伦比的性能和可扩展性。理论上讲它实现了通用的目的，但是通常被用于科学计算。通常认为它的生产率甚至低于 C/C++环境。

OpenMP：这个编译指令集能被用于并行循环，因此被用于特定任务，这也限制了它的性能。但是，它比 MPI 和并行 C/C++更简单。

SQL：结构化编程语言 SQL 非常特殊，仅仅运用于数据库查询。但是，它的性能非常好，TPC 测试性能非常出色。生产率也很优秀，实际上，这个并行编程环境允许对并行编程知之甚少的人员很好的使用大型并行机器。

要从并行编程环境中解脱出来，并且提供优秀的性能、生产率、通用性，这

样的环境仍然不存在。在这样的环境产生以前，需要在性能、生产率、通用性之间进行权衡。其中一种权衡如图 1.3 所示：

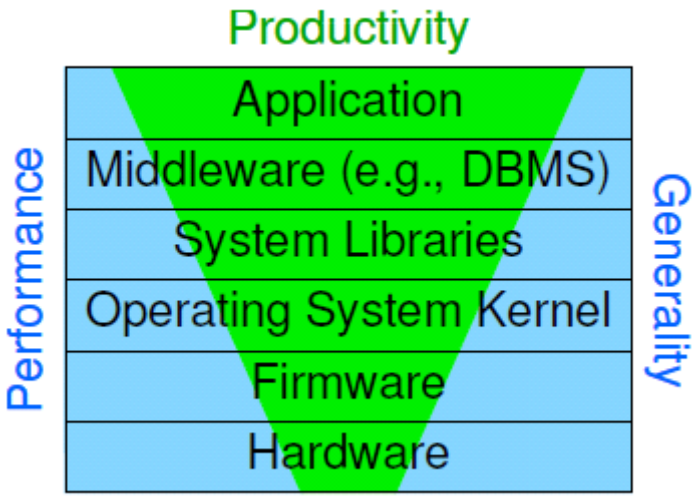


图 1.3: Software Layers and Performance, Productivity, and Generality

它说明一个事实：越往上层，生产率是如何变得越来越重要的。然而越往下层，性能和通用性就变得越来越重要。一方面，大量的开发工作消耗在上层，并且必须考虑通用性以降低成本。下层的性能损失很不容易在上层得到恢复。在靠近堆栈的顶端，也许只有少数的用户工作于特定的应用。这种情况下，生产率是最重要的。这解释了这样一种趋势：越往上层，采用额外的硬件通常比额外的开发者更划算。本书主要为底层开发者准备，性能和通用性是主要关心的地方。

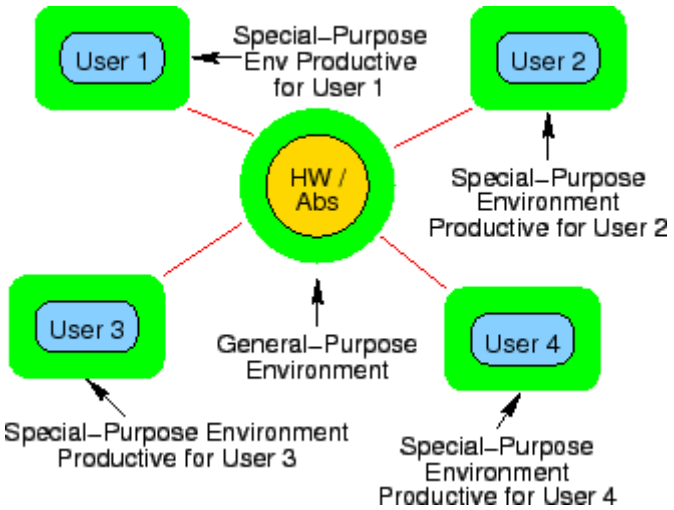


图 1.4 在生产率和通用性之间权衡

不得不承认，在许多领域，生产率和通用性之间，长时间以来都是存在矛盾的。例如，射钉枪就远远比一个铁锤更具有产品性，但相对于射钉枪，铁锤除了敲钉子外还能做很多其他事情。因此我们在并行计算领域看到类似的折衷就不足为奇了。这种均衡如图 1.4 所示。这里，用户 1，2，3 和 4 需要计算机帮他们完成特定的任务。对于某个用户来说，最具生产率的语言或环境，就是专注为此用

户服务，而不需要做任何编程，配置或者其他设置。

小问题 1.10：这个理想真可笑！为什么不专注于一些实际可行的方面？

不幸的是，一个正在为用户 1 服务的系统不大可能同时做用户 2 的任务。也就是说，最具生产率的语言和环境是按域划分，并且牺牲了一些通用性。

另外一个选择是为硬件系统量体裁衣的修改语言或环境（例如，底层语言汇编，C，C++，或者 Java），或者是一些抽象语言（例如，Haskell，Prolog，或者 Snobol）如图 1.4 中心的圆形区域。这些语言被认为是对所有的用户 1，2，3，4 都不适合。也就是说，相对于按域划分的语言和环境来说，系统的通用性，随着生产率的减少而此消彼长。

我们要时刻牢记，并行计算的性能、生产率和通用性之间经常互相冲突，现在，是时候进行深入分析来避免这些并行计算冲突的解决方案了。

1.3. 并行编程的替代方案

在考虑并行计算的替代方案前，必须先想想自己希望并行计算能为你做什么。正如第 1.2 节所介绍，并行计算的目标是性能，生产率，以及通用性。

尽管从历史上说，大多数开发者最关心第一个目标，其他目标的好处是可以减轻您为使用并行计算进行辩护的必要。这节的剩下部分只关注性能方面的提升。需要牢记的是，使用并行计算只是提高性能的方案之一。其他熟知的方案按实现难度递增的顺序罗列如下：

- ✓ 运行多个顺序应用实例
- ✓ 利用现有的并行软件构建应用
- ✓ 对串行应用进行逻辑优化

1.3.1. 顺序应用多实例化

多实例可以避免使用并行计算。有很多种方法来实现这种方案，这依赖于应用的结构。

如果你的程序需要分析大量不同场景，或者独立的数据集，一个简单且有效的方式，是创建单个顺序执行的分析程序，然后在脚本环境下（例如 bash）启动一定数量的程序实例，让他们并行执行。某些情况下，这个办法可以轻松的扩展到集群机器中。

这种方法看上去像是欺骗，实际上确实也有一些人诋毁这种程序为“另人尴尬的并行”。确实，折衷方法有一些潜在的缺点，包括增加了内存消耗，CPU 指令周期浪费在重复计算中间结果上，以及增加了数据拷贝操作。但是，这种方案通常是非常有效的，在较少甚至几乎没有额外修改下获得了极大的性能提升。

1.3.2. 使用现有的并行软件

目前并行软件已经可以顺畅的用于单线程程序，包括关系数据库，web 服务器，以及 map-reduce 环境。例如，一个通常的设计是为每个用户提供一个单独的处理进程，每个进程生成 SQL 语句，然后在关系数据库上并行执行。用户程序只需要负责用户接口，而关系数据库则负责处理所有的并行和持久性问题。

采用这种方法通常会牺牲性能，至少逊色于细致构思的并行程序。但是，折衷牺牲经常被证明是值得的，因为可以显著的减少开发难度。

1.3.3. 性能优化

追溯到 21 世纪初期，CPU 的性能每 18 个月翻一番。在这样的背景下，增加新功能的重要性远大于对反复对性能进行优化。现在，摩尔定律仅仅适用于增加晶体管的密度，对于同时增加晶体管密度和单个晶体管的性能来说，已经不再适用。因此，是时候重新审视性能优化的重要性了。

毕竟，性能优化可以减少能源消耗以及提升性能。

从这种观点来看，并行计算是另外一种性能优化，尽管当并行系统变的更便宜和更可靠后，性能优化变的更有吸引力。但是，我们最好保持清醒，来自并行计算的速度提升，与 CPU 个数大约成正比。然而对软件进行优化，带来的性能提升是指数级的。

还有一点：不同的程序会有不同的性能瓶颈。并行计算只能淡化某些瓶颈。比如，假设你的程序花费最多的时间在等待磁盘驱动的数据。在这种情况下，让你的程序在多 CPU 下运行并不大可能会改善性能。实际上，如果进程正在读取一个旋转的磁盘上的大型顺序文件，并行设计程序也许会使它变得更慢。相反，你应该添加更多的磁盘、优化数据以使这个文件能变得更小（因此读的更快），或者，如果可能的话，避免读取如此多的数据。

小问题 1.11：还有其他的哪些瓶颈会阻碍通过添加 CPU 带来性能提升？

并行技术是一个有效的优化技术，但是它并不是唯一的技术，同时也不是对所有的情景都适用。当然，你的程序越是容易并行化，那么并行计算对你的程序来说就越有可能成为一种优化措施。

并行化号称非常复杂，自然而然的就引出这个问题：“到底是什么让并行计算变的如此复杂？”

1.4. 是什么使并行编程变得复杂？

需要注意的是：并行计算的困难，有人为因素的原因与并行计算本身的技术属性的原因，二者给并行计算带来的困难是差不多的。这是由于我们需要人为干涉并行计算的过程，人和计算机间的双向交互需要人和机器都执行同等复杂的操作。因此，采用抽象或者数学分析将极大的限制实用性。

工业革命时，人机接口被人类行为学重新评估，也就是所谓的工时-动作研究（xie.baoyou 注：可以回忆一下《政治经济学》）。虽然有一些基于人类行为的并行计算研究，但由于这些研究关注面过窄，导致不能展示任何通用的结果。另外，给出一个跨越一个数量级的程序员生产率正常范围，要对生产率检测到（比如 10%）的差别是不现实的。尽管能检测到的多个数量级的差异是非常有用的，但最令人振奋的改进是能够检测到一个 10%生产率提升。

因此，我们必须采取不同的方法。

一个这样做的方法是周密的考虑并行程序员的任务，这些任务对于线性编程来说是不需要的。这样，我们就能评估一个语言或者环境支持开发者的程度。这些任务分解成 4 个分类，如图 1.5 所示，每个接下来都会在下面的章节讲到。

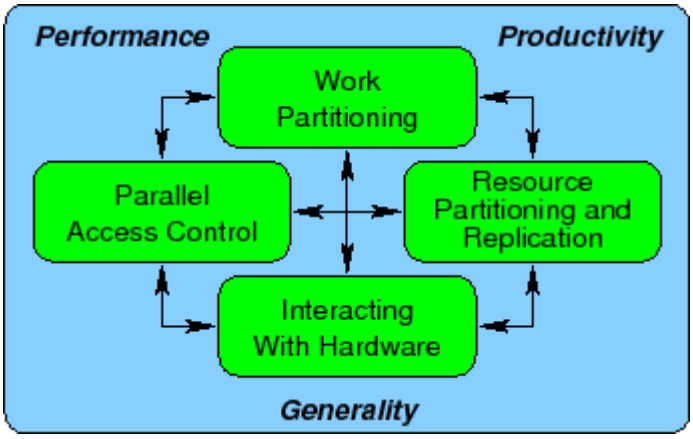


图 1.5 并行计算任务分工

1.4.1. 工作分割

工作分割绝对是并行计算需要的：如果存在一个小型的工作，那么它同时只能被一个 CPU 执行，这是由顺序执行所定义的。但是，分割代码需要十分小心，比如，不均匀的分割.一旦分割结束后，会导致顺序执行的结果。在不是很极端的情况下，负载均衡能被用来完全利用硬件，因此获得更优化的性能。

另外，工作分割可能导致全局错误和事件处理变得复杂：一个并行程序可能需要实施一些同步措施以便安全的处理这些全局事件。

每个分割部分需要一些交互：毕竟，如果一个给定的线程基本不交互，那么他就没有作用并且不大需要被执行。但是，由于交互会引起开销，不仔细的分割选择会导致严重的性能退化。

而且，并行线程的数量常常必须被控制，因为每个线程都会占用一些资源，比如 CPU cache 空间。如果过多的线程同时执行，CPU cache 将会溢出，引起过高的 cache miss，从而降低性能。从另外一方面看，大量的线程可能会带来大量计算和 I/O 操作。

小问题 1.12：除了 CPU cache 容量外，还有其他哪些因素需要限制并行线程个数？

最后，合法的并行线程会大量增加程序的状态空间，导致程序难以理解，降低生产率。在其他条件相同的情况下，更小的状态空间有更通用的结构，更容易被理解。好的并行设计可能拥有大量及其巨大的状态空间，不过却由于他们采用了通用的结构，而便于理解，可是糟糕的设计可能会难以理解，就算他们采用相对小的状态空间。最好的设计采用尴尬的并行主义，或者将问题转交给尴尬并行解决方案。在以上情况中，尴尬并行实际上是财富的尴尬。当前的状态催生优秀的设计，更多的工作需要被引入以在状态空间大小和结构上作出更好的判断。

1.4.2. 并行访问控制

给定一个单线程的顺序进程，单线程对所有进程的资源都有访问权。这些资源主要是内存数据结构，但也可能是 CPU，内存（包括 cache），I/O 设备，计算加速器，文件，以及更多其他的。

第一个并行访问控制问题是访问特定的资源是否受限于资源的位置。比如，在许多消息传递环境中，本地变量的访问是通过表达式和赋值，但是远程变量访问是通过一套完全不同的语法，经常引入消息传递机制。POSIX 线程环境，结构化查询语言(SQL)，以及分割的全局地址空间环境例如通用并行 C，提供了隐式访问，然后消息传递结构通常提供显式的访问，因为访问远程数据需要显式的消息传递。

其他的并行访问控制问题是线程如何协调访问资源。这种协调是由非常多的同步机制通过提供不同的并行语言和环境来实施的，包括消息传递，加锁，事务，引用计数，显式计时，共享原子变量，以及数据所有权。需要传统的并行编程关注由此引出的死锁，活锁，和事务回滚。这种框架如果要详细讨论需要涉及到同步机制的比较，比如加锁相对于事务内存，但这些讨论不在这节的讨论范围内。

1.4.3. 资源分割和复制

最有效的并行算法和系统，专注于资源并行算法，所以比较明智的做法是通过分割写资源操作和读资源操作来开始并行化。问题是这些频繁访问的数据，可能会在计算机系统间，大存储设备间，NUMA 结点间，CPU 之间（或者核与硬件线程），页面，cache 缓存线，同步原语的实例，或者代码临界区之间造成问题。比如，分割加锁原语叫做“数据加锁”。

资源分割通常依赖于用例，比如，数值计算频繁的将矩阵按行，列，或者子矩阵进行分割，而商业计算则频繁的分割写数据结构操作和读数据结构操作。比如，商业计算进程可能会为一个特定用户从集群里分配一些小计算系统。进程可能会静态的分割数据，或者随时间改变分区。

资源分割非常有效，但是随着带来的复杂数据结构也非常有挑战性。

1.4.4. 与硬件交互

硬件交互通常是操作系统的核心，编译器、库，或者其他的软件环境基础。开发者涉及到新的硬件特性或者组件时，经常需要直接与这些硬件打交道。并且，当需要榨取系统的最后一点性能时，通常需要直接访问硬件。在这种情况下，开发者需要根据目标硬件，对 cache 分布，系统拓扑结构，或者内部协议连接进行量体裁衣。

在某些情况下，硬件可能会被认为是一种受制于分割或访问控制的资源，正如上一节描述的。

1.4.5. 组合使用

尽管这四种功能是基础性的（可单独存在），但好的工程实践会组合使用这些功能。比如，数据并行方案首先把数据分割以便减少组件内的交互需求，然后分割相应的代码，最后对数据分区和线程进行映射以便提升吞吐和减少线程内交互，如图 1.6 所示。开发者于是对每个分区单独进行考虑，显著减少了相关状态空间的大小，进而增加生产率。当然，有些问题是不可分割的，但是从另外一个角度来说，巧妙的变换成可分区的架构，可以大大提高性能和扩展性。

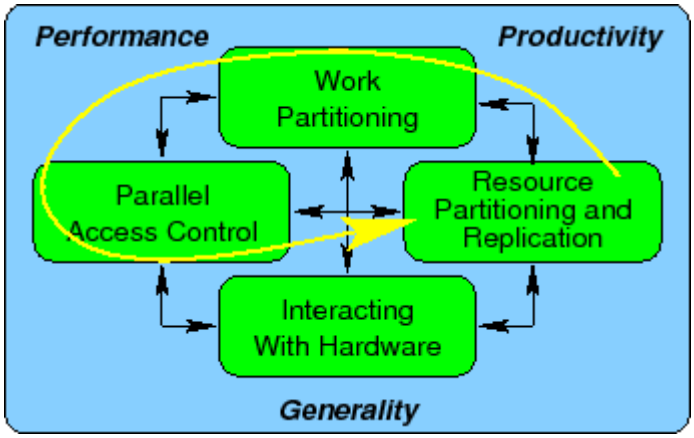


图 1.6 并行计算任务流程

1.4.6. 语言和环境如何对这样的任务进行支持？

尽管许多环境下，开发者需要手动处理这些任务，但在其他一些环境，可以提供有效的自动化支持。比较典型的代表是 SQL，基本实现了对单个长查询和多个独立的查询和更新操作进行自动并行化。

这 4 种任务必须在所有的并行编程中体现出来，但当然不是说开发者必须手工的实现这些任务。随着并行系统变的越来越便宜，越来越有效，我们可以预见这 4 种任务会越来越自动化。

小问题 1.13：并行计算还有其他障碍吗？

1.5. 本书导读

这本书不是一个只关注某些方面的算法优化集合，相反的，他是一本涵盖较多方面和高技术的手册。我们当然不能抗拒将一些我们喜欢的，但是没有通过测试的内容也囊括其中的冲动，因此我们咬牙切齿的把我们心爱的内容放到附录中了。也许早迟有一天，他们中的一些会得到广泛应用，然后我们就能把他们加入到本书的主体中了。

1.5.1. 小问题

“小问题”贯穿全书。有些问题是上下文相关的，但是其他一些需要你联系多个章节，有时候，甚至需要对全书通盘考虑。你能从这本书获得的东西，极大的取决于你对本书的投入程度。因此，读者如果花点心思在这些小问题上，那么将会发现多努力一点是值得的，这样就会得到对并行编程更深入理解的回报。

这些小问题的答案可以在附录 F 找到。

小问题 1.14：这些小问题的答案在哪里？

小问题 1.15：有些小问题似乎是站在读者的角度而不是作者的角度。真是这样吗？

小问题 1.16：这些小问题不太合口味。你有什么意见？

1.5.2. 随书源码

本书附带源码，通常情况下可以从本书 git 树的 CodeSamples 目录下载到。比如，在 UNIX 系统上，你可以敲：

Find CodeSamples –name rcu_rcpls.c –print 来定位文件 rcu_rcpls.c，在 8.3.4 会被用到。其他系统有相应的按文件名检索的方法。

本书的源码可以在

git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git 找到，git 本身可以在 Linux 发行版中找到。PDF 版本以分散的方式，存在于以下链接

http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html

2. 硬件的习性

大多数人根据直觉就知道，在系统间传递消息要比在单个系统上执行简单计算更加耗时。不过，在共享同一块内存的系统的线程间传递消息是不是也更加耗时，这点可就不一定了。本章主要关注共享内存系统中的同步和通信的开销，只涉及了一些共享内存并行硬件设计的皮毛，想了解更多信息的读者，可以翻看 Hennessy 和 Patterson 的经典教材最新版[HP95]。

小问题 4.1：为什么并行软件程序员需要如此痛苦地学习硬件的低级属性？如果只学习更高级些的抽象是不是更简单，更好，更通用？

2.1. 概述

如果只是粗略地扫过计算机系统规范手册，很容易让人觉得 CPU 的性能就像在一条清晰跑道上的赛跑，如下图所示，总是最快的人赢得比赛。

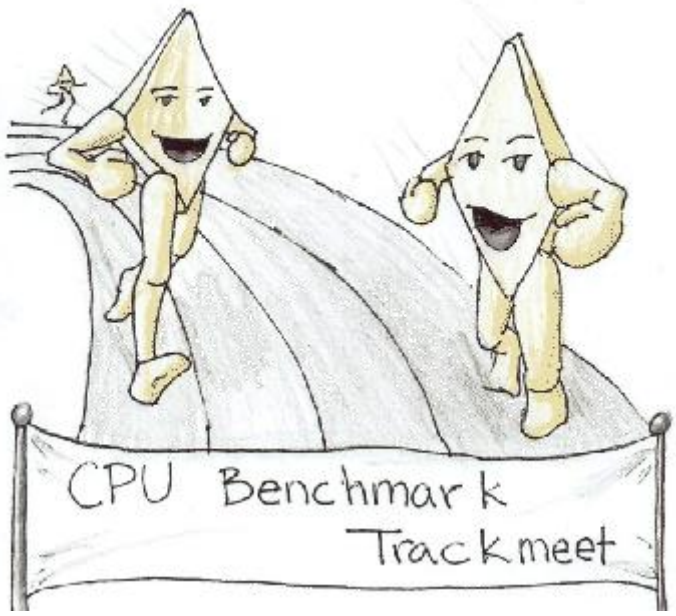


图 2.1 CPU 的最佳性能

虽然有一些只限于 CPU 的基准测试能够让 CPU 达到上图中显示的理想情况，但是典型的程序不像跑道，更像是一条带有障碍的道路。托摩尔定律的福，最近几十年间 CPU 的内部架构发生了急剧的变化。在后面的章节中将描述这些变化。

2.1.1. CPU 流水线

在上世纪 80 年代初，典型的微处理器在处理下一条指令之前，至少需要取指，解码和执行 3 个时钟周期来完成本条指令。与之形成鲜然对比是，上世纪 90 年代末期和本世纪初的 CPU 可以同时处理多条指令，通过一条很长的“流水线”来控制 CPU 内部的指令流，图 2.2 显示了这种差异。

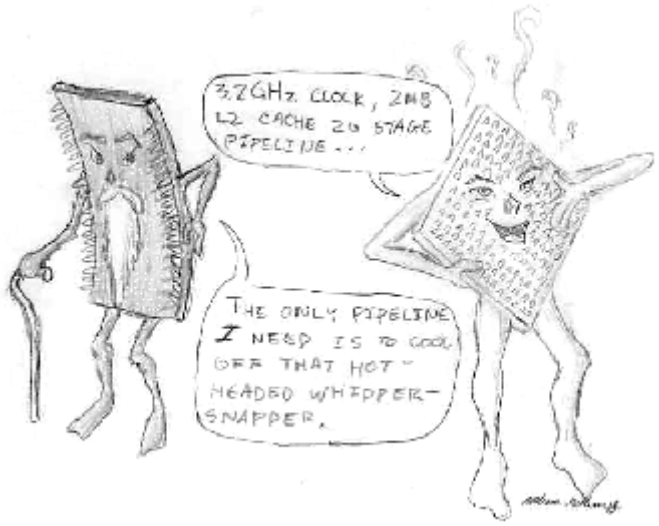


图 2.2 新 CPU 和旧 CPU

带有长流水线的 CPU 想要达到最佳性能，需要程序给出高度可预测的控制流。代码主要在紧凑循环中执行的程序，可以提供恰当的控制流，比如大型矩阵或者向量中做算术计算的程序。此时 CPU 可以正确预测在大多数情况下，代码循环结束后的分支走向。在这种程序中，流水线可以一直保持在满状态，CPU 高速运行。

另一方面，如果程序中带有许多循环，且循环计数都比较小；或者面向对象的程序中带有许多虚对象，每个虚对象都可以引用不同的实对象，而这些实对象都有频繁被调用的成员函数的不同实现，此时 CPU 很难或者完全不可能预测某个分支的走向。这样 CPU 要么等待控制流进行到足以知道分支走向的方向时，要么干脆猜测——由于此时程序的控制流不可预测——CPU 常常猜错。在这两种情况中，流水线都是空的，CPU 需要等待流水线被新指令填充，这将大幅降低 CPU 的性能，就像图中的画一样。

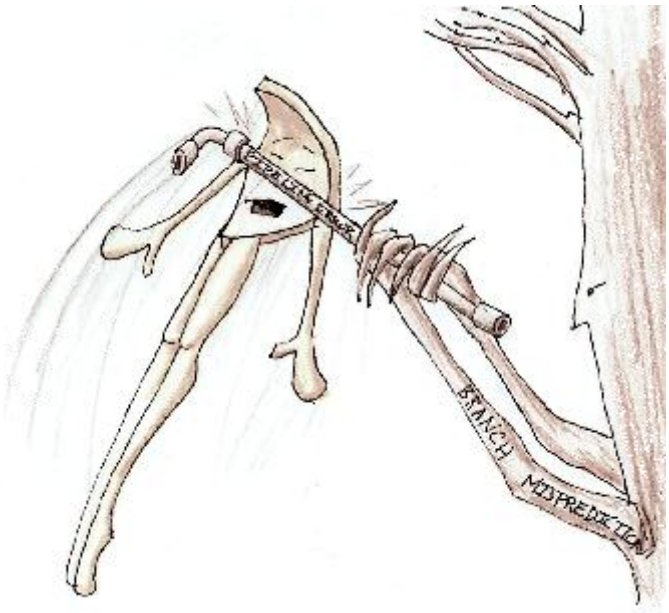


图 2.3 CPU 遭遇 pipeline flush

不幸的是，流水线冲刷并不是唯一影响现代 CPU 运行性能的的障碍。下一节将讲述内存引用带来的危害。

2.1.2. 内存引用

在上世纪 80 年代，微处理器从内存读取一个值的时间比执行一条指令的时间短。在 2006 年，同样是读取内存的时间，微处理器可以在这段时间执行上百条甚至上千条指令。这个差异来源于摩尔定律使得 CPU 性能的增长速度大大超过内存性能的增长速度，也有部分是由于内存大小的增长速度。比如，70 年代的微型计算机通常带有 4KB 主存（是的，是 KB，不是 MB，更别提 GB 了），访问需要一个周期。到 2008 年，CPU 设计者仍然可以构造单周期访问的 4KB 内存，即使是在几 GHz 时钟频率的系统上。事实上这些设计者经常构造这样的内存，但他们现在称呼这种内存为“0 级 cache”。

虽然现代微型计算机上的大型缓存极大地减少了内存访问延迟，但是只有高度可预测的数据访问模式才能让缓存发挥最大效用。不幸的是，一般像遍历链表这样的操作的内存访问模式非常难以预测——毕竟如果这些模式是可预测的，我们也就不会被指针所困扰了，是吧？

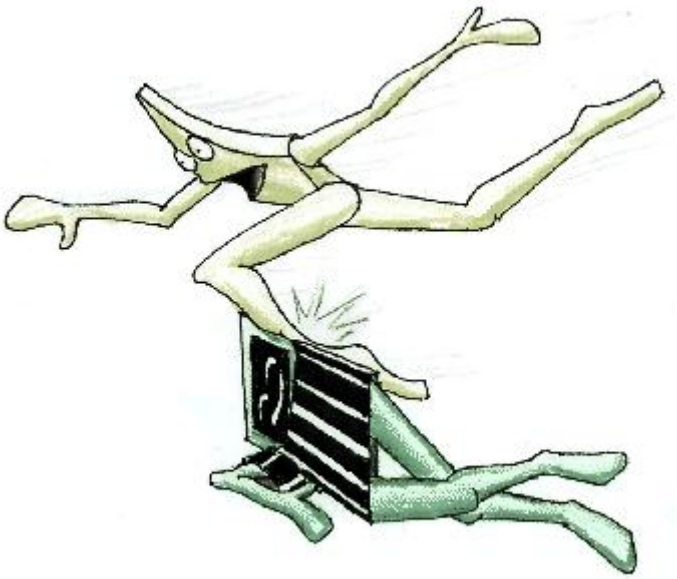


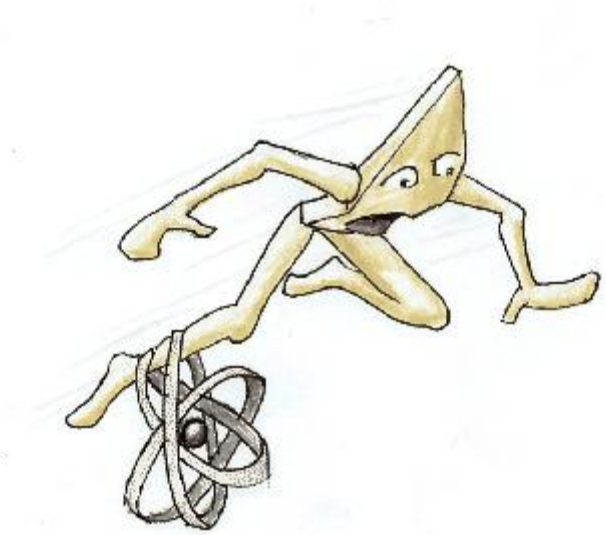
图 2.4 CPU 遭遇内存引用

因此，正如图中显示的，内存引用常常是影响现代 CPU 性能的重要因素。

到现在为止，我们只考虑了 CPU 在单线程代码中执行时会遭遇的性能障碍。多线程会为 CPU 带来额外的性能障碍，我们将在下面的章节中接着讲述。

2.1.3. 原子操作

其中一种障碍是原子操作。原子操作的概念在某种意义上与 CPU 流水线上的一次执行一条的汇编操作冲突了。拜硬件设计者的精密设计所赐，现代 CPU 使用了很多非常聪明的手段让这些操作看起来是原子的，即使这些指令实际上不是原子的。不过即使如此，也还是有一些指令是流水线必须延迟甚至需要冲刷，以便一条原子操作成功完成。



与 2.5 CPU 遭遇原子操作

原子指令对性能的影响见上图。

不幸的是，原子操作通常只用于数据的单个元素。由于许多并行算法都需要在更新多个数据元素时，保证正确的执行顺序，大多数 CPU 都提供了内存屏障。内存屏障也是影响性能的因素之一，下一节将对它进行描述。

小问题 4.2：什么样的机器会允许对多个数据元素进行原子操作？

2.1.4. 内存屏障

内存屏障的更多细节在第 12.2 节和附录 C 中。下面是一个简单的基于锁的临界区。

```
1 spin_lock(&mylock);
2 a = a + 1;
3 spin_unlock(&mylock);
```

如果 CPU 没有按照上述语句的顺序执行，变量”a”会在没有得到“mylock”保护的情况下加一，这肯定和我们取”a”的值的目的不一致。为了防止这种有害的乱序执行，锁操作原语必须包含或显式或隐式的内存屏障。由于内存屏障的作用是防止 CPU 为了提升性能而进行的乱序执行，所以内存屏障几乎一定会降低 CPU 性能，如下图所示。

图 2.6 CPU 遭遇内存屏障

2.1.5. Cache Miss

对多线程程序来说，还有个额外的障碍影响 CPU 性能提升——“Cache Miss”。正如前文提到的，现代 CPU 使用大容量的高速缓存来降低由于较低的内存访问速度带来的性能惩罚。但是，CPU 高速缓存事实上对多 CPU 间频繁访问的变量起反效果。因为当某个 CPU 想去更改变量的值时，极有可能该变量的值刚被其他 CPU 修改过。在这种情况下，变量存在于其他 CPU 而不是当前 CPU 的缓存中，这将导致代价高昂的 Cache Miss（详细内容见附录 C.1 节），如上图所示。

图 2.7 CPU 遭遇 Cache Miss

2.1.6. I/O 操作

缓存未命中可以视为 CPU 之间的 I/O 操作，这应该是代价最低廉的 I/O 操作之一。I/O 操作涉及网络、大容量存储器，或者（更糟的）人类本身，I/O 操作对性能的影响远远大于之前几个章节提到的各种障碍，如下图所示。

图 2.8 CPU 等待 I/O 完成

共享内存式的并行计算和分布式系统式的并行计算的其中一个不同点是，共享内存式并行计算的程序一般不会处理比缓存未命中更糟的情况，而分布式并行计算的程序则很可能遭遇网络通信延迟。这两种情况的延迟都可看作是通信的代价——在串行程序中所没有的代价。因此，通信的开销占执行的实际工作的比率是一项关键设计参数。并行设计的一个主要目标是尽可能的减少这一比率，以达到性能和可扩展性上的目的。

当然，说 I/O 操作属于性能障碍是一方面，说 I/O 操作对性能的影响非常明显则是另一方面。下面的章节将讨论两者的区别。

2.2. 开销

本节将概述前一节列出的性能障碍的实际开销。不过在此之前，需要读者对硬件体系结构有一个粗略认识，下一小节将对该主题进行阐述。

2.2.1. 硬件体系结构

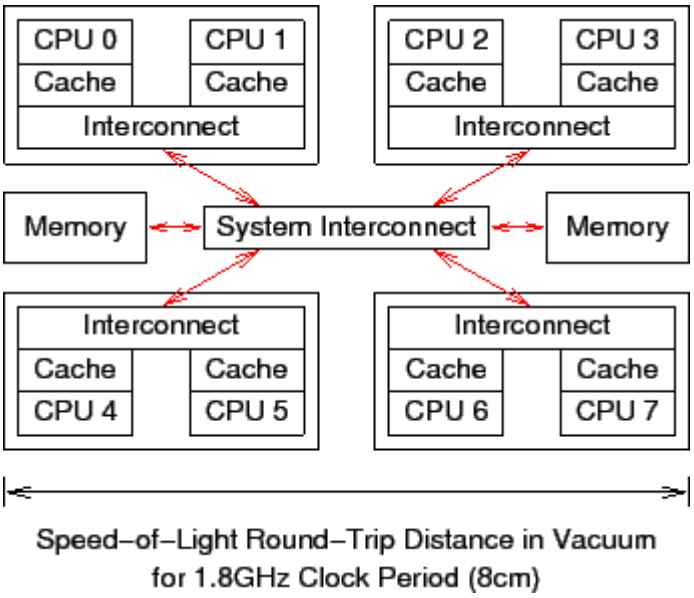


图 2.9 系统硬件体系结构

上图是一个粗略的八核计算机系统概要图。每个管芯有两个 CPU 核，每个核带有自己的高速缓存，管芯内还带有一个互联模块，使管芯内的两个核可以互相通信。在图中央的系统互联模块可以让四个管芯相互通信，并且将管芯与主存连接起来。

数据以“缓存线”为单位在系统中传输，“缓存线”对应于内存中一个 2 的幂大小的字节块，大小通常为 32 到 256 字节之间。当 CPU 从内存中读取一个变量到它的寄存器中时，必须首先将包含了该变量的缓存线读取到 CPU 高速缓存。同样地，CPU 将寄存器中的一个值存储到内存时，不仅必须将包含了该值的缓存线读到 CPU 高速缓存，还必须确保没有其他 CPU 拥有该缓存线的拷贝。

比如，如果 CPU0 在对一个变量执行“比较并交换”（CAS）操作，而该变量所在的缓存线在 CPU7 的高速缓存中，就会发生以下经过简化的事件序列：

1. CPU0 检查本地高速缓存，没有找到缓存线。
2. 请求被转发到 CPU0 和 CPU1 的互联模块，检查 CPU1 的本地高速缓存，没有找到缓存线。
3. 请求被转发到系统互联模块，检查其他三个管芯，得知缓存线被 CPU6 和 CPU7 所在的管芯持有。
4. 请求被转发到 CPU6 和 CPU7 的互联模块，检查这两个 CPU 的高速缓存，在 CPU7 的高速缓存中找到缓存线。
5. CPU7 将缓存线发送给所属的互联模块，并且刷新自己高速缓存中的缓存线。

6. CPU6 和 CPU7 的互联模块将缓存线发送给系统互联模块。
7. 系统互联模块将缓存线发送给 CPU0 和 CPU1 的互联模块。
8. CPU0 和 CPU1 的互联模块将缓存线发送给 CPU0 的高速缓存。
9. CPU0 现在可以对高速缓存中的变量执行 CAS 操作了。

小问题 4.3：这是一个简化后的事件序列吗？还有可能更复杂吗？

小问题 4.4：为什么必须刷新 CPU7 高速缓存中的缓存线？

2.2.2. 操作的开销

The overheads of some common operations important to parallel programs are displayed in Table . This system's clock period rounds to 0.6ns. Although it is not unusual for modern microprocessors to be able to retire multiple instructions per clock period, the operations will be normalized to a full clock period in the third column, labeled ``Ratio". The first thing to note about this table is the large values of many of the ratios. 一些在并行程序中很重要的常见操作开销如上图所示。该系统的时钟周期为 0.6ns。虽然在现代微处理器上每时钟周期 retire 多条指令并不常见，但是在表格的第三列，操作被标准化到了整个时钟周期，称作“比率”。关于上表，第一个需要注意的是比率的值都很大。

Table: Performance of Synchronization Mechanisms on 4-CPU 1.8GHz AMD Opteron 844 System

Operation	Cost (ns)	Ratio
Clock period	0.	1.0
Best-case CAS	37.9	63.2
Best-case lock	65.6	109.3
Single cache miss	139.5	232.5
CAS cache miss	306.0	510.0
Comms Fabric	3,000	5,000
Global Comms	130,000,000	216,000,000

最好情况下的 CAS 操作消耗大概 40 纳秒，超过 60 个时钟周期。这里的“最好情况”是指对某一个变量执行 CAS 操作的 CPU 正好是最后一个操作该变量的 CPU，所以对应的缓存线已经在 CPU 的高速缓存中了，类似地，最好情况下的锁操作（一个“round trip 对”包括获取锁和随后的释放锁）消耗超过 60 纳秒，超过 100 个时钟周期。这里的“最好情况”意味着用于表示锁的数据结构已经在获取和释放锁的 CPU 所属的高速缓存中了。锁操作比 CAS 操作更加耗时，是因

为锁操作的数据结构中需要两个原子操作。

缓存未命中消耗大概 140 纳秒，超过 200 个时钟周期。需要在存储新值时查询变量的旧值的 CAS 操作，消耗大概 300 纳秒，超过 500 个时钟周期。想想这个，在执行一次 CAS 操作的时间里，CPU 可以执行 500 条普通指令。这表明了细粒度锁的局限性。

小问题 4.5：硬件设计者肯定被要求过改进这种情况！为什么他们满足于这些单指令操作的糟糕性能呢？

I/O 操作开销更大。一条高性能（也是高价的）光纤通信，比如 Infiniand 或者其他私有的 interconnects，它的通讯延迟大概有 3 微秒，在这期间 CPU 可以执行 5000 条指令。基于某种标准的通信网络一般需要一些协议的处理，这更进一步增加了延迟。当然，物理距离也会增加延迟，理论上光速绕地球一周需要大概 130 毫秒，这相当于 2 亿个时钟周期。

小问题 4.6：这些数字大的让人发疯！我怎么才能记住它们？

2.3. 硬件的免费午餐?

最近几年并行计算受到大量关注的主要原因是摩尔定律的终结，摩尔定律带来的单线程程序性能提升（或者叫“免费午餐”[Sut08]）也结束了，见第？？页的图。本节简短地介绍一些硬件设计者可能采用的办法，这些办法可以带来某种形式上的“免费午餐”。

不过，前文中概述了一些影响并发性的硬件障碍。其中对硬件设计者来说最严重的一个限制莫过于有限的光速。如第？？的图所示，在一个 1.8GHz 的时钟周期内，光只能在真空中传播大约 8 厘米远。在 5GHz 的时钟周期内，这个距离更是降到 3 厘米。这个距离对于一个现代计算机系统的体积来说，还是太小了点。

更糟糕的是，电子在硅中的移动速度比真空中的光慢 3 到 30 倍，and common clocked logic constructs run still more slowly, 比如，内存引用需要在将请求发送给系统的其他部分前，等待查找本地缓存操作结束。此外，相对低速和高耗电的驱动器需要将电信号从一个硅制管芯传输到另一个管芯，比如 CPU 和主存间的通信。

不过，以下（包括硬件和软件的）技术也许可以改善这一情况：

1. 3D 集成，
2. 新材料和新工艺，
3. 用光来替换电子，
4. 专用加速器，
5. 已有的并行计算软件。

在下面的小节中将会分别介绍这些技术。

2.3.1. 3D 集成

不过，由于时钟逻辑级别造成的延迟是无法通过 3D 集成的方式降低的，并且必须解决诸如生产、测试、电源和散热等 3D 集成中的重大问题。散热问题需要用基于钻石的半导体来解决，钻石是热的良好导体，但却是电的绝缘体。据说生成大型单晶体钻石仍然很困难，更别说将钻石切割成晶圆了。另外，看起来上述这些技术不大可能让性能出现指数级增长，如同某些人习惯的那样（这句翻译的烂啊）。这就是说，还必须走上 Jim Gray 最新的 smoking hairy golf balls。

2.3.2. 新材料和新工艺

据说斯蒂芬·霍金曾经声称半导体制造商面临两个基本问题：（1）有限的光速，（2）物质的原子本质。半导体制造商很有可能已经逼近这两个限制，不过有一些研究报告和开发过程关注于如何规避这两个基本闲置。

其中一个规避物质的原子本质的办法是一种称为“high-K”绝缘体材料，这种材料允许较大的设备模拟较小型设备的电气属性。这种材料存在一些严重的生产困难，但是能让研究的前沿再向前推进一步。另一个比较奇异的规避方法，根据电子可以存在于多个能级之上的事实，在电子中存储多个二进制位。不过这种方法还有待观察，确定能否在生产的半导体设备中稳定工作。

还有一种称为“量子点”的规避方法，使得可以制造体积小得多的半导体设备，不过该方法还处于研究阶段。

虽然光速是一个很难跨越的限制，但是半导体设备更多的是受限于电子移动的速度，而非光速，在半导体材料中移动的电子速度仅是真空中光速的 3%到 30%。在半导体设备间用铜来连接是一种提升电子移动速度的方法，并且出现其他技术让电子移动速度接近光速的可能性是很大的。另外，还有一些实验用微小的光纤作为芯片内和芯片间的传输通道，因为在玻璃中的光速能达到真空中光速的 60%还多。这种方法存在的一个问题是光电/电光转换的效率，会产生电能消耗和散热问题。

这也就是说，除开在物理学领域的基础性进展以外，任何让数据流传输速度出现指数级增长的想法都受限于真空中的光速。

2.3.3. 专用加速器

用通用 CPU 来处理某个专门问题，通常会将大量的时间和能源消耗在 only

tangentially related to the problem at hand.比如，当对一对向量进行点积操作时，通用 CPU 一般会使用一个带循环计数的循环（一般是展开的）。对指令解码、增加循环计数、测试计数和跳转回循环的开始处，这些操作在某种意义上来说都是无用功：真正的目标是计算两个向量对应元素的乘积。因此，在硬件上设计一个专用于向量乘法的部件会让这项工作做的既快速又节能。

这就是在很多商用微处理器上出现向量指令的原因。这些指令可以同时操作多个数据，能让点积计算减少解码指令消耗和循环开销。

类似的，专用硬件可以更有效地进行加密/解密、压缩/解压缩、编码/解码和许多其他的任务。不过不幸的是，这种效率提升不是免费的。包含特殊硬件的计算机系统需要更多的晶体管，即使在不用时也会带来能源消耗。软件也必须进行修改以利用专用硬件的长处，同时这种专门硬件又必须足够通用，这样高昂的 up-front 设计费用才能摊到足够多的用户身上，让专用硬件的价钱变得可以承受。部分由于以上经济考虑，专用硬件到目前为止只在几个领域出现，包括图形处理（GPU），矢量处理器（MMX、SSE 和 VMX 指令），以及相对规模较小的加密领域。

不过，随着摩尔定律带来的单线程性能提升的结束，我们可以安全的预测：未来各种各样的专用硬件会大大增加。

2.3.4. 现有的并行软件

虽然多核 CPU 曾经让计算机行业惊讶了一把，但是事实上基于共享内存的并行计算机已经商用了超过半个世纪了。这段时间足以让一些重要的并行软件登上舞台，事实也确实如此。并行操作系统已经非常常见了，比如并行线程库，并行关系数据库管理系统和并行数值计算软件。这些现有的并行软件在解决我们可能遇见的并行难题上已经走了很长一段路。

也许最常见的例子是并行关系数据库管理系统。它和单线程程序不同，并行关系数据库管理系统一般用高级脚本语言书写，并发地访问位于中心的关系数据库。在现在的高度并行化系统中，只有数据库是真正需要直接处理并行化的。因此它运用了许多非常好的技术。

2.4. 软件设计 Implication

表 2.1 上的比率值至关重要，因为它们限制了并行计算程序的效率。为了弄清这点，我们假设一款并行计算程序使用了 CAS 操作来进行线程间通信。假设进行通信的线程不是与自身而主要是与其他线程通信，那么 CAS 操作一般会涉及到缓存未命中。进一步假设对应每个 CAS 通信操作的工作需要消耗 300 纳秒，

这足够执行几个浮点 transendental 函数了。其中一半的执行时间消耗在 CAS 通信操作上！这也意味着有两个 CPU 的系统运行这样一个并行程序的速度，并不比单 CPU 系统运行一个串行执行程序的速度快。

在分布式系统中结果还会更糟糕，因为单次通信操作的延迟时间可能和几千条甚至上百万条浮点操作的时间一样长。这就说明了会产生大量工作量的通信操作应该尽可能减少（通顺吗？）

小问题 2.7：既然分布式系统的通信操作代价如此昂贵，为什么人们还要使用它？

这一课应该非常明了：并行算法必须将每个线程设计成尽可能独立运行的线程。越少使用线程间通信手段，比如原子操作、锁或者其它消息传递方法，应用程序的性能和可扩展性就会更好。简而言之，想要达到优秀的并行性能和可扩展性，就意味着在并行算法和实现中挣扎，小心的选择数据结构和算法，使用现有的并行软件和环境，或者将并行问题转换成已经有并行解决方案存在的问题。

下一章将讨论可以提升性能和可扩展性的设计纪律。

3. 工具

本章简要地介绍了一些并行编程领域的基本工具，主要是类 Linux 系统上的应用程序可以使用的工具。3.1 节讲述了脚本语言，3.2 节讲述了 POSIX API 支持的多进程虚拟化和 POSIX 线程，最后 3.3 节讲述了原子操作。

请注意本章只是提供了概要性的介绍。更详细的内容请见参考文献，后续章节将会讲述如何更好地使用这些工具。

3.1. 脚本语言

Linux shell 脚本语言用一种简单而又有效的方法处理并行化。比如，假设你有一个叫做 compute_it 的程序，你需要用不同的参数运行两次。那么只需要这样写：

```
1 compute_it 1 > compute_it.1.out &
2 compute_it 2 > compute_it.2.out &
3 wait
4 cat compute_it.1.out
5 cat compute_it.2.out
```

第 1 行和第 2 行启动了程序的两个实例，通过&指定两个实例在后台运行，分别将程序输出重定向到一个文件。第 3 行等待两个实例执行完毕，第 4 行和第 5 行显示程序的输出。执行结果如图 3.1 所示：Compute_it 的两个实例并行执行，wait 在操作执行完毕后返回，然后顺序执行 cat 操作显示结果。

小问题 3.1：可是这个愚蠢透顶的 shell 脚本并不是真正的并行程序！这些垃圾有什么用？？

小问题 3.2：有没有更简单的方法创建并行 shell 脚本？如果有，怎么写？如果没有，为什么？

另一个例子，make 脚本语言提供了一个-j 选项，可以指定在构建过程中同时执行多少个 job。比如，键入 make -j4 来构建 Linux 内核，代表最多可以同时执行 4 个并行编译过程。

希望这些简单的例子能够让你相信，并行编程并不总是那么复杂或者困难。

小问题 3.3：但是如果基于脚本的并行编程这么简单，为什么还需要其他的东西呢？

3.2. POSIX 多进程

本节浅入浅出地介绍了 POSIX 环境，包括广泛应用的 pthreads[Ope97]。3.2.1 节介绍了 POSIX 的 fork()和相关原语，3.2.2 节介绍了线程创建和撤销，3.2.3 节介绍了 POSIX locking 机制，最后的 3.4 节展示了 Linux 内核中的类似操作。

3.2.1. POSIX 进程创建和撤销

进程通过 fork()原语创建，使用 kill()原语撤销，也可以用 exit()原语自我撤销。执行 fork()的进程被称为新创建进程的“父进程”。父进程可以通过 wait()原语等待子进程的执行完毕。

请注意，本节中的例子都极其简单。真实世界的应用使用这些原语时，还可能需要操纵信号、文件描述符、共享内存或者其他各种资源。另外，有些在某个子进程终止时还会执行特定操作的应用程序，还可能要关心子进程终止的原因。这些情况下的判断逻辑显然会增加代码的复杂性。想要获得更多的信息，请看任意一本[Ste92]的教科书。

```
1 pid = fork();
2 if (pid == 0) {
3     /* child */
4 } else if (pid < 0) {
5     /* parent, upon error */
6     perror("fork");
7     exit(-1);
8 } else {
9     /* parent, pid == child ID */
10 }
```

图 3.2：使用 fork()原语

如果 fork()执行成功会返回两次，一次是父进程，另一次是子进程。fork()的返回值可以让调用者区分出这两种情况，如图 3.2 所示(forkjoin.c)。第 1 行执行 fork()原语，用本地变量 pid 存储返回值。第 2 行检查 pid 是否为 0，如果为 0 则是子进程，并继续执行第 3 行。如上文所说的那样，子进程也可以调用 exit()原语来终止。如果小于 0，则是父进程返回错误，第 4 行检查 fork()调用返回的错误码，在第 5-7 行打印错误并退出。如果 pid 大于 0，则 fork()成功执行，父进程执行第 9 行，此时的 pid 包含子进程的进程 ID 号。

```
1 void waitall(void)
```

```
2 {
3     int pid;
4     int status;
5
6     for (;;) {
7        pid = wait(&status);
8        if (pid == -1) {
9           if (errno == ECHILD)
10          break;
11          perror("wait");
12          exit(-1);
13       }
14    }
15 }
```

图 3.3：使用 wait()原语

父进程还可以通过 wait()原语来等待子进程执行完毕。但是，使用该原语要比在 shell 脚本中使用它复杂一些，因为每次调用 wait()只能等待一个子进程。我们可以将 wait()封装成一个类似 waitall()的函数，如图 3.3 所示(api-pthread.h)，这个 waitall()函数的语义与 shell 脚本中的 wait 一样。第 6-15 行的循环每执行一次，就等待一个子进程执行完毕一次。第 7 行调用 wait()原语开始阻塞直到一个子进程退出，它返回子进程的进程 ID 号。如果该进程号为-1，那么说明 wait()无法等待子进程执行完毕。接着第 9 行检查错误码 errno 是否为 ECHILD，这代表没有其他子进程了，所以第 10 行退出循环。不然的话，第 11 和第 12 行打印错误并退出。

小问题 3.4: wait()原语有必要这么复杂吗？为什么不让它像 shell 脚本的 wait 一样呢？

```
1 int x = 0;
2 int pid;
3
4 pid = fork();
5 if (pid == 0) { /* child */
6     x = 1;
7     printf("Child process set x=1\n");
8     exit(0);
9 }
```

```
10 if (pid < 0) { /* parent, upon error */
11    perror("fork");
12    exit(-1);
13 }
14 waitall();
15 printf("Parent process sees x=%d\n", x);
```

图3.4：通过fork()创建的进程不共享内存

读者请注意：父进程和子进程并不共享内存，这很重要，如图 3.4 程序（forkjoinvar.c）所示。子进程在第 6 行为全局变量 x 赋 1，在第 7 行打印一条消息，在第 8 行退出。父进程从第 14 行开始执行，等待子进程结束，在第 15 行打印，发现全局变量 x 的值还是 0。输出如下：

```
Child process set x=1
Parent process sees x=0
```

小问题 3.5：fork()和 wait()还有什么这里没讲的用法吗？

最细粒度的并行化需要共享内存，这是 3.2.2 节将要讲述的。共享内存式的并行化可要比 fork-join 式的并行化复杂的多。

3.2.2. POSIX 线程的创建和撤销

```
1 int x = 0;
2
3 void *mythread(void *arg)
4 {
5     x = 1;
6     printf("Child process set x=1\n");
7     return NULL;
8 }
9
10 int main(int argc, char *argv[])
11 {
12    pthread_t tid;
13    void *vp;
14
15    if (pthread_create(&tid, NULL, mythread, NULL) !=
0) {
16       perror("pthread_create");
```

```
17        exit(-1);
18    }
19    if (pthread_join(tid, &vp) != 0) {
20        perror("pthread_join");
21        exit(-1);
22    }
23    printf("Parent process sees x=%d\n", x);
24    return 0;
25 }
```

图 3.5：通过 pthread_create()创建的线程要共享内存

在一个已有的进程中创建线程，需要调用 pthread_create()原语，比如图 3.5 中第 15 行所示（pcreate.c）。pthread_create()的第一个参数是指向 pthread_t 类型的指针，用于存放将要创建线程的线程 ID 号，第二个 NULL 参数是一个可选的指向 pthread_attr_t 结构的指针，第三个参数是新线程将要调用的函数（在本例中是 mythread()），最后一个 NULL 参数是传递给 mythread()的参数。

在这个例子中，mythread()直接就 return 了，但是它也可以选择调用 pthread_exit()结束。

小问题 3.6：如果图 3.5 中的 mythread()可以直接返回，为什么还要用 pthread_exit()?

第 19 行的 pthread_join()原语是对 fork-join 中的 wait()的模仿，它一直阻塞到 tid 变量指定的线程返回。线程返回有两种方式，要么调用 pthread_exit()返回，要么通过线程的顶层函数返回。线程的返回值存放在 pthread_join()的第二个指针类型参数中。线程的返回值要么是传给 pthread_exit()的返回值，要么是线程顶层函数返回的值，这取决于问题中的线程如何退出。

图 3.5 中的程序显示的输出如下，说明了一个事实：线程之间共享内存。

```
Child process set x=1
Parent process sees x=1
```

请读者注意，这个程序小心地构造出一个场景，确保一次只有一个线程为变量 x 赋值。任何一个线程为某变量赋值而另一线程读取该变量的值的场景，都会产生一种被称为“data race”的情况。因为 C 语言并不保证出现 data race 时结果的合理性，所以我们需要一些手段来安全地并发读取数据，比如下一节将会提到的加锁原语。

小问题 3.7：如果 C 语言在 data race 时不做任何保证，为什么 Linux 内核还会有那么多 data race 呢？你是准备告诉我 Linux 内核就是个破烂玩意儿吗？？

3.2.3. POSIX 锁

POSIX 规范允许程序员使用“POSIX 锁”来避免 data race。POSIX 锁包括几个原语，其中最基础的是 pthread_mutex_lock()和 pthread_mutex_unlock()。这些原语操作类型为 pthread_mutex_t 的锁。该锁的静态声明和初始化由 PTHREAD_MUTEX_INITIALIZER 完成，或者由 pthread_mutex_init()原语来动态分配并初始化。本节的示例代码将采用前者。

pthread_mutex_lock()原语“获取”一个指定的锁，pthread_mutex_unlock()原语“释放”一个指定的锁。因为这些原语是互相排斥的加锁/解锁原语，所以一次只能有一个线程在一个特定时刻“持有”一把特定的锁。比如，如果一对线程尝试同时获取同一把锁，那么其中一个线程会先“获准”持有该锁，另一个线程只能等待第一个线程释放该锁。

小问题 3.8：如果我想让多个线程同时获取同一把锁会发生什么？

```
1 pthread_mutex_t lock_a = PTHREAD_MUTEX_INITIALIZER;
2 pthread_mutex_t lock_b = PTHREAD_MUTEX_INITIALIZER;
3 int x = 0;
4
5 void *lock_reader(void *arg)
6 {
7     int i;
8     int newx = -1;
9     int oldx = -1;
10    pthread_mutex_t *pmlp = (pthread_mutex_t *)arg;
11
12    if (pthread_mutex_lock(pmlp) != 0) {
13       perror("lock_reader:pthread_mutex_lock");
14       exit(-1);
15    }
16    for (i = 0; i < 100; i++) {
17       newx = ACCESS_ONCE(x);
18       if (newx != oldx) {
19          printf("lock_reader(): x = %d\n", newx);
20       }
21       oldx = newx;
22       poll(NULL, 0, 1);
```

```
23    }
24    if (pthread_mutex_unlock(pmlp) != 0) {
25       perror("lock_reader:pthread_mutex_unlock");
26       exit(-1);
27    }
28    return NULL;
29 }
30
31 void *lock_writer(void *arg)
32 {
33    int i;
34    pthread_mutex_t *pmlp = (pthread_mutex_t *)arg;
35
36    if (pthread_mutex_lock(pmlp) != 0) {
37       perror("lock_reader:pthread_mutex_lock");
38       exit(-1);
39    }
40    for (i = 0; i < 3; i++) {
41       ACCESS_ONCE(x)++;
42       poll(NULL, 0, 5);
43    }
44    if (pthread_mutex_unlock(pmlp) != 0) {
45       perror("lock_reader:pthread_mutex_unlock");
46       exit(-1);
47    }
48    return NULL;
49 }
```

图3.6：互斥锁代码示例

图 3.6 所使用的代码（lock.c）展示了这种互相排斥的加锁/解锁特性。第 1 行定义并初始化了一个名为 lock_a 的 POSIX 锁，第 2 行又定义了一个类似的锁 lock_b。第 3 行定义并初始化了一个全局变量 x。

第 5-28 行定义了函数 lock_reader()，在持有 arg 指定的锁后重复读取全局变量 x 的值。第 10 行将 arg 转换成一个指向 pthread_mutex_t 的指针，该指针随后被传给 pthread_mutex_lock()和 pthread_mutex_unlock()作为参数。

小问题 3.9：为什么不直接将第 5 行 lock_reader()的参数直接弄成指向

pthread_mutex_t 的指针？

第 12-15 行获取了指定的 pthread_mutex_t 锁，检查错误值，如果有错则程序退出。第 16-23 行重复检查 x 的值，如果值发生改变就将新值打印出来。第 22 行随眠了 1 个毫秒，让这个示例代码能在单核计算机上运转良好。第 24-27 行释放 pthread_mutex_t 锁，接着再一次检查错误值，如果有错误则程序退出。最后，第 28 行返回 NULL，满足 pthread_create()所要求的返回值类型。

小问题 3.10：每次获取和释放一个 pthread_mutex_t 都要写四行代码！有没有什么办法能减少这种折磨呢？？

图 3.6 的第 31-49 行是 lock_writer()函数，在持有指定的 pthread_mutex_t 后周期性地更新全局变量 x 的值。和 lock_reader()一样，第 34 行将 arg 转换成指向 pthread_mutex_t 的指针，第 36-39 行获取指定的锁，第 44-47 行释放这把锁。当持有这把锁时，地道 40-48 行增加全局变量 x 的值，每次增加后都睡眠 5 个毫秒。

```
1 printf("Creating two threads using same lock:\n");
2 if (pthread_create(&tid1, NULL,
3     lock_reader, &lock_a) != 0) {
4     perror("pthread_create");
5     exit(-1);
6 }
7  if (pthread_create(&tid2, NULL,
8     lock_writer, &lock_a) != 0) {
9     perror("pthread_create");
10    exit(-1);
11 }
12 if (pthread_join(tid1, &vp) != 0) {
13    perror("pthread_join");
14    exit(-1);
15 }
16 if (pthread_join(tid2, &vp) != 0) {
17    perror("pthread_join");
18    exit(-1);
19 }
```

图 3.7：使用相同互斥锁的代码示例

图 3.7 显示了一段执行 lock_reader()和 lock_writer()的代码片段，两个线程都使用同一把锁，lock_a。第 2-6 行创建一个执行 lock_writer()的线程，第 7-11 行创建一个执行 lock_reader()的线程。第 12-19 行等待两个线程返回。该代码片段

的输出如下：

```
Creating two threads using same lock:
lock_reader(): x = 0
```

因为两个线程都使用同一把锁，lock_reader()线程无法看到 lock_writer()线程在持有锁时产生的任何变量 x 的中间值。

小问题 3.11：“x = 0”是图 3.7 所示代码片段的唯一可能输出吗？如果是，为什么？如果不是，还可能输出什么，为什么？

```
1    printf("Creating two threads w/different
locks:\n");
2    x = 0;
3    if (pthread_create(&tid1, NULL,
4                lock_reader, &lock_a) != 0) {
5       perror("pthread_create");
6       exit(-1);
7    }
8    if (pthread_create(&tid2, NULL,
9                lock_writer, &lock_b) != 0) {
10       perror("pthread_create");
11       exit(-1);
12    }
13    if (pthread_join(tid1, &vp) != 0) {
14       perror("pthread_join");
15       exit(-1);
16    }
17    if (pthread_join(tid2, &vp) != 0) {
18       perror("pthread_join");
19       exit(-1);
20    }
```

图3.8：使用不同互斥锁的代码示例

图3.8展示了一块类似的代码片段，只不过这次用的是不同的锁：lock_reader()线程用 lock_a，lock_writer()线程用 lock_b。这块代码片段的输出如下：

```
Creating two threads w/different locks:
lock_reader(): x = 0
lock_reader(): x = 1
lock_reader(): x = 2
```

```
lock_reader(): x = 3
```

由于两个线程使用不同的锁，它们并不能相互互斥，因此可以同时执行。lock_reader()这下可以看见 lock_writer()存储的全局变量 x 的中间状态了。

小问题 3.12：使用不同的锁可能产生很多混乱，比如线程之间可以看到对方的中间状态。所以是否一个编写很好的并行程序应该限制自身只使用相同的锁，以避免这种混乱？

小问题 3.13：在图 3.8 所示代码中，lock_reader()能保证看见所有 lock_writer()产生的中间值吗？如果能，为什么能？如果不能，为什么不能？

小问题 3.14：等等!! 图 3.7 里没有初始化全局变量 x，为什么图 3.8 里要去初始化它？？

虽然 POSIX 互斥锁还有很多内容，但是本节介绍的原语已经提供了一个美好的起点，在绝大多数情况下这些原语都足够了。下一节将简要介绍 POSIX 的读写锁。

3.2.4. POSIX 读写锁

POSIX API 提供了一种读写锁，用 pthread_rwlock_t 类型来表示。和 pthread_mutex_t 一样，pthread_rwlock_t 也可以由 PTHREAD_RWLOCK_INITILIZER 静态初始化，或者由 pthread_rwlock_init()原语动态初始化。pthread_rwlock_rdlock()原语获取 pthread_rwlock_t 的读锁，pthread_rwlock_wrlock()获取它的写锁，pthread_rwlock_unlock()原语负责释放锁。在任意时刻只能有一个线程持有给定 pthread_rwlock_t 的写锁，但同时可以有多个线程持有给定 pthread_rwlock_t 的读锁，至少在没有线程持有写锁时是如此。

正如读者期望的那样，读写锁是专门为大多数读的情况设计的。在这种情况中，读写锁可以提供比互斥锁大得多的扩展性，因为互斥锁从定义上已经限制了任意时刻只能有一个线程持有锁，而读写锁允许任意多数目的读者线程同时持有读锁。不过我们需要知道读写锁到底增加了多少可扩展性。

```
1  pthread_rwlock_t  rwl  = PTHREAD_RWLOCK_INITIALIZER;
2  int  holdtime  =  0;
3  int  thinktime  =  0;
4  long  long  *readcounts;
5  int  nreadersrunning  =  0;
6
7  #define  GOFLAG_INIT  0
8  #define  GOFLAG_RUN    1
9  #define  GOFLAG_STOP  2
```

```
10 char goflag = GOFLAG_INIT;
11
12 void *reader(void *arg)
13 {
14   int i;
15   long long loopcnt = 0;
16   long me = (long)arg;
17
18   __sync_fetch_and_add(&nreadersrunning, 1);
19   while (ACCESS_ONCE(goflag) == GOFLAG_INIT) {
20     continue;
21   }
22   while (ACCESS_ONCE(goflag) == GOFLAG_RUN) {
23     if (pthread_rwlock_rdlock(&rwl) != 0) {
24       perror("pthread_rwlock_rdlock");
25       exit(-1);
26     }
27     for (i = 1; i < holdtime; i++) {
28       barrier();
29     }
30     if (pthread_rwlock_unlock(&rwl) != 0) {
31       perror("pthread_rwlock_unlock");
32       exit(-1);
33     }
34     for (i = 1; i < thinktime; i++) {
35       barrier();
36     }
37     loopcnt++;
38   }
39   readcounts[me] = loopcnt;
40   return NULL;
41 }
```

图3.9：衡量读写锁可扩展性的代码示例

图 3.9（rwlockscale.c）显示了一种衡量读写锁可扩展性的方法。第 1 行是读写锁的定义和初始化，第 2 行的 holdtime 参数控制每个线程持有读写锁的时间，

第 3 行的 thinktime 参数控制释放读写锁和下一次获取读写锁之间的间隔，第 4 行定义了 readcounts 数组，每个读者线程将获取锁的次数放在里面，第 5 行定义了变量 nreadersruning，用于控制所有的读者线程何时开始执行。

第 7-10 行定义了 goflag，用于同步测试的开始和结束。goflag 的初始值为 GOFLAG_INIT，当所有读者线程都启动后，设置为 GOFLAG_RUN，最后设置为 GOFLAG_STOP 来终止测试程序运行。

第 12-41 行定义了 reader()，也就是读者线程。第 18 行原子的增加 nreadersrunning 的值，用来表示线程现在正在运行，第 19-21 行等待测试开始。ACCESS_ONCE()原语强迫编译器在每次循环中都去取 goflag 的值——否则编译器会拿起它的权利假定 goflag 的值没有改变。

第 22-38 行的循环执行性能测试。第 23-26 行获取锁，第 27-29 行在一段指定的时间间隔内持有锁（barrier()指令阻止编译器优化循环代码），第 30-33 行释放锁，第 34-36 行在重新获取锁前等待一段指定的时间间隔。第 37 行统计锁的获取次数。

第 38 行将获取锁次数的统计值放入 readcounts[]数组对应本线程的元素中，第 40 行返回，结束本线程。

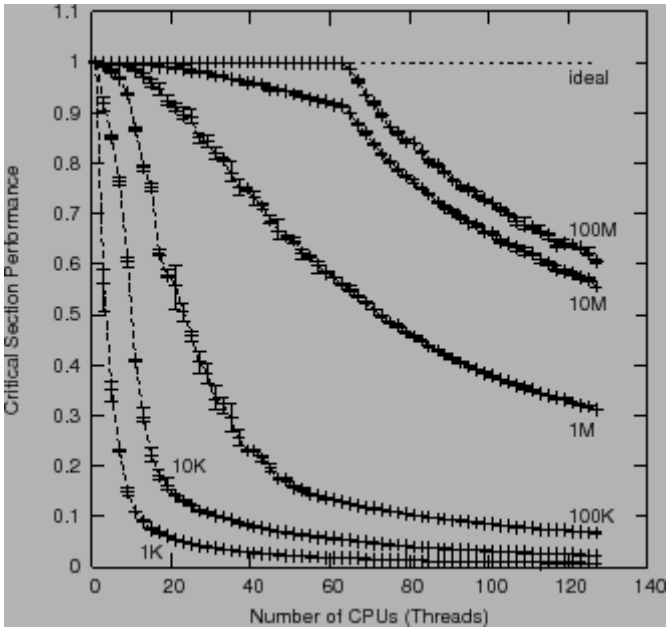


图 3.10：读写锁的可扩展性

图 3.10 是测试的结果，在每个核带两个硬件线程的 64 核 Power-5 系统上执行，整个系统有 128 个软件可见的 CPU。在所有测试中 thinktime 参数都为 0，holdtime 参数的取值从 1000（图中的“1K”）到 1 亿（图中的“100M”）。图中绘制的值的计算公式是：

$$\frac{L_N}{NL_1}$$

N 是线程数，LN 是 N 个线程获取锁的次数，L1 是单个线程获取锁得次数。

在理想的硬件和软件扩展性条件下，该公式计算出的值应该一直为 1.0。

正如在图中见到的，读写锁的可扩展性显然说不上理想，临界区较小时尤其如此。为什么读锁的获取这么慢呢，这应该是由于所有想获取读锁的线程都要更新 pthread_rwlock_t 的数据结构。因此，一旦全部 128 个线程同时尝试获取读写锁的读锁，那么这些线程必须一个一个的更新读锁中的 pthread_rwlock_t 结构。最幸运的线程可能几乎立刻就获取读锁了，最倒霉的线程则必须等待其他 127 个线程更新后才能获取读锁。增加 CPU 只能让这种情况变得更糟。

小问题 3.15：Isn’t comparing against single-CPU throughput a bit harsh?

小问题 3.16：可是 1000 条指令对于临界区来说已经不算小了。如果我想用一个比这小得多的临界区，比如只有几十条指令的，那么我该怎么做？

小问题 3.17：在图 3.10 中，除了 100M 以外的其他曲线都和理想曲线有相当偏差。相反，100M 曲线在 64 个 CPU 时开始大幅偏离理想曲线。另外，100M 曲线和 10M 曲线之间的间隔远远小于 10M 曲线和 1M 曲线之间的间隔。和其他曲线相比，为什么 100M 曲线如此不同？

小问题 3.18：Power 5 已经是几年前的机器了，现在的硬件运行地更快。那么还有什么担心读写锁缓慢的必要吗？

尽管存在这些不足，在很多情况下读写锁仍然十分有用，比如当读者必须进行高延迟的文件或者网络 I/O 时。第四章和第八章将介绍一些读写锁的替代者。

3.3. 原子操作

图 3.10 显示，读写锁在临界区最小时开销最大，考虑到这一点，那么最好能有其他手段来保护极其短小的临界区。我们已经见过一种原子操作了，图 3.9 中第 18 行的__sync_fetch_and_add()原语。该原语自动将它的第二个参数增加到它第一个参数所引用的值上去，返回其原值（在 3.9 的例子中被忽略了）。如果有一对线程并发地对同一个变量执行__sync_fetch_and_add()，变量的值将会包括两次相加的结果。

gcc 编译器提供了许多附加的原子操作，包括__sync_fetch_and_sub()、__sync_fetch_and_or()、__sync_fetch_and_and()、__sync_fetch_and_xor()和__sync_fetch_and_nand()原语，这些操作都返回参数的原值。如果你一定需要变量的新值，可以使用__sync_add_and_fetch()、__sync_sub_and_fetch()、__sync_or_and_fetch()、__sync_and_and_fetch()、__sync_xor_and_fetch()和__sync_nand_and_fetch()原语。

小问题 3.19：这一套原语真的有必要存在吗？

有一对原语提供了经典的“比较并交换”操作，

__sync_bool_compare_and_swap()和__sync_val_compare_and_swap()。当变量的原值与指定的 oldval 相等时，这两个原语自动将 newval 写给指定变量。bool 版本的原语在操作成功时返回 1，操作失败时返回 0，比如变量原值和指定的 oldval 不相等时。val 版本的原语在变量的原值等于指定的 oldval 时，返回变量的原值，表明操作执行成功。任何对单一变量进行操作的原子操作都可以用“比较并交换”的方式实现，从这种意义上说，上述两个“比较并交换”的操作是 universal 的，虽然 bool 版本的原语在应用中效率更高。“比较并交换”操作通常可以作为其他原子操作的基础，不过这些原子操作通常存在复杂性、可扩展性和性能等诸方面问题。[Her90]

__sync_synchronize()原语是一个“内存屏障”，它限制编译器和 CPU 对指令乱序执行的优化，详见第 12.2 节的讨论。在某些情况下，只限制编译器对指令的优化就足够了，CPU 的优化可以保留，此时就需要使用 barrier()原语，就像图 3.9 中第 28 行那样。在某些情况下，只需要让编译器不优化某个内存访问就行了，此时可以使用 ACCESS_ONCE()原语，就像图 3.6 中第 17 行那样。后两个原语不是由 gcc 直接提供的，但是可以像下面这样直接实现：

```
#define ACCESS_ONCE(x) (*(volatile typeof(x) *)&(x))
#define barrier() __asm__ __volatile__("": : : "memory")
```

小问题 3.20：既然这些原子操作通常都会生成指令集直接支持的单个原子指令，那么它们是不是最快办法呢？

3.4. Linux 内核中类似 POSIX 的操作

不幸的是，远在各种标准委员会出现之前，线程操作，加锁、解锁原语和原子操作就已经存在了。因此，这些操作有很多种变体。用汇编语言实现这些操作也十分常见，不仅因为历史原因，还因为可以在某些特定场合获得更好的性能。比如，gcc 的__sync_族原语都是提供 memory-ordering 的语义，这激励许多程序员实现自己的函数，来满足许多不需要 memory ordering 语义的场景。

***表 3.1:** POSIX 原语与 Linux 内核函数对应表*

Category	POSIX	Linux Kernel
Thread Management	pthread_t	struct task_struct
	pthread_create()	kthread_create

	pthread_exit()	kthread_should_stop() (rough)
	pthread_join()	kthread_stop() (rough)
	poll(NULL, 0, 5)	schedule_timeout_interruptible()
POSIX Locking	pthread_mutex_t	spinlock_t (rough)
		struct mutex
	PTHREAD_MUTEX_INITIALIZER	DEFINE_SPINLOCK()
		DEFINE_MUTEX()
	pthread_mutex_lock()	spin_lock() (and friends)
		mutex_lock() (and friends)
	pthread_mutex_unlock()	spin_unlock() (and friends)
		mutex_unlock()
POSIX Reader-Writer	pthread_rwlock_t	rwlock_t (rough)
Locking		struct rw_semaphore
	PTHREAD_RWLOCK_INITIALIZER	DEFINE_RWLOCK()
		DECLARE_RWSEM()
	pthread_rwlock_rdlock()	read_lock() (and friends)
		down_read() (and friends)
	pthread_rwlock_unlock()	read_unlock() (and friends)
		up_read()
	thread_rwlock_wrlock()	write_lock() (and friends)
		down_write() (and friends)
	pthread_rwlock_unlock()	write_unlock() (and friends)
		up_write()
Atomic Operations	C Scalar Types	atomic_t
		atomic64_t
	__sync_fetch_and_add()	atomic_add_return()
		atomic64_add_return()

	__sync_fetch_and_sub()	atomic_sub_return()
		atomic64_sub_return()
	__sync_val_compare_and_swap()	cmpxchg()
	__sync_lock_test_and_set()	xchg() (rough)
	__sync_synchronize()	smp_mb()

因此，表 3.1 中提供了一个粗略的对应关系，比较 POSIX、gcc 原语和 Linux 内核中使用的版本。精准的对应关系很难给出，因为 Linux 内核有各种各样的加锁、解锁原语，gcc 则有很多 Linux 内核中不能直接使用的原子操作。当然，一方面，用户态的代码不需要 Linux 内核中各种类型的加锁、解锁原语，同时另一方面，gcc 的原子操作也可以直接用 cmpxchg()来模拟。

小问题 3.21：Linux 内核中对应 fork()和 join()的是什么？

3.5. 趁手的工具——该如何选择？

根据经验定律，应该在能完成工作的工具中选择最简单的一个。如果可以，尽量串行编程。如果这还不够，那么使用 shell 脚本来实现并行化。如果 shell 脚本的 fork()/exec()开销（在 Intel 双核笔记本中最简单的 C 程序需要大概 480 毫秒）太大，那么使用 C 语言的 fork()和 wait()原语。如果这些原语的开销也太大（最小的子进程也需要 80 毫秒），那么你可能需要用 POSIX 线程库原语，选择合适的加锁、解锁原语和/或者原子操作。如果 POSIX 线程库原语的开销仍然太大（一般低于毫秒级），那么就需要使用第 8 章介绍的原语了。永远记住，进程内的通信和消息传递总是比共享内存的多线程执行要好。

当然，实际的开销并不只取决于你的硬件，更重要的是你如何使用这些原语。因此，做正确的设计和选择正确的原语是非常有必要的，下一章花了大量篇幅讨论这一问题。

4. 计数

计算机能做的事情里，计数也许是最简单也是最自然的了。不过在一台大型的共享内存的多处理器系统上高效并且 scalably 的计数，仍然具有相当的挑战性。更进一步，计数背后隐含概念的简单性使得我们可以探索并发中的基本问题，而无需被繁复的数据结构或者复杂的同步原语干扰。因此，计数是并行编程的极佳切入对象。

本章涵盖了许多适用于特殊情况的简单、快速并且可扩展的计数算法。但是首先，让我们先看看您度低并发计数知道多少？

小问题 4.1：到底是什么让即高效又可扩展的计数这么难？毕竟计算机有专门的硬件只负责计数、加法、减法这些，难道没用吗？

小问题 4.2：网络报文计数问题。假设你需要收集对接收或者发送的网络报文数目（或者总的字节数）的统计。报文可能被系统中任意一个 CPU 接收或者发送。我们更进一步假设一台大型计算机可以在一秒钟处理一百万个报文，而有一个监控系统报文的程序每 5 秒钟读一次计数。你准备如何实现这个计数？

小问题 4.3：粗略的结构分配数目限制值问题。假设你需要维护一个已分配结构数目的计数，这个计数用来防止该结构分配的数目超出限制（比如 10000 个）。我们更进一步假设这些结构的生命周期很短，很不容易超出限制，有一个“马马虎虎的”限制值就可以了。

小问题 4.4：精确的结构分配数目限制值问题。假设你需要维护一个已分配结构数目的计数，这个计数用来防止该结构分配的数目超出限制（比如 10000 个）。我们更进一步假设这些结构的生命周期很短，很不容易超出限制，并且几乎在所有时间都至少有一个结构在使用中。我们再进一步假设我们需要精确地知道何时计数值变成 0，比如，除非还有至少一个结构在使用，否则可以释放一些不再需要的内存。

小问题 4.5：可移除 I/O 设备的访问计数问题。假设你需要维护一个频繁使用的可移除海量存储器的引用计数，这样你就可以告诉用户何时可以安全地移除设备。这台设备遵循着通常的移除过程，用户发起移除设备的请求，系统告诉用户何时可以安全地移除。

本章剩余的部分将回答这些问题。

4.1. 为什么并发计数不可小看？

```
1  long  counter  =  0;
2
3  void  inc_count(void)
4  {
5     counter++;
6  }
7
8  long  read_count(void)
9  {
10     return  counter;
11  }
```

图 4.1: 直接计算!

让我们从简单的问题开始，比如图 4.1 中给出的一个算法（count_nonatomic.c）。我们在第 1 行有个计数器，我们在第 5 行对它加 1，我们在第 10 行读出它的值。还有什么比这个更简单的？

当你大量去读而几乎不增加计数时，这种方法快的让人炫目，在小型系统中，性能也非常好。

不过让人扫兴的是：这个方法会丢失计数。在我的双核笔记本上，程序短短执行一会儿，一共调用了 inc_count()函数 100,014,000 次，但是最终计数器的值只有 52,909,118。虽然近似值在计算领域有一定价值，但是做的更好总是有必要的。

小问题 4.6：++操作符在 x86 上不是会产生一个 add-to-memory 的指令么？为什么 CPU 高速缓存没有把这个指令当成原子的？

小问题 4.7：误差有 8 位数的精确度，说明作者确实是用心测试了的。但是为什么有必要去测试这么一个小小的程序呢，特别是 bug 用检查都可以看出？

```
1 atomic_t counter = ATOMIC_INIT(0);
2
3 void inc_count(void)
4 {
5     atomic_inc(&counter);
6 }
7
8 long read_count(void)
```

```
9 {
10     return atomic_read(&counter);
11 }
```

图 4.2：原子地直接计算!

精确计数的最直接的办法是使用原子操作，如图 4.2（count_atomic.c）。第 1 行定义了一个原子变量，第 5 行原子地加 1，第 10 行读出原子变量的值。因为都是原子操作，所以计数非常精确。不过，程序执行得要慢一些：在 Intel 双核笔记本上，当只有单线程自增时，它比非原子自增慢 6 倍，当有两个线程自增时，它比非原子自增要慢超过 10 倍。

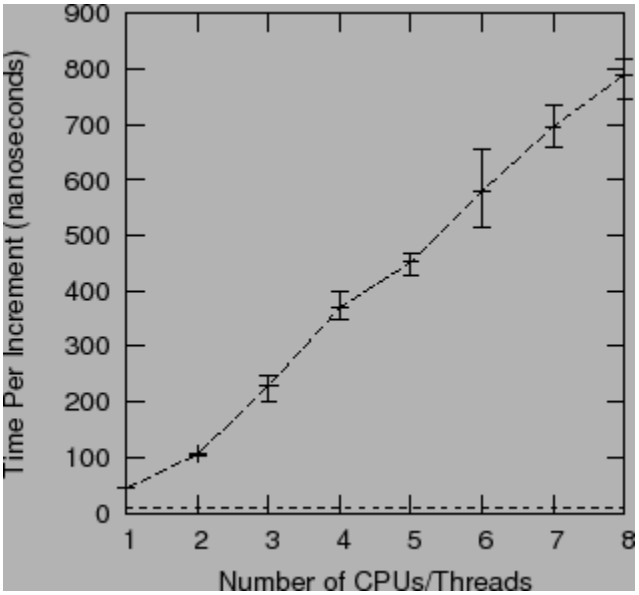


图 4.3：原子自增在 Nehalem 机器上的扩展性

这种糟糕的性能并不让人惊讶，第 2 章已经讨论过这一情况了，原子自增的性能随着 CPU 和线程个数的增加而下降这一事实也是如此，如图 4.3 所示。在这张图中，x 轴上的水平虚线代表完美的可扩展算法达到的理想性能：在这样的算法中，CPU 和线程数目的增长带来的开销和单线程时的开销相同。对单一全局变量的原子自增显然是不理想的，并且在你增加 CPU 后变得更糟。

小问题 4.8：为什么 x 轴上的虚线没有在 y=1 时与对角线相交？

小问题 4.9：但是原子自增还是很快啊。在一个紧凑循环中不断增加变量的值对我来说不太现实，毕竟程序是用来执行实际工作的，而不是计算它做了多少事！为什么我要关心如何让自增再快一点？

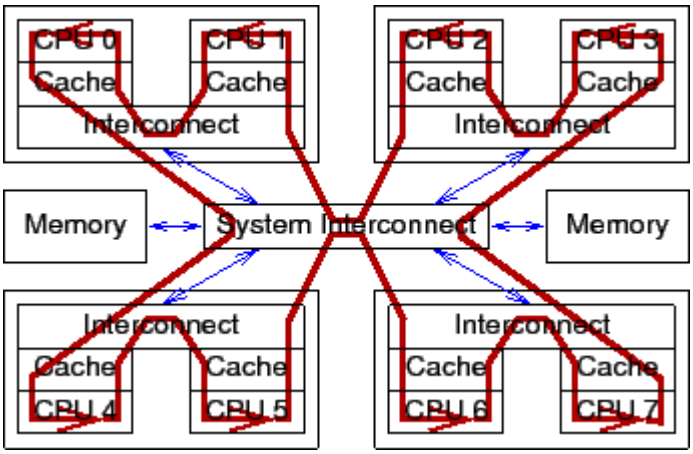


图 4.4：用全局原子操作自增的数据流

图 4.4 是另一种全局原子自增的视角。为了让每个 CPU 得到机会增加一个指定全局变量，包含变量的缓存线需要在所有 CPU 间传播，如图中红箭头所示。这种传播相当耗时，从而导致了图 4.3 中的糟糕性能。

下一节将讨论高性能计数，这种计数方法可以避免这种传播导致的内在延迟。

小问题 4.10：但是为什么 CPU 设计者不能简单地把操作发给数据，从而避免这种传播包含全局变量的缓存线的需要？

4.2. 统计计数器

本节包括常见的使用统计计数器的场景，计数极其频繁地被更新，但极少被读出，有时甚至根本不读。统计计数器用于解决小问题 4.2 中的网络报文计数问题。

4.2.1. 设计

统计计数一般以每个线程一个计数器的方式处理（或者在内核运行时，每个 CPU 一个），所以每个线程只更新自己的计数器。根据加法的交换律和结合律，总的计数值就是所有线程计数器值的简单相加。这就是 5.3.4 节中将引入的 Data Ownership 模式。

小问题 4.11：But doesn’t the fact that C’s “integers” are limited in size complicate things?

4.2.2. 基于数组的实现

一种实现每线程变量的方法是分配一个数组，数组每个元素对应一个线程（假设已经对齐并且填充过了，防止共享到错误的值）

小问题 4.12：数组？？？这会不会限制线程的个数？

```
1 DEFINE_PER_THREAD(long, counter);
2
3 void inc_count(void)
4 {
5     __get_thread_var(counter)++;
6 }
7
8 long read_count(void)
9 {
10    int t;
11    long sum = 0;
12
13    for_each_thread(t)
14       sum += per_thread(counter, t);
15    return sum;
16 }
```

图 4.5：基于数组的每线程统计计数器

这样一个数组可以封装成一句每线程原语，如图 4.5 所示（count_stat.c）。第 1 行定义了一个数组，包含一套类型为 long 的每线程计数器，计数器的名字叫 counter，很有创意吧(笑)。

第 3-6 行是增加计数器的函数，使用__get_thread_var()原语去定位当前运行线程对应 counter 数组的元素。因为这个元素只能由对应的线程修改，非原子的自增足够了。

第 8-16 行是读计数器总计数的函数，使用 for_each_thread()原语遍历当前运行的所有线程，使用 per_thread()原语去获取指定线程的计数器。因为硬件可以原子地存取正确对齐的 long 型数据，并且因为慈祥的 gcc 利用了这一点，所以普通的读取操作就足够了，不需要专门的原子指令。

小问题 4.13：尽管如此，gcc 还有没有其他选择？？？

小问题 4.14：图 4.5 中的每线程 counter 变量是如何初始化的？

小问题 4.15：假设图 4.5 中允许超过一个计数器，代码该怎么写？

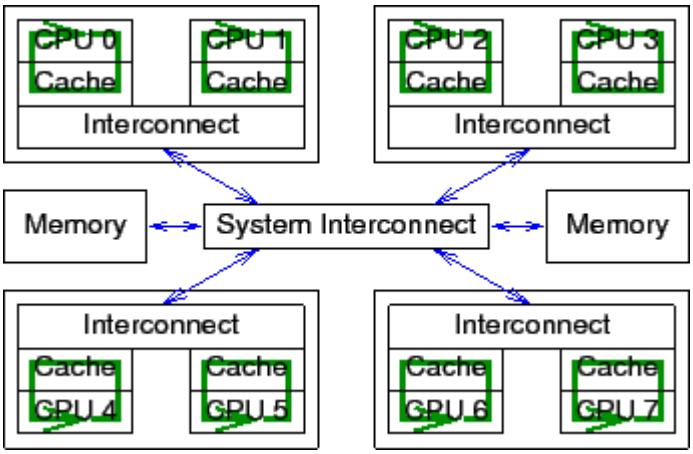


图 4.6：用每线程方法自增的数据流

该方法随着调用 inc_count()函数的更新者线程增加而线性扩展。如图 4.6 中绿色箭头所示，其原因是每个 CPU 可以快速地增加自己线程的变量值，不再需要代价昂贵的跨越整个计算机系统的通信。但是这种在“更新”上扩展极佳的方法，在存在大量线程时，会带来“读取”上的巨大代价。下一节将展示一种方法，能在保留“更新”侧扩展性的同时，减少“读取”侧产生的代价。

4.2.3. 结果一致的实现

一种保留“更新”侧可扩展性的同时又提升“读取”侧性能的方法是削弱一致性的要求。前一节介绍的计数算法要求保证返回的值在 read_count()执行前一刻的理想计数值和 read_count()执行完毕时的理想计数值之间。“结果的一致性”[Vog09]提供了一种弱一些的保证：不调用 inc_count()时，调用 read_count()最终会返回正确的值。

我们通过维护一个全局计数器来利用“最终一致性”。但是因为写者只操纵它自己的每线程计数器，所以单独有一个线程负责将每线程计数器的计数值传递给全局计数器。读者只是简单地访问全局计数器的值。如果写者正在更新计数，读者读出的值将不是最新的，不过一旦写者更新完毕，全局计数器最终会回归正确的值——这就是为什么这种方法被称为“最终一致性”的原因。

```
1  DEFINE_PER_THREAD(atomic_t,  counter);
2  atomic_t  global_count;
3  int  stopflag;
4
5  void  inc_count(void)
6  {
7     atomic_inc(&__get_thread_var(counter));
8  }
```

```
 9
10 unsigned long read_count(void)
11 {
12   return atomic_read(&global_count);
13 }
14
15 void *eventual(void *arg)
16 {
17   int t;
18   int sum;
19
20   while (stopflag < 3) {
21     sum = 0;
22     for_each_thread(t)
23       sum += atomic_xchg(&per_thread(counter,
t), 0);
24     atomic_add(sum, &global_count);
25     poll(NULL, 0, 1);
26     if (stopflag) {
27       smp_mb();
28       stopflag++;
29     }
30   }
31   return NULL;
32 }
33
34 void count_init(void)
35 {
36   thread_id_t tid;
37
38   if (pthread_create(&tid, NULL, eventual,
NULL) != 0) {
39     perror("count_init:pthread_create");
40     exit(-1);
41   }
```

```
42  }
43
44  void  count_cleanup(void)
45  {
46     stopflag  =  1;
47     while  (stopflag  <  3)
48        poll(NULL,  0,  1);
49     smp_mb();
50  }
```

图4.7：基于数组的每线程结果一致计数器

图 4.7（count_stat_eventual.c）中展示这种实现。第 1-2 行定义了跟踪计数器值的每线程变量和全局变量，第 3 行定义了 stopflag，用于控制程序结束（在我们想终止程序查看精确计数时）。第 5-8 行的 inc_count()函数和图 4.5 中的同名函数一样。第 10-13 行的 read_count()函数简单地返回变量 global_count 的值。

但是，第 34-42 行的 count_init()函数创建了第 15-32 行的 eventual()线程，该线程遍历所有线程，用 atomic_xchg()函数减少每个线程的本地计数器的值，将减去的值的总和加到变量 global_count 中。eventual()线程在每次循环之间等待 1 毫秒（随便选取的）。第 44-50 行的 count_cleanup()函数用来控制程序结束。

本方法在提供极快的读取计数性能的同时，仍然保持线性的更新计数性能曲线。但是，这种卓越的读取侧性能和更新侧可扩展性带来的是高昂的更新侧开销，因为__get_thread_var()原语中隐藏的原子操作和数组寻址对于某些带有很长管道的 CPU 来说十分耗时。

小问题 4.16：为什么图 4.16 中的 inc_count()需要使用原子指令？

小问题 4.17：图 4.7 的 eventual()函数中的单个全局线程是否会像全局锁一样成为性能瓶颈？

小问题 4.18：图 4.7 的 read_count()返回的估计值是否会在线程数增加时变得越来越不准确？

4.2.4. 基于每线程变量的实现

幸运的是，gcc 提供了一个用于每线程存储的__thread 存储类。图 4.8（count_end.c）中使用了这个类来实现统计计数器，这个统计计数器不仅能扩展，而且相对于简单的非原子自增来说几乎没有带来性能损失。

```
1  long  __thread  counter  =  0;
2  long  *counterp[NR_THREADS]  =  {  NULL  };
3  long  finalcount  =  0;
```

```
4  DEFINE_SPINLOCK(final_mutex);
5
6  void  inc_count(void)
7  {
8     counter++;
9  }
10
11  long  read_count(void)
12  {
13    int  t;
14    long  sum;
15
16    spin_lock(&final_mutex);
17    sum  =  finalcount;
18    for_each_thread(t)
19       if  (counterp[t]  !=  NULL)
20          sum  +=  *counterp[t];
21    spin_unlock(&final_mutex);
22    return  sum;
23  }
24
25  void  count_register_thread(void)
26  {
27    int  idx  =  smp_thread_id();
28
29    spin_lock(&final_mutex);
30    counterp[idx]  =  &counter;
31    spin_unlock(&final_mutex);
32  }
33
34  void  count_unregister_thread(int
nthreadsexpected)
35  {
36    int  idx  =  smp_thread_id();
37
```

```
38    spin_lock(&final_mutex);
39    finalcount  +=  counter;
40    counterp[idx]  =  NULL;
41    spin_unlock(&final_mutex);
42  }
```

图 4.8 基于每线程变量的统计计数器

第 1-4 行定义了所需的变量：counter 是每线程计数器变量，counterp[]数组允许线程访问彼此的计数器，finalcount 在各个线程退出时将计数值累加到总和里，final_mutex 协调累加计数器总和值的线程和退出的线程。

小问题 4.19：为什么我们需要一个显式的数组来找到其他线程的计数器？为什么 gcc 不提供一个像内核 per_cpu()原语一样的 per_thread()接口，让线程可以更容易地访问彼此的每线程变量？

更新者调用的 inc_count()函数非常简单，见第 6-9 行。

写者调用的 read_count()函数稍微复杂一点。第 16 行获取锁与正在退出的线程互斥，第 21 行释放锁。第 17 行初始化已退出线程的每线程计数的总和，第 18-20 行将还在运行的线程的每线程计数累加进总和。最后，第 22 行返回总和。

小问题 4.20：为什么我们需要像互斥锁这种重量级的手段来保护图 4.8 的 read_count()函数中的累加总和操作？

第 25-32 行是 count_register_thread()函数，每个线程在访问自己的计数器前都要调用它。该函数简单地将本线程对应 counterp[]数组中的元素指向线程的每线程变量 counter。

小问题 4.21：为什么我们需要在图 4.8 的 count_register_thread()函数获取锁？这只是一个将对齐的机器码存储在一块指定位置的操作，而又没有其他线程修改这个位置，所以这应该是一个原子操作，对吧？

第 34-42 行是 count_unregister_thread()函数，每个之前调用过 count_register_thread()函数的线程在退出时都需要调用该函数。第 38 行获取锁，第 41 行释放锁，因此排除了有进程调用 read_count()同时又有进程调用 count_unregister_thread()的情况。第 39 行将本线程的每线程计数加到全局的 finalcount 里，然后将 counterp[]数组的对应元素设置为 NULL。随后的 read_count()调用可以在全局变量 finalcount 里找到退出线程的计数值，并且在顺序访问 coutnerp[]数组时可以跳过已退出的线程，这样就能获得正确的结果。

这个方法让更新者的性能几乎和非原子自增计独数一样，并且也能线性地扩展。另一方面，并发的读者竞争一个全局锁，因此性能不佳，扩展能力也很差。但是，这不是统计计数器需要面对的问题，因为统计计数器总是在增加计数，很少读取计数。另外，本方法比基于数组的方法要复杂一些，因为在线程退出时，

线程的每线程变量也清零了。

小问题 4.22：很好，但是 Linux 内核在读取每 CPU 计数器的总和值时没有用锁保护。为什么用户态的代码需要这么做？？？

4.2.5. 讨论

上述的两种实现显示，尽管是并行运行，但是完全有可能让统计计数器达到在单处理器上的性能。

小问题 4.23：如果每个报文的大小不一，那么统计报文个数和统计报文的总字节数之间到底有什么本质的区别？

小问题 4.24：读者并须累加所有线程的计数器，当线程数较多时会花费很长时间。有没有什么办法能让累加操作既快又扩展性良好，同时又能让读者的性能和扩展性很合理呢？

通过学习本节的内容，现在你应该能回答本章开始处的小问题中关于网络统计计数器的问题了吧。

4.3. 近似上限计数器

另一个关于计数的情景是上限检查。比如，本章开始处的小问题提到的结构分配的近似上限问题，假设你需要维护一个已分配结构数目的计数，来防止分配超过一个上限，呃，比如 10000。我们再进一步假设这些结构的生命周期很短，极少超出上限。

4.3.1. 设计

有一种可能实现上限计数器的方法是将 10000 的限制值平均划分给每个线程，然后给每个线程一个固定个数的资源池。假如有 100 个线程，每个线程管理一个有 100 个结构的资源池。这种方法简单，在有些情况下也有效，但是这种方法无法处理一种常见情况：某个结构由一个线程创建，但由另一个线程释放[MS93]。

一方面，如果线程释放一个结构就积一分的话，那么一直在分配的线程很快就分配光了资源池，而一直在释放的线程积攒了大量分数却无法使用。另一方面，如果每个释放的结构都能让分配它的 CPU 加一分，CPU 就需要操纵其他 CPU 的计数器，这将会带来很多代价昂贵的原子操作。此外，因为结构的大小不一，我们需要用 add_count()和 sub_count()来替代 inc_count()和 dec_count()，前两个函数能正确的处理变长结构。

简而言之，在很多重要的情景下我们都不能将计数器问题完全分治。但是，我们可以采用部分分治的办法，这样在通常情况下，每个线程只需操纵自己的私有状态，而计数还是可以在进程间传递。4.2.4 节讨论的统计计数器方案提供了一个有趣的出发点，在该方案中统计计数器维护了每线程计数器和一个全局计数器，总的统计值时所有计数器值之和，每线程计数器加上全局计数器。关键的部分在于将仍在运行的线程的统计值加入全局的合计值，而不是等待这些线程退出。毋庸置疑，we want threads to pull in their own counts, as cross-thread accesses are expensive and scale poorly.

这就留下一个问题，到底应该在何时将线程计数器加入全局计数器中。在最初的实现中，我们维护一个每线程计数器的上限值。当超过这个上限时，线程将自己的计数加到全局计数器上。当然，我们不能只简单地在分配结构时增加计数值：我们还必须在结构释放时减少计数值。因此我们必须在减少计数值时利用上全局计数器，否则每线程计数器的值就可能降到 0 以下。但是，如果这个上限足够大，那么几乎所有的加减操作都是在每线程计数器中执行的，这就给我们带来了良好的性能和可扩展性。

这种设计就是一个“并行捷径”的例子，这是一种重要的设计模式，适用于在多数情况没有线程间通信和交互的开销，而偶尔进行的跨进程通信又使用了精心设计的全局算法的情况。

4.3.2. 简单的上限计数器实现

```
1  unsigned  long  __thread  counter  =  0;
2  unsigned  long  __thread  countermax  =  0;
3  unsigned  long  globalcountmax  =  10000;
4  unsigned  long  globalcount  =  0;
5  unsigned  long  globalreserve  =  0;
6  unsigned  long  *counterp[NR_THREADS]  =  {  NULL  };
7  DEFINE_SPINLOCK(gblcnt_mutex);
```

图 4.9：简单上限计数器定义的变量

图 4.9 是实现代码所用的每线程变量和全局变量。每线程变量 counter 和 countermax 各自对应线程的本地计数器和计数器上限。第 3 行的 globalcountmax 变量代表合计计数器的上限，第 4 行的 globalcount 变量是全局计数器。globalcount 和每个线程的 counter 之和就是合计计数器的值。第 5 行的 globalreserve 变量是所有每线程变量 countermax 的和。这些变量之间的关系见图 4.10。

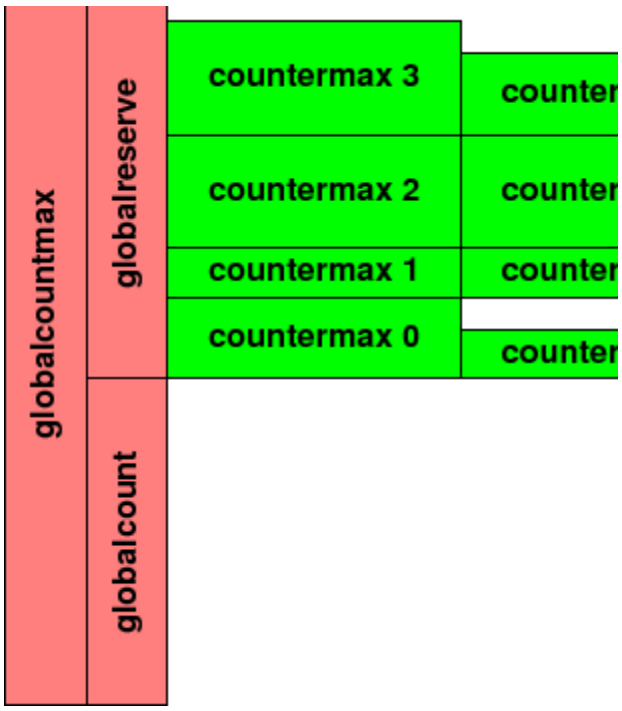


图 4.10 简单上限计数器定义的变量之间的关系

1. globalcount 和 globalreserve 的和小于等于 globalcountmax 的值。
2. 所有线程的 countermax 值的总和小于等于 globalreserve 的值。
3. 每个线程的 counter 值小于等于该线程的 countermax 的值。

counterp[]数组的每个元素指向对应线程的 counter 变量，最后，gblcnt_mutex 自旋锁保护所有全局变量，也就是说，除非线程获取了 gblcnt_mutex 锁，否则不能访问或者修改任何全局变量。

```
1  int  add_count(unsigned  long  delta)
2  {
3     if  (countermax  -  counter  >=  delta)  {
4        counter  +=  delta;
5        return  1;
6     }
7     spin_lock(&gblcnt_mutex);
8     globalize_count();
9     if  (globalcountmax  -
10 globalcount  -  globalreserve  <  delta)  {
11        spin_unlock(&gblcnt_mutex);
12        return  0;
13     }
14     globalcount  +=  delta;
15     balance_count();
```

```
16    spin_unlock(&gblcnt_mutex);
17    return 1;
18  }
19
20  int sub_count(unsigned long delta)
21  {
22    if (counter >= delta) {
23       counter -= delta;
24       return 1;
25    }
26    spin_lock(&gblcnt_mutex);
27    globalize_count();
28    if (globalcount < delta) {
29       spin_unlock(&gblcnt_mutex);
30       return 0;
31    }
32    globalcount -= delta;
33    balance_count();
34    spin_unlock(&gblcnt_mutex);
35    return 1;
36  }
37
38  unsigned long read_count(void)
39  {
40    int t;
41    unsigned long sum;
42
43    spin_lock(&gblcnt_mutex);
44    sum = globalcount;
45    for_each_thread(t)
46       if (counterp[t] != NULL)
47          sum += *counterp[t];
48    spin_unlock(&gblcnt_mutex);
49    return sum;
50  }
```

图 4.11：简单上限计数器的加、减和读函数

图 4.11 是 add_count()、sub_count()和 read_count()函数(count_lim.c)。

第 1-18 行是 add_count()，将指定的值 delta 加到 counter 上。第 3 行检查本线程的 counter 是否还有足够的空间给 delta，如果有，第 4 行让 counter 加上 delta，第 6 行返回成功。这是 add_counter()的快速路径，不做原子操作，只引用每线程变量，因此也不会触发任何 cache miss。

小问题 4.25：图 4.11 中第 3 行的判断条件为什么那么奇怪？为什么不用下面这个更直观的形式判断进不进快速路径？

```
3  if  (counter  +  delta  <=  countermax){
4     counter  +=  delta;
5     return  1;
6  }
```

如果第 3 行的测试失败，我们必须访问全局变量，这就需要获取第 7 行的 gblcnt_mutex 锁，如果测试失败，在第 11 行释放该锁，否则在第 16 行释放该锁。第 8 行调用了 globalize_count()，如图 4.12 所示，该函数清除线程本地变量，根据需要调整全局变量，这就简化了全局处理过程。（别光看我的话，试着自己写代码!!）第 9 行和第 10 行检查新增的 delta 能不能被容纳进去，小于号前面的表达式的含义就是图 4.10 中两个红柱之间的高度差。如果容纳不下 delta 的大小，第 11 行释放 gblcnt_mutex 锁（前面已经提过），第 12 行返回 1 表示错误。

如果可以容纳 delta 的大小，第 14 行从 globalcount 中减去 delta，第 15 行调用 balance_count()（如图 4.12）来更新全局变量和每线程变量（最好设置该线程的 countermax 以后又可以重新进入快速路径）。如果可以的话，重新进入快速路径的处理，第 16 行释放 gblcnt_mutex 锁（前面已经提过），最后第 17 行返回 0 表示成功。

小问题 4.26：为什么在图 4.11 里，globalize_count()将每线程变量设为 0，只是用来留给后面的 balance_count()重新填充它们？为什么不直接让每线程变量的值非 0？

第 20-36 行的 sub_count()从 counter 中减去指定的 delta。第 22 行检查每线程计数器减去 delta 后是否大于 0，如果是，第 23 行将执行减法操作，第 24 行返回成功。这就是 sub_count()的快速路径，和 add_count()一样，这条快速路径并不执行耗时的操作。

```
1  static  void  globalize_count(void)
2  {
3     globalcount  +=  counter;
4     counter  =  0;
```

```
    5     globalreserve  -=  countermax;
    6     countermax  =  0;
    7  }
    8
    9  static  void  balance_count(void)
   10  {
   11     countermax  =  globalcountmax  -
   12                    globalcount  -  globalreserve;
   13     countermax  /=  num_online_threads();
   14     globalreserve  +=  countermax;
   15     counter  =  countermax  /  2;
   16     if  (counter  >  globalcount)
   17        counter  =  globalcount;
   18     globalcount  -=  counter;
   19  }
   20
   21  void  count_register_thread(void)
   22  {
   23     int  idx  =  smp_thread_id();
   24
   25     spin_lock(&gblcnt_mutex);
   26     counterp[idx]  =  &counter;
   27     spin_unlock(&gblcnt_mutex);
   28  }
   29
   30  void  count_unregister_thread(int
nthreadsexpected)
   31  {
   32     int  idx  =  smp_thread_id();
   33
   34     spin_lock(&gblcnt_mutex);
   35     globalize_count();
   36     counterp[idx]  =  NULL;
   37     spin_unlock(&gblcnt_mutex);
   38  }
```

图 4.12：简单上限计数器的功能函数

如果无法满足减去 delta 后大于 0 的要求，就要进入第 26-35 行的慢速路径了。因为慢速路径必须访问全局状态，所以第 26 行获取了 gblcnt_mutex 锁，在第 29 行释放（失败的情况）或者在第 34 行释放（成功的情况）。第 27 行调用 globalize_count()，如图 4.12，再一次清零线程的每线程变量，根据需要调整全局变量的值。第 28 行检查计数器值是否够 delta 减，如果不够，第 29 行释放 gblcnt_mutex 锁（之前提到过）并在第 30 行返回失败。

小问题 4.27：add_count()中的 globalreserve 对我们不利，为什么在图 4.11 中的 sub_count()里没有对我们不利？

另一方面，如果第 28 行发现计数器的值够 delta 去减，那么在第 32 行执行减法操作，第 33 行调用 balance_count()（见图 4.12）来更新全局变量和每线程变量的值，第 34 行释放 gblcnt_mutex 锁，第 35 行返回成功。

小问题 4.28：为什么图 4.11 中要同时有 add_count()和 sub_count()？为什么不简单的给 add_count()传一个负值？

第 38-50 行的 read_count()返回计数器的合计值。该函数在第 43 行获取 gblcnt_mutex 锁，在第 48 行释放，保证了与 add_count()和 sub_count()中全局操作之间的互斥访问，并且正如我们见到的，还对线程创建和退出进行了互斥保护。第 44 行初始化了本地变量 sum，将 globalcount 的值赋给它，然后第 45-47 行的循环计算每线程变量 counter 的总和。第 49 行返回总和的值。

图 4.12 中是 add_count(), sub_count()和 read_count()中使用到的函数。

第 1-7 行是 globalize_count()，清零当前线程的每线程计数器，适当地调整全局变量的值。需要注意的是，此函数并不改变计数器合计值，而是改变计数器当前值表现的方式。第 3 行将线程的 counter 变量加到 globalize_count 上，第 4 行清零 counter。类似地，第 5 行从 globalreserve 中减去每线程变量 countermax 的值，第 6 行清零 countermax。重新看一遍图 4.10 有助于理解本函数和 balance_count()函数，也就是下面要讲的函数。

第 9-19 行是 balance_count()，简单地说就是 globalize_count()的反向操作。该函数设置当前线程的 counter 和 countermax 变量（同时调整对应的 globalcount 和 globalreserve），以尽量满足 add_count()和 sub_count()的快速路径条件。和 globalize_count()一样，balance_count()并不改变计数器合计值。第 11-13 行计算本线程的计数器值在 globalcountmax 中占的比例，确保没有超过 globalcount 或者 globalreserve，并将计算出的值赋给本线程的 coutnermax。第 14 行对 globalreserve 做相应的调整。第 15 行将本线程的 counter 设置为 0 到 countermax 的中间值。第 16 行检查是否 globalcount 可以够 counter 减，如果不够，第 17 行相应地减少 counter。最后，无论检查结果如何，第 18 行对 globalcount 进行相应

的调整。

第 21-28 行是 count_register_thread()，未新创建的线程设置状态。该函数在 gblcnt_mutex 锁的保护下，简单地将指向新创建线程的 counter 变量的指针放入 counterp[]数组的对应元素。

最后，第 30-38 行是 count_unregister_thread()，销毁即将退出的线程的状态。第 34 行获取 gblcnt_mutex 锁，第 37 行释放锁。第 35 行调用 globalize_count()清除本线程的计数器状态，第 36 行清除 counterp[]中对应本线程的元素。

4.3.3. 关于简单上限计数器的讨论

当合计值接近 0 时，简单上限计数器运行的相当块，只在 add_count()和 sub_count()的快速路径中的比较和判断时存在一些开销。但是，每线程变量 countermax 的使用表明 add_count()即使在计数器的合计值离 globalcountmax 很远时也可能失败。同样地，sub_count()在计数器合计值远不到 0 时也可能失败。

在许多情况下，这都是不可接受的。即使 globalcountmax 只不过是一个近似的上限，一般也有一个近似度的容忍限度。一种限制近似度的方法是对每线程变量 countermax 的值强加一个上限。这个任务将在下一节完成。

4.3.4. 近似上限计数器的实现

```
1 unsigned long __thread counter = 0;
2 unsigned long __thread countermax = 0;
3 unsigned long globalcountmax = 10000;
4 unsigned long globalcount = 0;
5 unsigned long globalreserve = 0;
6 unsigned long *counterp[NR_THREADS] = { NULL };
7 DEFINE_SPINLOCK(gblcnt_mutex);
8 #define MAX_COUNTERMAX 100
```

图 4.13：近似上限计数器定义的变量

因为实现代码（count_lim_app.c）和上几节的例子很相像（图 4.9，图 4.11，图 4.12），这里就是只描述差别部分了。除了 MAX_COUNTERMAX 以外，图 4.13 和图 4.9 完全一样，MAX_COUNTERMAX 设置了每线程变量 countermax 的最大允许值。

```
1 static void balance_count(void)
2 {
3   countermax = globalcountmax - globalcount -
```

```
globalreserve;
4     countermax  /=  num_online_threads();
5     if  (countermax  >  MAX_COUNTERMAX)
6        countermax  =  MAX_COUNTERMAX;
7     globalreserve  +=  countermax;
8     counter  =  countermax  /  2;
9     if  (counter  >  globalcount)
10        counter  =  globalcount;
11    globalcount  -=  counter;
12  }
```

图 4.14：近似上限计数器的平衡函数

类似地，图 4.14 和图 4.12 中的 balance_count()几乎完全一样，除了第 5 行和第 6 行，对每线程变量 countermax 增加了 MAX_COUNTERMAX 的限制。

4.3.5. 关于近似上限计数器的讨论

上述的改动极大地减小了在前一个版本中出现的上限不准确程度，但是又带来了另一个问题：任何给定大小的 MAX_COUNTERMAX 将导致一部份访问无法进入快速路径，这部份访问的数目取决于工作负荷。随着线程数的增加，非快速路径的执行将成为一个性能和可扩展性上的问题。不过，我们暂时放下这个问题，先去看带有精确上限的计数器。

4.4. 精确上限计数器

为了解决本章开头部分关于结构分配数目精确上限问题的小问题，我们需要一个上限计数器，它能精确的知道何时计数超过上限。一种实现上限计数器的办法是 cause threads that have reserved counts to give them up. 另一种办法是采用原子操作。当然，原子操作会减慢快速路径，但是另一方面，如果连试一试都不干又有点过于愚蠢了。

4.4.1. 原子上限计数器的实现

不幸的是，当某个线程丢弃它的计数时，需要自动地操纵线程的 counter 和 countermax 变量。通常的做法是将这两个变量合并成一个变量，比如，一个 32 位的变量，高 16 位代表 counter，低 16 位代表 countermax。

```
1  atomic_t  __thread  counterandmax  =  ATOMIC_INIT(0);
```

```
2  unsigned  long  globalcountmax  =  10000;
3  unsigned  long  globalcount  =  0;
4  unsigned  long  globalreserve  =  0;
5  atomic_t  *counterp[NR_THREADS]  =  {  NULL  };
6  DEFINE_SPINLOCK(gblcnt_mutex);
7  #define  CM_BITS  (sizeof(atomic_t)  *  4)
8  #define  MAX_COUNTERMAX  ((1  <<  CM_BITS)  -  1)
9
10  static  void
11  split_counterandmax_int(int  cami,  int  *c,
                            int  *cm)
12  {
13    *c  =  (cami  >>  CM_BITS)  &  MAX_COUNTERMAX;
14    *cm  =  cami  &  MAX_COUNTERMAX;
15  }
16
17  static  void
18  split_counterandmax(atomic_t  *cam,  int  *old,
19                      int  *c,  int  *cm)
20  {
21    unsigned  int  cami  =  atomic_read(cam);
22
23    *old  =  cami;
24    split_counterandmax_int(cami,  c,  cm);
25  }
26
27  static  int  merge_counterandmax(int  c,  int  cm)
28  {
29    unsigned  int  cami;
30
31    cami  =  (c  <<  CM_BITS)  |  cm;
32    return  ((int)cami);
33  }
```

图4.15：原子上限计数器定义的变量和访问函数

一个简单的原子上限计数器所用的变量和访问函数如图 4.15 所示

（count_lim_atomic.c）。根据刚才的算法，counter 和 countermax 变量组合成第 1 行的 counterandmax 变量，高字节是 counter，低字节是 countermax。该变量的类型为 atomic_t，实际上是由 int 来表示。

第 2-6 行是 globalcountmax，globalcount，globalreserve，counterp 和 gblcnt_mutex 的定义，所有变量的含义都和图 4.13 中的一样。第 7 行定义了 CM_BITS，代表 counterandmax 的高位或者低位所占的 bit 数，第 8 行定义了 MAX_COUNTERMAX，代表 counterandmax 的高位或者低位可能表示的最大值。

小问题 4.29：图 4.15 中的第 7 行违反了 C 标准的哪一条？

第 10-15 行是 split_counterandmax_int()函数，从 atomic_t 类型的 counterandmax 变量中分解出类型为 int 的高 16 位和低 16 位。第 13 行算出 counterandmax 的高 16 位，将结果赋给参数 c，第 14 行算出 counterandmax 的低 16 位，将结果赋给参数 cm。

第 17-25 行是 split_counterandmax()函数，从第 21 行指定的变量中读取值，在第 23 行将该值赋给参数 old，然后在第 24 行调用 split_counterandmax_int()分解该值。

小问题 4.30：既然只有一个 counterandmax 变量，为什么图 4.15 中的第 18 行还要传递一个指针给它。

第 27-33 行是 merge_counterandmax()函数，可以看作是 split_counterandmax()的反向操作。第 31 行传入的参数 c 和 cm 分别对应 counter 和 countermax，将它们合并成一个 int，然后返回结果。

小问题 4.31：为什么图 4.15 中的 merge_counterandmax()返回 int 而不是直接返回 atomic_t?

```
1  int  add_count(unsigned  long  delta)
2  {
3     int  c;
4     int  cm;
5     int  old;
6     int  new;
7
8     do  {
9        split_counterandmax(&counterandmax,  &old,
                              &c,  &cm);
10        if  (delta  >  MAX_COUNTERMAX  ||  c  +
               delta  >  cm)
11 goto  slowpath;
```

```
12        new = merge_counterandmax(c + delta, cm);
13    } while (atomic_cmpxchg(&counterandmax,
14                old, new) != old);
15    return 1;
16 slowpath:
17    spin_lock(&gblcnt_mutex);
18    globalize_count();
19    if (globalcountmax - globalcount -
20         globalreserve < delta) {
21        flush_local_count();
22        if (globalcountmax - globalcount -
23             globalreserve < delta) {
24            spin_unlock(&gblcnt_mutex);
25            return 0;
26        }
27    }
28    globalcount += delta;
29    balance_count();
30    spin_unlock(&gblcnt_mutex);
31    return 1;
32 }
33
34 int sub_count(unsigned long delta)
35 {
36    int c;
37    int cm;
38    int old;
39    int new;
40
41    do {
42        split_counterandmax(&counterandmax, &old,
                              &c, &cm);
43        if (delta > c)
44 goto slowpath;
45        new = merge_counterandmax(c - delta, cm);
```

```
46     } while (atomic_cmpxchg(&counterandmax,
47                 old, new) != old);
48     return 1;
49 slowpath:
50     spin_lock(&gblcnt_mutex);
51     globalize_count();
52     if (globalcount < delta) {
53         flush_local_count();
54         if (globalcount < delta) {
55           spin_unlock(&gblcnt_mutex);
56           return 0;
57         }
58     }
59     globalcount -= delta;
60     balance_count();
61     spin_unlock(&gblcnt_mutex);
62     return 1;
63 }
```

图4.16：原子上限计数器的加、减函数

图 4.16 是 add_count()，sub_count()和 read_count()参数。

第 1-32 行是 add_count()，第 8-15 行是它的快速路径。第 8-14 行的快速路径组成了一个比较并交换（CAS）循环，第 13-14 行的 atomic_cmpxchg()原语执行实际的 CAS。第 9 行将当前线程的 counterandmax 变量拆分成线程的 counter（在 c 中）和 countermax（在 cm 中），并把原值赋给 old。第 10 行检查本地的每线程变量能否容纳 delta（要小心避免整形溢出），如果不能容纳，第 11 行开始进入慢速路径。如果可以容纳，第 12 行将更新过后的 counter 和原来的 countermax 值合并成成变量 new。第 13-14 行的 atomic_cmpxchg()原语自动比较该线程的 countermax 和 old，如果比较成功，将交互后的值赋给 new。如果比较成功，第 15 行返回成功，否则继续执行第 9 行的循环。

小问题 4.32：呃！图 4.16 第 11 行那个丑陋的 goto 是干什么用的？你难道没听说 break 吗？？？

小问题 4.33：为什么图 4.16 的第 13-14 行的 atomic_cmpxchg()会失败？我们在第 9 行取出 old 值以后可没改过它！

图 4.16 的第 16-32 行是 add_count()的慢速路径，由 gblcnt_mutex 锁保护，该锁在第 17 行获取，在第 24 行和第 30 行释放。第 18 行调用了 globalize_count()

函数，将本线程的状态赋给全局计数器。第 19-20 行检查当前的全局状态能否容纳 delta 的值，如果不能，第 21 行调用 flush_local_count()将所有线程的本地状态刷新到全局计数器上，然后第 22-23 行重新检查是否可以容纳 delta。如果这样还是不行，那么第 24 行释放 gblcnt_mutex 锁（前面提过），然后在第 25 行返回失败。

如果可以容纳 delta 的值，第 28 行将 delta 加到全局计数器中，如果可以，第 29 行将计数赋给本地状态，第 30 行释放 gblcnt_mutex 锁（前面提过），最后第 31 行返回成功。

图 4.16 的第 34-63 行是 sub_count()函数，结构和 add_count()很相像，第 41-48 行是一条快速路径，第 49-62 行是一条慢速路径。本函数的逐行分析工作就留给读者来做了。

```
1  unsigned  long  read_count(void)
2  {
3     int  c;
4     int  cm;
5     int  old;
6     int  t;
7     unsigned  long  sum;
8
9     spin_lock(&gblcnt_mutex);
10     sum  =  globalcount;
11     for_each_thread(t)
12        if  (counterp[t]  !=  NULL)  {
13             split_counterandmax(counterp[t],  &old,
                                        &c,  &cm);
14             sum  +=  c;
15        }
16     spin_unlock(&gblcnt_mutex);
17     return  sum;
18 }
```

图 4.17：原子上限计数器的读函数

图 4.17 是 read_count()。第 9 行获取 gblcnt_mutex 锁，第 16 行释放。第 10 行用 globalcount 的值初始化局部变量 sum，第 11-15 行的循环将每线程计数器的值累加到 sum 中，第 13 行用 split_counterandmax()函数分解出每线程计数器的值。最后第 17 行返回 sum。

```
1  static  void  globalize_count(void)
2  {
3     int  c;
4     int  cm;
5     int  old;
6
7     split_counterandmax(&counterandmax,  &old,  &c,
                          &cm);
8     globalcount  +=  c;
9     globalreserve  -=  cm;
10    old  =  merge_counterandmax(0,  0);
11    atomic_set(&counterandmax,  old);
12 }
13
14 static  void  flush_local_count(void)
15 {
16    int  c;
17    int  cm;
18    int  old;
19    int  t;
20    int  zero;
21
22    if  (globalreserve  ==  0)
23       return;
24    zero  =  merge_counterandmax(0,  0);
25    for_each_thread(t)
26       if  (counterp[t]  !=  NULL)  {
27            old  =  atomic_xchg(counterp[t],  zero);
28            split_counterandmax_int(old,  &c,  &cm);
29            globalcount  +=  c;
30            globalreserve  -=  cm;
31       }
32 }
33
34 static  void  balance_count(void)
```

```
35 {
36    int  c;
37    int  cm;
38    int  old;
39    unsigned  long  limit;
40
41    limit  =  globalcountmax  -  globalcount  -
                globalreserve;
42    limit  /=  num_online_threads();
43    if  (limit  >  MAX_COUNTERMAX)
44       cm  =  MAX_COUNTERMAX;
45    else
46       cm  =  limit;
47    globalreserve  +=  cm;
48    c  =  cm  /  2;
49    if  (c  >  globalcount)
50       c  =  globalcount;
51    globalcount  -=  c;
52    old  =  merge_counterandmax(c,  cm);
53    atomic_set(&counterandmax,  old);
54 }
55
56 void  count_register_thread(void)
57 {
58    int  idx  =  smp_thread_id();
59
60    spin_lock(&gblcnt_mutex);
61    counterp[idx]  =  &counterandmax;
62    spin_unlock(&gblcnt_mutex);
63 }
64
65 void
   count_unregister_thread(int  nthreadsexpected)
66 {
67    int  idx  =  smp_thread_id();
```

```
68
69      spin_lock(&gblcnt_mutex);
70      globalize_count();
71      counterp[idx]  =  NULL;
72      spin_unlock(&gblcnt_mutex);
73  }
```

图 4.18：原子上限计数器的功能函数

图 4.18 是功能函数 globalize_count()、flush_local_count()、balance_count()、count_register_thread()和 count_unregister_thread()。第 1-12 行是 globalize_count()，和上一个版本的算法很相似，除了第 7 行，将 countermax 分解成 counter 和 countermax。

第 14-32 行是 flush_local_count()的代码，将所有线程的本地计数器状态赋给全局计数器。第 22 行检查 globalreserve 的值，看是否允许其他每线程计数器计数，如果不允许，第 23 行返回。如果允许，第 24 行初始化一个局部变量 zero，将两个为 0 的 counter 和 countermax 合并过后的值赋给 zero。第 25-31 行的循环遍历所有线程。第 26 行检查是否当前线程有计数器状态，如果有，第 27-30 行自动从当前线程状态中取值，然后将状态值赋为 0。第 28 行将读出的状态值分解为 counter(局部变量 c)和 countermax（局部变量 cm）。第 29 行将本线程的 counter 加到 globalcount 上，第 30 行从 globalreserve 中减去本线程的 countermax。

小问题 4.34： What stops a thread from sim- ply refilling its counterandmax variable immediately after flush local count() on line 14 of Figure 4.18 empties it?

小问题 4.35：当图 4.18 中第 27 行的 flush_local_count()在清零 counterandmax 变量时，是什么阻止了 atomic_add()或者 atomic_sub()的快速路径对 counterandmax 变量的干扰？

第 34-54 行是 balance_count()函数的代码，重填调用线程的本地 counterandmax 变量。该函数和前一个版本的算法很相似，只除了需要处理合并后的 counterandmax 变量。对该代码的详细分析就留给读者做练习了，第 56 行开始的 count_register_thread()函数和第 65 行开始的 count_unregister_thread()也留给读者分析了。

小问题 4.36：既然 atomic_set()原语将数据存放到指定的 atomic_t 中，图 4.18 中第 53 行的 balance_count()面对并发的 flush_local_count()，怎样才能正确地更新变量呢？

4.4.2. 关于原子上限计数器的讨论

这是第一个真正允许计数器一直自增直到它的上限的计数器，但是这也带来了快速路径上原子操作的开销，让快速路径明显变慢了。虽然在某些场合上这种减慢是允许的，但是去探索让“读取”侧性能更好的算法仍然值得我们去做。而使用信号处理函数从其他线程窃取计数就是其中一种算法。因为信号处理函数可以运行在收到信号线程的上下文，所以就不需要原子操作了，下一节将讨论这一算法。

小问题 4.37：但是信号处理函数可能会在运行时迁移到其他 CPU 上执行。难道这种情况就不需要原子操作和内存屏障来保证线程和中断线程的信号处理函数之间通信的可靠性吗？

4.4.3. Signal-Theft 上限计数器的设计

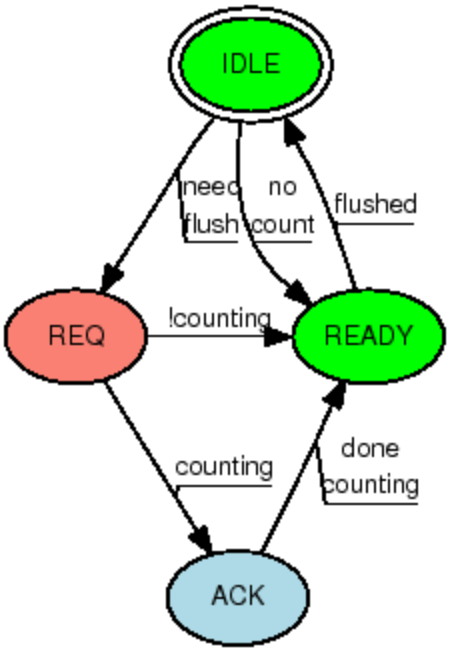


图 4.19：Signal-Theft 状态机

图 4.19 是 Signal-Theft 状态机。状态机从 IDLE 状态开始，当 add_count()和 sub_count() 发现线程的本地计数和全局计数之和已经不足以容纳请求的大小时，对应的慢速路径将每个线程的 theft 状态设置为 REQ（除非线程没有计数值，这样它就直接转为 READY）。只有在慢速路径获得 gblcnt_mutex_lock 后，才允许从 IDLE 状态转为其他状态，如图中绿色部分所示。然后慢速路径向每个线程发送一个信号，对应的信号处理函数检查本线程的 theft 和 counting 状态。如果 theft 状态不为 REQ，那么信号处理函数就不能改变其状态，只能直接返回。而 theft

状态为 REQ 时，如果设置了 counting 变量，表明当前线程正处于快速路径，信号处理函数将 theft 状态设置为 ACK，而不是 READY。

如果 theft 状态是 ACK，那么只有快速路径才有权改变 theft 的状态，如图中蓝色部分所示。当快速路径完成时，会将 theft 状态设置为 READY。

一旦慢速路径发现某个线程的 theft 状态为 READY，这时慢速路径有权窃取此线程的计数。然后慢速路径将线程的 theft 状态设置为 IDLE。

小问题 4.38：在图 4.19 中，为什么 REQ theft 状态被涂成蓝色？

小问题 4.39：在图 4.19 中，分别设置 REQ 和 ACK theft 状态的目的是什么？为什么不合并这两种状态来简化状态机？这样无论是哪个信号处理函数还是快速路径，先进入者可以将状态设置为 READY。

4.4.4. Signal-Theft 上限计数器的实现

```
1  #define  THEFT_IDLE    0
2  #define  THEFT_REQ     1
3  #define  THEFT_ACK     2
4  #define  THEFT_READY   3
5
6  int  __thread  theft  =  THEFT_IDLE;
7  int  __thread  counting  =  0;
8  unsigned  long  __thread  counter  =  0;
9  unsigned  long  __thread  countermax  =  0;
10  unsigned  long  globalcountmax  =  10000;
11  unsigned  long  globalcount  =  0;
12  unsigned  long  globalreserve  =  0;
13  unsigned  long  *counterp[NR_THREADS]  =
{  NULL  };
14  unsigned  long  *countermaxp[NR_THREADS]  =
{  NULL  };
15  int  *theftp[NR_THREADS]  =  {  NULL  };
16  DEFINE_SPINLOCK(gblcnt_mutex);
17  #define  MAX_COUNTERMAX  100
```

图 4.20：Signal-Theft 上限计数器定义的变量

图 4.20（count_lim_sig.c）显示了基于 signal-theft 计数器的实现所需的数据结构。第 1-7 行定义了上一节所说的 per-thread 状态机所需的状态值和变量。第

8-17 行和之前的实现比较类似，只增加了第 14 和 15 行，允许远程访问线程的 countermax 和 theft 变量。

```
1  static  void  globalize_count(void)
2  {
3     globalcount  +=  counter;
4     counter  =  0;
5     globalreserve  -=  countermax;
6     countermax  =  0;
7  }
8
9  static  void  flush_local_count_sig(int  unused)
10  {
11     if  (ACCESS_ONCE(theft)  !=  THEFT_REQ)
12        return;
13     smp_mb();
14     ACCESS_ONCE(theft)  =  THEFT_ACK;
15     if  (!counting)  {
16        ACCESS_ONCE(theft)  =  THEFT_READY;
17     }
18     smp_mb();
19  }
20
21  static  void  flush_local_count(void)
22  {
23     int  t;
24     thread_id_t  tid;
25
26     for_each_tid(t,  tid)
27        if  (theftp[t]  !=  NULL)  {
28          if  (*countermaxp[t]  ==  0)  {
29             ACCESS_ONCE(*theftp[t])  =  THEFT_READY;
30             continue;
31          }
32          ACCESS_ONCE(*theftp[t])  =  THEFT_REQ;
33          pthread_kill(tid,  SIGUSR1);
```

```
34      }
35    for_each_tid(t, tid) {
36      if (theftp[t] == NULL)
37        continue;
38      while (ACCESS_ONCE(*theftp[t]) !=
                THEFT_READY) {
39        poll(NULL, 0, 1);
40        if (ACCESS_ONCE(*theftp[t]) == THEFT_REQ)
41          pthread_kill(tid, SIGUSR1);
42      }
43      globalcount += *counterp[t];
44      *counterp[t] = 0;
45      globalreserve -= *countermaxp[t];
46      *countermaxp[t] = 0;
47      ACCESS_ONCE(*theftp[t]) = THEFT_IDLE;
48    }
49  }
50
51  static void balance_count(void)
52  {
53    countermax = globalcountmax -
54                 globalcount - globalreserve;
55    countermax /= num_online_threads();
56    if (countermax > MAX_COUNTERMAX)
57      countermax = MAX_COUNTERMAX;
58    globalreserve += countermax;
59    counter = countermax / 2;
60    if (counter > globalcount)
61      counter = globalcount;
62    globalcount -= counter;
63  }
```

图 4.21：Signal-Theft 上限计数器的计数迁移函数

图 4.21 显示了负责在 per-thread 变量和全局变量之间迁移计数的函数。第 1-7 行是 global_count()，和之前的实现一样。第 9-19 行是 flush_local_count_sig()，它是窃取过程要用到的信号处理函数。第 11 和 12 行检查 theft 状态是否为 REQ，

如果不是，什么也不改变直接返回。第 13 行是一个内存屏障，确保对 theft 变量的采样在修改操作之前执行。第 14 行将 theft 状态设置为 ACK，如果第 15 行发现本线程的快速路径没有执行，第 16 行将 theft 状态设置为 READY。

小问题 4.40：在图 4.21 的 flush_local_count_sig()中，为什么要用 ACCESS_ONCE()封装对 per-thread 变量 theft 的读取？

第 21-49 行是 flush_local_count()，由慢速路径调用，刷新所有线程的本地计数。第 26-34 行的循环推进每个带有本地计数的线程对应的 theft 状态，同时给该线程发送一个信号。第 27 行跳过任何不存在的线程。对于存在的线程，第 28 行检查当前线程是否持有任何本地计数，如果没有，第 29 行将当前线程的 theft 状态设置为 READY，第 30 行跳过本次循环到另一个线程。如果有，第 32 行将当前线程的 theft 状态设置为 REQ，第 33 行给当前进程发送一个信号。

小问题 4.41：在图 4.21 中，为什么第 28 行直接访问其他线程的 countermax 变量是安全的？

小问题 4.42：在图 4.21 中，为什么第 33 行没有检查当前线程，来给自己发一个信号？

小问题 4.43：图 4.21 中的代码可以在 gcc 和 POSIX 下运行。如果要遵守 ISO C 标准，还需要做些什么？

第 35-48 行的循环等待每个线程达到 READY 状态，然后窃取线程的计数。第 36-37 行跳过任何不存在的线程，第 38-42 行的循环等待当前线程的 theft 状态变为 READY。第 39 行为避免优先级反转阻塞 1 毫秒，如果第 40 行判断线程的信号还没有到达，第 41 行重新发送信号。当执行到第 43 行时，线程的 theft 状态已经是 READY，所有第 43-46 行进行窃取。第 47 行将线程的 theft 状态设置回 IDLE。

小问题 4.44： 在图 4.21 中，为什么第 41 行要重新发送信号？

第 51-63 行是 balance_count()，和之前的例子类似。

```
1  int  add_count(unsigned  long  delta)
2  {
3     int  fastpath  =  0;
4
5     counting  =  1;
6     barrier();
7     if  (countermax  -  counter  >=  delta  &&
8          ACCESS_ONCE(theft)  <=  THEFT_REQ)  {
9        counter  +=  delta;
10       fastpath  =  1;
```

```
11    }
12    barrier();
13    counting = 0;
14    barrier();
15    if (ACCESS_ONCE(theft) == THEFT_ACK) {
16       smp_mb();
17       ACCESS_ONCE(theft) = THEFT_READY;
18    }
19    if (fastpath)
20       return 1;
21    spin_lock(&gblcnt_mutex);
22    globalize_count();
23    if (globalcountmax - globalcount -
24         globalreserve < delta) {
25       flush_local_count();
26       if (globalcountmax - globalcount -
27            globalreserve < delta) {
28          spin_unlock(&gblcnt_mutex);
29          return 0;
30       }
31    }
32    globalcount += delta;
33    balance_count();
34    spin_unlock(&gblcnt_mutex);
35    return 1;
36 }
37
38 int sub_count(unsigned long delta)
39 {
40    int fastpath = 0;
41
42    counting = 1;
43    barrier();
44    if (counter >= delta &&
45         ACCESS_ONCE(theft) <= THEFT_REQ) {
```

```
46        counter  -=  delta;
47        fastpath  =  1;
48    }
49    barrier();
50    counting  =  0;
51    barrier();
52    if  (ACCESS_ONCE(theft)  ==  THEFT_ACK)  {
53        smp_mb();
54        ACCESS_ONCE(theft)  =  THEFT_READY;
55    }
56    if  (fastpath)
57        return  1;
58    spin_lock(&gblcnt_mutex);
59    globalize_count();
60    if  (globalcount  <  delta)  {
61        flush_local_count();
62        if  (globalcount  <  delta)  {
63          spin_unlock(&gblcnt_mutex);
64          return  0;
65        }
66    }
67    globalcount  -=  delta;
68    balance_count();
69    spin_unlock(&gblcnt_mutex);
70    return  1;
71  }
```

图 4.22：Signal-Theft 上限计数器的加、减函数

图 4.22 的第 1-36 行是 add_count()函数。第 5-20 行是快速路径，第 21-35 行是慢速路径。第 5 行设置 per-thread 变量 counting 为 1，这样任何之后中断本线程的信号处理函数将会设置 theft 状态为 ACK，而不是 READY，这就允许快速路径可以恰当的完成。第 6 行阻止编译器进行乱序优化，确保快速路径执行体不会在设置 counting 之前执行。第 7 行和第 8 行检查是否 per-thread 数据可以容纳 add_count()的结果，以及是否当前没有任何正在进行的窃取过程，如果结果为真，第 9 行进行快速路径的加法，第 10 行表明执行的是快速路径。

无论上个判断结果如何，第 12 行阻止编译器对快速路径执行体的乱序优化，

之后是第 13 行，允许任何后继的信号处理函数进行窃取操作。第 14 行再一次禁止编译器乱序优化，然后第 15 行检查是否信号处理函数延迟将 theft 状态设置为 READY，如果是，第 16 行执行一条内存屏障，确保任何看见第 17 行设置 READY 状态的 CPU，也能看见第 9 行的效果。如果第 9 行的快速路径加法被执行了，那么第 20 行返回成功。

```
1  unsigned  long  read_count(void)
2  {
3     int  t;
4     unsigned  long  sum;
5
6     spin_lock(&gblcnt_mutex);
7     sum  =  globalcount;
8     for_each_thread(t)
9        if  (counterp[t]  !=  NULL)
10          sum  +=  *counterp[t];
11    spin_unlock(&gblcnt_mutex);
12    return  sum;
13 }
```

图 4.23：Signal-Theft 上限计数器的读函数

否则，我们进入从第 21 行开始的慢速路径。慢速路径的结构和之前的例子类似，所以对它的分析就留给读者当作练习了。同样地，第 38-71 行的 sub_count()函数和 add_count()一样，所以对 sub_count()的分析也留给读者了，图 4.23 对 read_count()的分析也留给读者。

```
1  void  count_init(void)
2  {
3     struct  sigaction  sa;
4
5     sa.sa_handler  =  flush_local_count_sig;
6     sigemptyset(&sa.sa_mask);
7     sa.sa_flags  =  0;
8     if  (sigaction(SIGUSR1,  &sa,  NULL)  !=  0)  {
9        perror("sigaction");
10       exit(-1);
11    }
12 }
```

```
13
14  void  count_register_thread(void)
15  {
16    int  idx  =  smp_thread_id();
17
18    spin_lock(&gblcnt_mutex);
19    counterp[idx]  =  &counter;
20    countermaxp[idx]  =  &countermax;
21    theftp[idx]  =  &theft;
22    spin_unlock(&gblcnt_mutex);
23  }
24
25  void  count_unregister_thread(int
                                  nthreadsexpected)
26  {
27    int  idx  =  smp_thread_id();
28
29    spin_lock(&gblcnt_mutex);
30    globalize_count();
31    counterp[idx]  =  NULL;
32    countermaxp[idx]  =  NULL;
33    theftp[idx]  =  NULL;
34    spin_unlock(&gblcnt_mutex);
35  }
```

图 4.24：Signal-Theft 上限计数器的初始化函数

图 4.24 的第 1-12 行是 count_init()，设置 flush_local_count_sig()为 SIGUSR1 的信号处理函数，让 flush_local_count()中的 pthread_kill()可以调用 flush_local_count_sig()。线程注册和撤销的代码与之前的例子类似，所以对它们的分析就留给读者了。

4.4.5. Signal-Theft 上限计数器讨论

signal-theft 的实现在我的 Intel Core Duo 笔记本比原子操作的实现快两倍。它总是这么好吗？

由于原子指令的相对缓慢，signal-theft 实现在 Pentium-4 处理器上比原子操

作好的多，但是后来，老式的基于 80386 的对称多处理器系统在原子操作实现的路径深度更短，原子操作的性能也随之提升。如果考虑最终的性能，你需要在实际部署应用程序的系统上测试这两种手段。

这就是为什么高质量的 API 如此重要的一个理由：它们让实现代码可以随着持续变动的硬件性能特征一起改变。

小问题 4.45： 如果想要一个只考虑精确下限的精确上限计数器，该如何实现？

4.5. 特殊的并行计数器

虽然第 4.4 节的精确上限计数器的实现非常有用，但是如果计数器的值总是在零附近变动，精确上限计数器就没什么用了，正如统计对 I/O 设备的访问计数一样。虑到我们一般并不关心当前有多少计数，这种统计值总在零附近变动的计数开销很大。正如我们在第 31 页所说的移动 I/O 设备访问计数问题。访问次数在除了有人想可插拔设备以外的情况是完全不重要的，而这种例外情况本身又是很少见的。

一种简单的解决办法是，为计数器增加一个很大的“偏差值”（比如 10 亿），确保计数器的值远离零，让计数器可以有效工作。当有人想拔出设备时，计数器又减去“偏差值”。统计最后几次访问将是非常低效的，但是对之前所有访问的统计却可以全速进行。

小问题 4.46： 当使用加上偏差的计数器时，还需要去做什么事情？

虽然加上偏差的计数器既有效又有用，但这只是第 31 页的可插拔 I/O 设备访问计数问题的部分解决办法。当尝试拔出设备时，我们不仅需要知道当前精确的 I/O 访问次数，还需要从现在开始阻止未来的访问请求。一种解决的办法是在更新计数器时使用读写锁的读锁，在读取计数器时使用同一把读写锁的写锁。执行 I/O 的代码如下：

```
1  read_lock(&mylock);
2  if (removing) {
3      read_unlock(&mylock);
4      cancel_io();
5  } else {
6      add_count(1);
7      read_unlock(&mylock);
8      do_io();
9      sub_count(1);
10 }
```

第 1 行获取读锁，第 3 行和第 7 行释放它。第 2 行检查是否将要拔出设备，如果是，第 3 行释放读锁，第 4 行取消 I/O，或者随便执行一些在将要拔出设备时应该采用的操作。如果不是，第 6 行增加访问计数，第 7 行释放读锁，第 8 行执行 I/O，第 9 行减少访问计数。

小问题 4.47： 简直太荒谬了！用读锁来更新计数？你在玩什么把戏？？？

拔出设备的代码如下：

```
1  write_lock(&mylock);
2  removing = 1;
3  sub_count(mybias);
4  write_unlock(&mylock);
5  while (read_count() != 0) {
6      poll(NULL, 0, 1);
7  }
8  remove_device();
```

第 1 行获取写锁，第 4 行释放。第 2 行表明将要拔出设备，第 5 到 7 行的循环等待所有 I/O 操作完成。最后第 8 行进行任何在拔出设备前需要执行的准备工作。

小问题 4.48： 如果是真实的系统，还需要考虑哪些问题？

4.6. 并行计数的讨论

本章展示了传统计数原语会遇见的问题：可靠性、性能、可扩展性。C 语言的++操作符不能在多线程代码中保证函数的可靠性，对单个变量的原子操作性能不好，可扩展性也差。本章还展示了一系列在特殊情况下性能和可扩展性俱佳的计数算法。

表 4.1：统计计数器在 Power5 机器上的性能

			读端原语的效率	
算法	章节	写效率	1 核	64 核
Count_stat.c	4.2.2	40.4 ns	220 ns	220 ns
Count_end.c	4.2.4	6.7 ns	521 ns	205.000 ns
Count_end_rcu.c	9.1	6.7 ns	481 ns	3.700 ns

表 4.1 给出了三种并行统计计数算法的性能。三种算法在更新计数上都有极佳的线性扩展能力。per-thread 变量实现的更新计数上比基于数组的实现快很多，但在读取计数上较慢，并且在有多个并发读者时存在严重的锁竞争。这种竞争可以用第 8 章介绍的方法解决，如表 4.1 的最后一排。

小问题 4.49： 在表 4.1 的 count_stat.c 一排中，我们可以看到，更新计数的性能随着线程数增加而线性扩展。这怎么可能呢？毕竟线程更多，要加的 per-thread 计数器也更多。

小问题 4.50： 表 4.1 上的统计计数器算法的读端原语性能实在是惨不忍睹，就算是表 4.1 的最后一排也没挣回点面子，它们到底有什么用？

图 4.2 是并行上限计数算法的性能。对上限计数的精确性要求带来了不小的性能损失，虽然在 Power 5 系统上这种损失可以通过用读写方的信号替换更新方的原子操作来减轻。所有这些实现面对并发读者时，都存在读写方的锁竞争问题。

表 4.2：上限计数器在 Power5 机器上的性能

				读端原语的效率	
算法	章节	精确吗？	写效率	1 核	64 核
Count_lim.c	4.3.2	N	9.7 ns	517 ns	202.000 ns
Count_lim_app.c	4.3.4	N	6.6 ns	520 ns	205.000 ns
Count_lim_atomic.c	4.4.1	Y	56.1 ns	606 ns	166.000 ns
Count_lim_sig.c	4.4.4	Y	17.5 ns	520 ns	205.000 ns

小问题 4.51： 根据表 4.2 中的性能数据，我们应该尽量用更新方的信号而不是读写方的原子操作，对吗？

小问题 4.52： 能不能采用先进的技术解决表 4.2 中的锁竞争问题？

这些算法只在它们特定的领域运转良好，这可以说是并行计算的一个主要问题。毕竟，C 语言的++操作符在所有单线程程序中都运行的很好，不是只在特定领域，对吧？

上述推论有一点道理，但在本质上是误导读者的。我们提到的问题不仅是并行性，更是可扩展性。要弄明白这个，先来看看 C 语言的++操作符。事实上++操作符并不是“总能”工作，而只对一定范围的数字而言有效。假如您要处理 1000 位的十进制数，C 语言的++操作符对您来说就没用了。

小问题 4.53： ++操作符在 1000 位的数字上工作的很好！你没听过操作符重载吗？？？

我们提到的问题也不专属于算术。假设您需要存储和查询数据。您应该使用 ASCII 文件、XML、关系数据库、链表、紧凑数组、B 树、基树或者其他什么数据结构和环境来让数据可以被存储和查询吗？这取决于您需要做什么，您需要它做的有多快，还有您的数据集有多大。

同样地，如果您需要计数，您的方案取决于您要统计的数有多大、有多少个 CPU 并发操纵计数、如何使用计数，以及您所需要的性能和扩展性的程度。

这个问题也不仅限于软件。只需要简单的铺一块木板，就是一座让人跨过小溪的桥。但是木板的办法无法扩展。您不能用一块木板横跨数公里宽的 Columbia

River 河口，也不能用于承载卡车的桥。简而言之，桥的设计必须随着跨度和负载而改变，同样，软件的设计也许要随着 CPU 数的增加而改变。

本章的例子显示了 large numbers of CPUs to be brought to bear 的重要工具是“分割”。完全分割，比如第 4.2 节的统计计数器，或者部分分割，比如第 4.3 和 4.4 节讨论的上限计数器。下一章将对分割进行更加深入的讨论。

小问题 4.54： 但是如果我们能分割任何事物，为什么还要被共享内存的多线程所困扰？为什么不完全分割问题，然后像多进程一样运行，每个子问题都有自己的地址空间？

5. 分割和同步设计

在商用计算机中，多核系统已经越来越常见了，本章将描述如何设计能更好利用多核优势的软件。我们将介绍一些习语，或者叫“设计模式”，来帮助您权衡性能、可扩展性和响应时间。在上一章我们说过，您在编写并行软件时最重要的考虑是如何进行分割。正确的分割问题能够让解决办法简单、可扩展并且高性能，而不恰当的分割问题则会产生缓慢且复杂的解决方案。

5.1. 分割练习

本节用两个例子（哲学家就餐问题和双端队列问题）作为练习，来说明分割的价值。

5.1.1. 哲学家就餐问题

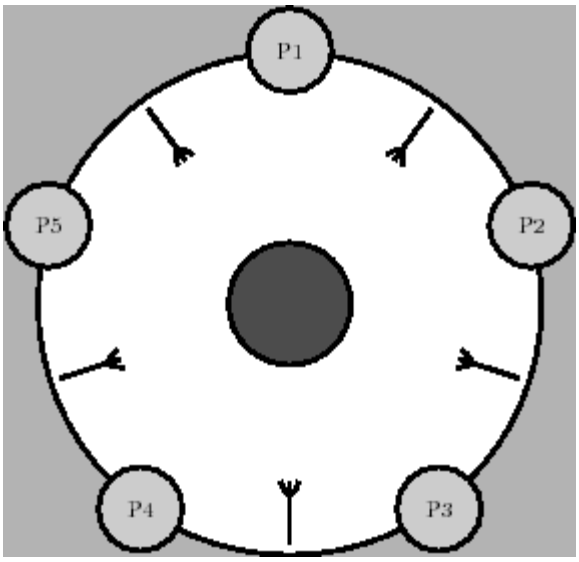


图 5.1：哲学家就餐问题

图 5.1 是经典的哲学家就餐问题的示意图[Dij71]。问题中的五个哲学家一天无所事事，不是思考就是吃一种需要用两把叉子才能吃下的“滑溜溜的”意大利面。每个哲学家只能用和他左手和右手旁的叉子，一旦哲学家拿起了叉子，那么不吃到心满意足是不会放下的。

我们的目标是构建一种算法来——有点儿文学化——阻止饥饿。一种饥饿的

场景是所有哲学家都同时拿起了左手边的叉子。因为他们在吃饱前不会放下叉子，并且他们还需要第二把叉子才能开始就餐，所以所有哲学家都会挨饿。

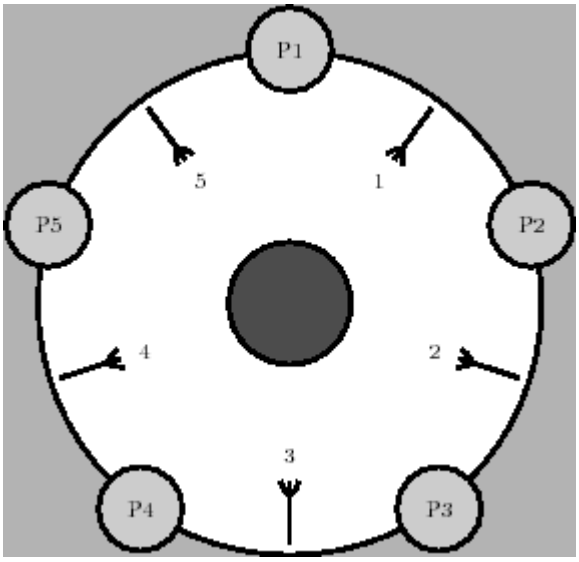


图 5.2：哲学家就餐问题，教科书解法

Dijkstra 的解决方法是使用一个全局信号量，假设通信延迟忽略不计的话，这种方法非常完美，可是在随后的几十年里这种假设变得无效了。因此，最近的解决办法是像图 5.2 一样为叉子编号。每个哲学家都先拿他盘子周围编号最小的叉子，然后再拿编号最高的叉子。这样坐在图中最上方的哲学家会先拿起左手边的叉子，然后是右边的叉子，而其他的哲学家则先拿起右手边的叉子。因为有两个哲学家试着去拿叉子 1，而只有一位会成功，所以只有 4 位哲学家抢 5 把叉子。至少这 4 位中的一位肯定能拿到两把叉子，这样就能开始就餐了。

这种为资源编号并按照编号顺序获取资源的通用技术在防止死锁上经常使用。但是很容易就能想象出一个事件序列来产生这种结果：虽然大家都在挨饿，但是一次只有一名哲学家就餐。

1. P2 拿起叉子 1，阻止 P1 拿起叉子。
2. P3 拿起叉子 2。
3. P4 拿起叉子 3。
4. P5 拿起叉子 4。
5. P5 拿起叉子 5，开始就餐。
6. P5 放下叉子 4 和 5。
7. P4 拿起叉子 4，开始就餐。

请在进一步阅读之前，思考哲学家就餐问题的分割方法。

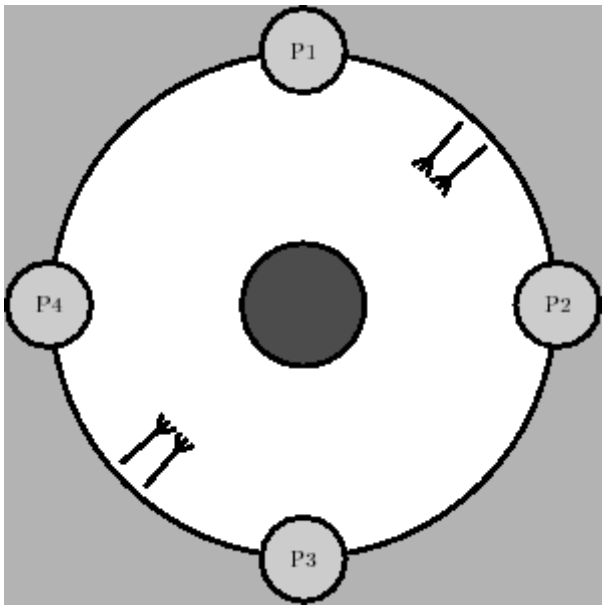


图 5.3：哲学家就餐问题，分割解法

图 5.3 是一种方法，里面只有 4 位而不是 5 位哲学家，这样可以更好的说明分割技术。最上方和最右边的哲学家合用一对叉子，而最下方和最左边的哲学家合用一对叉子。如果所有哲学家同时感觉饿了，至少有两位能同时就餐。另外如图中所示，现在叉子可以捆绑成一对儿，这样可以同时拿起或者放下，这就简化了获取和释放算法。

小问题 5.1： 哲学家就餐问题还有更好的解法吗？

这是“水平并行化”[Inm85]的一个例子，或者叫“数据并行化”，这么叫是因为哲学家之间没有依赖关系。在数据处理型的系统中，“数据并行化”是指一种类型的数据只会穿过一系列同类型软件组件其中的一个。

小问题 5.2：那么“水平并行化”里的“水平”是什么意思呢？

5.1.2. 双端队列

双端队列是一种元素可以从两端插入或删除的数据结构[Knu73]。据说实现一种基于锁的允许在双端队列的两端进行并发操作的方法非常困难[Gro07]。本节将展示一种分割设计策略，能实现合理且简单的解决方案，请看下面的小节中的三种通用方法。

5.1.2.1. 右手锁和左手锁

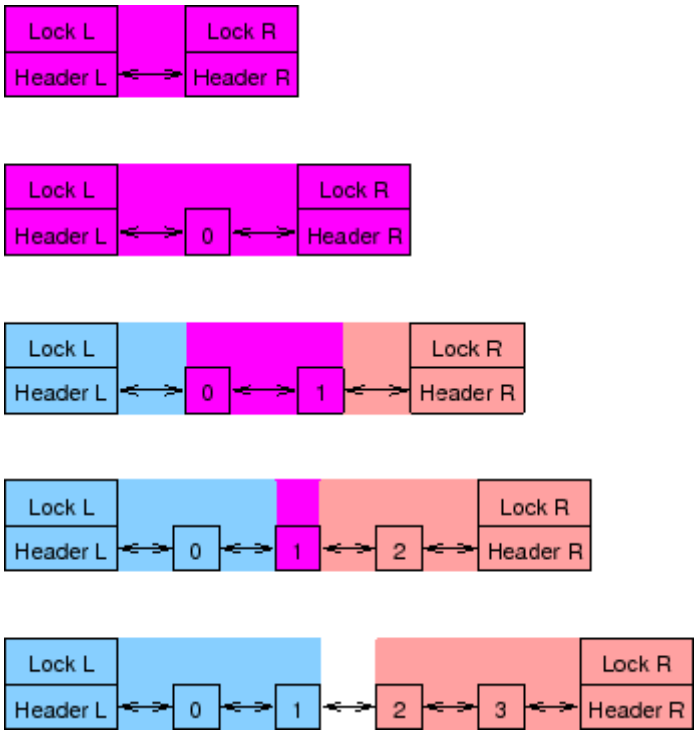


图 5.4：带有左手锁和右手锁的双端队列

右手锁和左手锁是一种看起来很直接的办法，为左手端的入列操作加一个左手锁，为右手端的出列操作加一个右手锁，如图 5.4 所示。但是，这种办法的问题是当队列中的元素不足四个时，两个锁的范围会发生重叠。这种重叠是由于移除任何一个元素不仅只影响元素本身，还要影响它左边和右边相邻的元素。这种范围在图中被涂上了颜色，蓝色表示左手锁的范围，红色表示右手锁的范围，紫色表示重叠的范围。虽然创建这样一种算法是可能的，但是至少要小心 5 种特殊情况，尤其是在队列另一端的并发活动会让队列随时可能从一种特殊情况变为另一种特殊情况。所以最好还是考虑考虑其他解决方案。

5.1.2.2. 复合双端队列

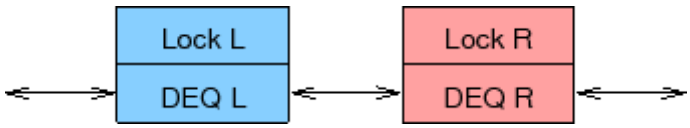


图 5.5：复合双端队列

图 5.5 是一种强制锁范围不重叠的办法。两个单独的双端队列串联在一起，每个队列用自己的锁保护。这意味着数据偶尔会从一个双端队列跑到另一个双端队列。此时必须同时持有两把锁。为避免死锁，可以使用一种简单的锁层级关系，

比如，在获取右手锁前先获取左手锁。这比在同一双端队列上用两把锁简单的多，因为我们可以无条件地让左边的入列元素进入左手队列，右边的入列元素进入右手队列。主要的复杂度来源于从空队列中出列，在这种情况下必须：

1. 如果持有右手锁，释放并获取左手锁，重新检查队列是否仍然为空。
2. 获取右手锁。
3. 重新平衡穿过两个队列的元素。
4. 移除指定的元素。
5. 释放两把锁。

小问题 5.3： 在复合双端队列实现中，如果队列在释放和获取锁时变的不为空，那么该怎么办？

重新平衡操作可能会将某个元素在两个队列间来回移动，这不仅浪费时间，而且想要获得最佳性能，还需针对工作负荷进行不断微调。虽然这在一些情况下可能是最佳方案了，但是带着更大的雄心，我们还将继续探索其它算法。

5.1.2.3. 哈希双端队列

哈希永远是分割一个数据结构的最简单和最有效的方法。可以根据元素在队列中的位置为每个元素分配一个序号，然后以此对双端队列进行哈希，这样第一个从左边进入空队列的元素编号为 0，第一个从右边进入空队列的元素编号为 1。其他从左边进入只有一个元素的队列的元素则递减的编号（-1，-2，-3，...），而其他从右边进入只有一个元素的队列的元素则递增的编号（2，3，4，...）。关键的是，实际上不用真正的为元素编号，元素的序号暗含在它在队列中的位置中。

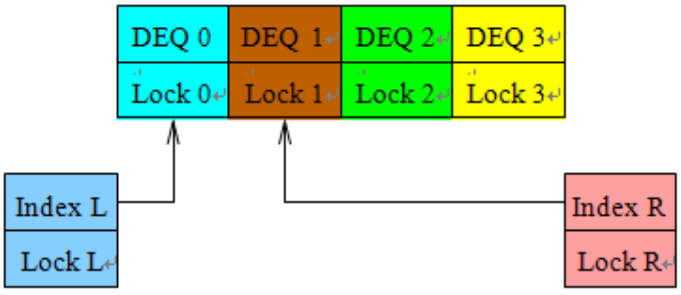


图 5.6：哈希双端队列

我们用一个锁保护左手的下标，用另一个锁保护右手的下标，再各用一个锁保护对应的哈希链表。图 5.6 显示了四个哈希链表的数据结构。注意到锁的范围没有重叠，为了避免死锁，只在获取链表锁之前获取下标锁，每种类型的锁（下标或者链表）每次从不获取超过一个。

每个哈希链表都是一个双端队列，在这里的例子中，每个链表都 holds every

fourth element。图 5.7 中最上面的部分是“R1”元素从右边入队后的状态，右手的下标增加，用来引用哈希链表 2。图中中间部分是又有三个元素从右边入队。正如您见到的那样，下标回到了它们初始的状态，但是每个哈希队列现在是非空的了。图中下方部分是另外三个元素从左边入队，而另外一个元素从右边入队后的状态。

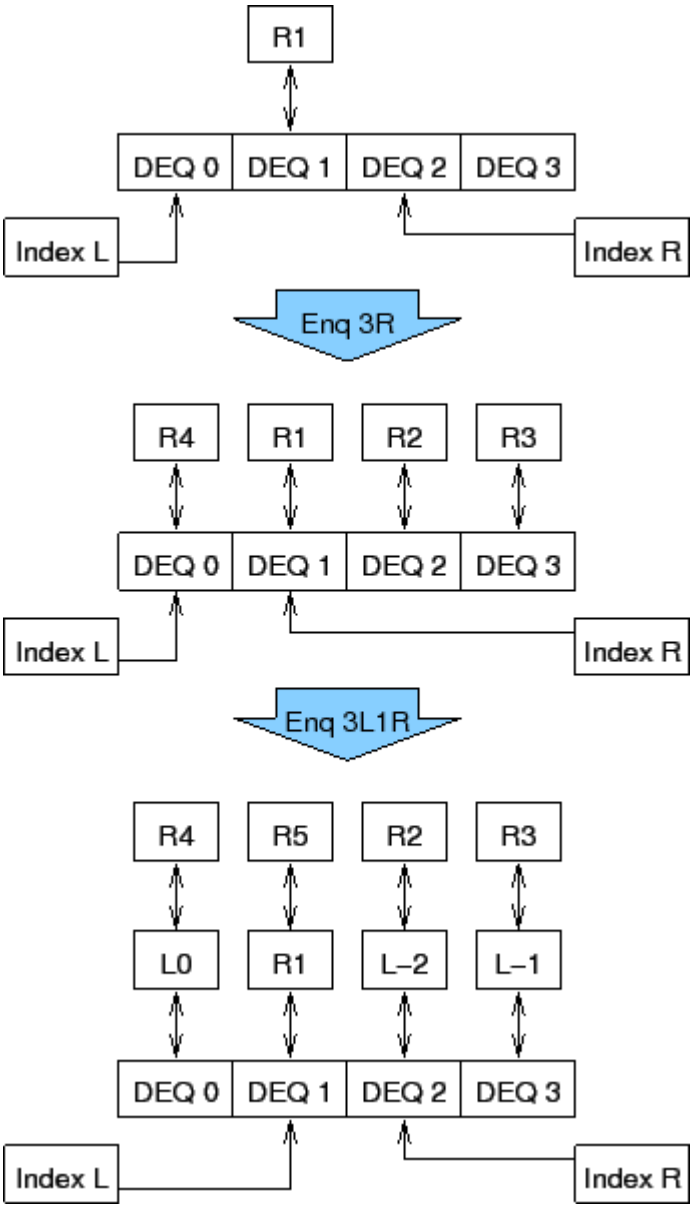


图 5.7：插入后的哈希双端队列

从图 5.7 中最后一个状态可以看出，左出队操作将返回元素“L-2”，让左手下标指向哈希链 2，此时该链表只剩下“R2”。在这种状态下，并发的左入队操作和右入队操作可能会导致锁竞争，但这种锁竞争发生的可能性可以通过使用更大的哈希表来降低。

R7	R6	R5	R4
L0	R1	R2	R3
L–4	L–3	L–2	L–1
L–8	L–7	L–6	L–5

图 5.8：12 个元素的哈希双端队列

图 5.8 显示了 12 个元素如何组成一个有 4 个并行哈希桶的双端队列。

```
1  struct  pdeq  {
2     spinlock_t  llock;
3     int  lidx;
4     spinlock_t  rlock;
5     int  ridx;
6     struct  deq  bkt[DEQ_N_BKTS];
7  };
```

图 5.9：基于锁的并行双端队列数据结构

图 5.9 显示了对应的 C 语言数据结构，假设已有 struct deq 来提供带有锁的双端队列实现。这个数据结构包括第 2 行的左手锁，第 3 行的左手下标，第 4 行的右手锁，第 5 行的右手下标，以及第 6 行的哈希后的基于简单锁实现的双端队列数组。高性能的实现当然还会使用填充或者是特殊对齐指令来避免 false sharing（http://en.wikipedia.org/wiki/False_sharing）。

```
1  struct  element  *pdeq_dequeue_l(struct  pdeq  *d)
2  {
3     struct  element  *e;
4     int  i;
5
6     spin_lock(&d->llock);
7     i  =  moveright(d->lidx);
8     e  =  deq_dequeue_l(&d->bkt[i]);
9     if  (e  !=  NULL)
10        d->lidx  =  i;
11    spin_unlock(&d->llock);
12    return  e;
13 }
14
15 void  pdeq_enqueue_l(struct  element  *e,  struct
                        pdeq  *d)
```

```
16 {
17   int  i;
18
19   spin_lock(&d->llock);
20   i  =  d->lidx;
21   deq_enqueue_l(e,  &d->bkt[i]);
22   d->lidx  =  moveleft(d->lidx);
23   spin_unlock(&d->llock);
24 }
25
26 struct  element  *pdeq_dequeue_r(struct  pdeq  *d)
27 {
28   struct  element  *e;
29   int  i;
30
31   spin_lock(&d->rlock);
32   i  =  moveleft(d->ridx);
33   e  =  deq_dequeue_r(&d->bkt[i]);
34   if  (e  !=  NULL)
35      d->ridx  =  i;
36   spin_unlock(&d->rlock);
37   return  e;
38 }
39
40 void  pdeq_enqueue_r(struct  element  *e,
                       struct  pdeq  *d)
41 {
42   int  i;
43
44   spin_lock(&d->rlock);
45   i  =  d->ridx;
46   deq_enqueue_r(e,  &d->bkt[i]);
47   d->ridx  =  moveright(d->lidx);
48   spin_unlock(&d->rlock);
49 }
```

图5.10：基于锁的并行双端队列实现代码

图 5.10 显示了入队和出队函数。讨论将集中在左手操作上，因为右手的操作都是源于左手操作。

第 1-13 行是 pdeq_dequeue_l()函数，从左边出队，如果成功返回一个元素，如果失败返回 NULL。第 6 行获取左手自旋锁，第 7 行计算要出队的下标。第 8 行让元素出队，如果第 9 行发现结果不为 NULL，第 10 行记录新的左手下标。不管结果为何，第 11 行释放锁，最后如果曾经有一个元素，第 12 行返回这个元素，否则返回 NULL。

第 15-24 行是 pdeq_enqueue_l()函数，从左边入队一个特定元素。第 19 行获取左手锁，第 20 行 pick up 左手下标。第 21 行让元素从左边入队，进入一个以左手下标标记的双端队列。第 22 行更新左手下标，最后第 23 行释放锁。

和之前提到的一样，右手操作完全是对对应的左手操作的模拟。

小问题 5.4：哈希过的双端队列是一种好的解决方法吗？如果对，为什么对？如果不对，为什么不对？

5.1.2.4. 再次回到复合双端队列

本节再次回到复合双端队列，准备使用一种重新平衡机制来将非空队列中的所有元素移动到空队列中。

小问题 5.5：让所有元素进入空的队列？这种脑残的方法是哪门子最优方案啊？？？

相比上一节提出的哈希式实现，复合式实现建立在对既不使用锁也不使用原子操作的双端队列的顺序实现上。

```
1  struct  list_head  *pdeq_dequeue_l(struct  pdeq  *d)
2  {
3     struct  list_head  *e;
4     int  i;
5
6     spin_lock(&d->llock);
7     e  =  deq_dequeue_l(&d->ldeq);
8     if  (e  ==  NULL)  {
9        spin_lock(&d->rlock);
10        e  =  deq_dequeue_l(&d->rdeq);
11        list_splice_init(&d->rdeq.chain,
                           &d->ldeq.chain);
```

```
12        spin_unlock(&d->rlock);
13    }
14    spin_unlock(&d->llock);
15    return e;
16 }
17
18 struct list_head *pdeq_dequeue_r(struct pdeq *d)
19 {
20    struct list_head *e;
21    int i;
22
23    spin_lock(&d->rlock);
24    e = deq_dequeue_r(&d->rdeq);
25    if (e == NULL) {
26        spin_unlock(&d->rlock);
27        spin_lock(&d->llock);
28        spin_lock(&d->rlock);
29        e = deq_dequeue_r(&d->rdeq);
30        if (e == NULL) {
31            e = deq_dequeue_r(&d->ldeq);
32            list_splice_init(&d->ldeq.chain,
                               &d->rdeq.chain);
33        }
34        spin_unlock(&d->llock);
35    }
36    spin_unlock(&d->rlock);
37    return e;
38 }
39
40 void pdeq_enqueue_l(struct list_head *e,
                       struct pdeq *d)
41 {
42    int i;
43
44    spin_lock(&d->llock);
```

```
45     deq_enqueue_l(e,  &d->ldeq);
46     spin_unlock(&d->llock);
47  }
48
49  void  pdeq_enqueue_r(struct  list_head  *e,
                         struct  pdeq  *d)
50  {
51     int  i;
52
53     spin_lock(&d->rlock);
54     deq_enqueue_r(e,  &d->rdeq);
55     spin_unlock(&d->rlock);
56  }
```

图 5.11：复合并行双端队列的实现代码

图 5.11 展示了这个实现。和哈希式实现不同，复合式实现是非对称的，所以我们必须单独考虑 pdeq_dequeue_l()和 pdeq_dequeue_r()的实现。

小问题 5.6：为什么复合并行双端队列的实现不能是对称的？

图 5.11 第 1-16 行是 pdeq_dequeue_l()的实现。第 6 行获取左手锁，第 14 行释放。第 7 行尝试从双端队列的左端左出列一个元素，如果成功，跳过第 8-13 行，直接返回该元素。否则，第 9 行获取右手锁，第 10 行从队列右端左出列一个元素，第 11 行将右手队列中剩余元素移至左手队列，第 12 行释放右手锁。如果有的话，最后将返回第 10 行出列的元素。

图 5.11 第 18-38 行是 pdeq_dequeue_r()的实现。和之前一样，第 23 行获取右手锁（第 36 行释放），第 24 行尝试从右手队列右出列一个元素，如果成功，跳过第 24-35 行，直接返回该元素。但是如果第 25 行发现没有元素可以出列，那么第 26 行释放右手锁，第 27-28 行以恰当的顺序获取左手锁和右手锁。然后第 29 行再一次尝试从右手队列右出列一个元素，如果第 30 行发现第二次尝试也失败了，第 31 行从左手队列（假设只有一个元素）右出列一个元素，第 32 行将左手队列中的剩余元素移至右手队列。最后，第 34 行释放左手锁。

小问题 5.7：为什么图 5.11 中第 29 行的重试右出列操作是必须的？

小问题 5.8：可以肯定的是，左手锁必须在某些时刻是可用的!!! 那么，为什么图 5.11 中第 26 行的无条件释放右手锁是必须的？

图 5.11 中第 40-47 行是 pdeq_enqueue_l()的实现。第 44 行获取左手自旋锁，第 45 行将元素左入列到左手队列，最后第 46 行释放自旋锁。pdeq_enqueue_r（图中第 49-56）的实现和此类似。

5.1.2.5. 关于双端队列的讨论

复合式实现在某种程度上比第 5.1.2.3 节所描述的哈希式实现复杂，但是仍然属于比较简单的。当然，更智能的重新平衡机制可能更加复杂，但是这里展现的简单机制和另一种软件[DCW+11]相比，已经执行的很好了，即使和使用硬件辅助的算法[DLM+10]相比也是如此。不过，从这种机制中我们最好也只能获得 2x 的扩展能力，因为最多只能有两个线程并发地持有出列的锁。 关键点在于从共享队列中入列或者出列的巨大开销。

5.1.3. 关于分割问题示例的讨论

第 5.1.1 节的小问题中，关于哲学家就餐问题最优解法的答案是“水平”并行化或者“数据并行化”的极佳例子。在这个例子中，同步开销接近于 0（或者就是）。相反，双端队列的实现是“垂直并行化”或者“管道”的极佳例子，因为数据从一个线程转移到另一个线程。管道需要密切的合作，因此需要更多的工作来获得某种程度的效率。

小问题 5.9：串联双端队列比哈希双端队列运行快两倍，即使我将哈希表大小增加到非常大也是如此。为什么会这样？

小问题 5.10：有没有一种更好的方法，能并发地处理双端队列？

这两个例子显示了分割在并行算法上的巨大威力。但是，这些例子还渴求更多和更好的并行程序设计准则，下一节将讨论此话题。

5.2. 设计准则

第 1.2 节给出了三个并行编程的目标：性能、生产率和通用性。但是还需要更详细的设计准则来真正的指导真实世界中的设计，这就是本节将解决的任务。在真实世界中，这些准则经常在某种程度上冲突，这需要设计者小心的权衡得失。

这些准则可以被认为是设计中的阻力，对这些阻力进行恰当权衡，这就称为“设计模式”[Ale79]，[GHJV95]。

基于三个并行编程目标的设计准则是加速、竞争、开销、读写比率和复杂性。

加速：如第 1.2 节所讲，性能增加是花费如此多时间和精力进行并行化的主要原因。加速的定义是运行程序的顺序执行版本所需要的时间，比上执行并行版本所需时间的比例。

竞争：如果对于一个并行程序来说，增加更多的 CPU 并不能让程序忙起来，那么多出来的 CPU 是因为竞争的关系而无法有效工作。可能是锁竞争、内存竞

争或者其他什么性能杀手的原因。

工作-同步比率：单处理器、单线程、不可抢占、不可中断版本的并行程序完全不需要任何同步原语。因此，任何消耗在这些原语上（通信中的 cache miss、消息延迟、加解锁原语、原子指令和内存屏障等）的时间都是对程序意图完成的工作没有直接帮助的开销。重要的衡量准则是同步开销与临界区中代码的开销之间的关系，更大的临界区能容忍更大的同步开销。工作-同步开销比率与同步效率的概念有关。

读-写比率：极少更新的数据结构更多是复制而不是分割，并且用非对称的同步原语来保护，以提高写者同步开销的代价来降低读者的同步开销。对频繁更新的数据结构的优化也是可以的，详见第 4 章的讨论。

复杂性：并行程序比相同的顺序执行程序复杂，这是因为并行程序要比顺序执行程序维护更多状态，虽然这些状态在某些情况下（有规律并且结构化的）理解起来很容易。并行程序员必须考虑同步原语、消息传递、锁的设计、临界区识别以及死锁等诸多问题。

更大的复杂性通常转换成了更高的开发和维护代价。因此，代码预算可以有效限制对现有程序修改的数目和类型，因为原有程序在一定程度的加速只值这么多的时间和精力。进一步说，还有可能对顺序执行程序进行一定程度的优化，这比并行化更廉价、更有效。如第 1.2.1 节所说，并行化只是众多优化中的一种，并且只是一种主要应用于 CPU 是瓶颈的优化。

这些准则合在一起，让程序达到了最大程度的加速。前三个准则相互交织在一起，所以本节剩下的部分将分析这三个准则的交互关系。

请注意，这些准则也是需求说明的一部份。必须，加速既是愿望（“越快越好”），又是工作负荷的绝对需求，或者说是“运行环境”（“系统必须至少支持每秒 1 百万次的 web 点击”）。

理解这些设计准则之间的关系，对于权衡并行程序的各个设计目标十分有用。

1. 程序在临界区上所花的时间越少，潜在的加速就越大。这是 Amdahl 定律的结果，这也是因为在一个时刻只能只有一个 CPU 进入临界区的原因。
2. 程序在某个互斥的临界区上所耗费的时间必须大大小于 CPU 数的倒数，因为这样增加 CPU 数目才能达到事实上的加速。比如在 10 个 CPU 上运行的程序只能在关键的临界区上花费小于 1/10 的时间，这样扩展性才好。
3. 竞争效应所浪费的多余 CPU 和/或者本来该是加速的墙上时间，应该少于可用 CPU 的数目。CPU 数和实际的加速之间的差距越大，CPU 使用的效率越低。同样，需要的效率越高，可以做到的加速就越小。

4. 如果可用的同步原语相较它们保护的临界区来说开销太大，那么加速程序运行的最佳办法是减少调用这些原语的次数（比如分批进入临界区、数据所有权、RCU 或者使用粒度更粗的设计——比如代码锁）。
5. 如果临界区相较守护这块临界区的原语来说开销太大，那么加速程序运行的最佳办法是增加程序的并行化程度，比如使用读写锁、数据锁、RCU 或者数据所有权。
6. 如果临界区相较守护这块临界区的原语来说开销太大，并且被守护的数据结构读多于写，那么加速程序运行的最佳办法是增加程序的并行化程度，比如读写锁或者 RCU。
7. 各种增加 SMP 性能的改动，比如减少锁竞争程度，能改善响应时间。

5.3. 同步粒度

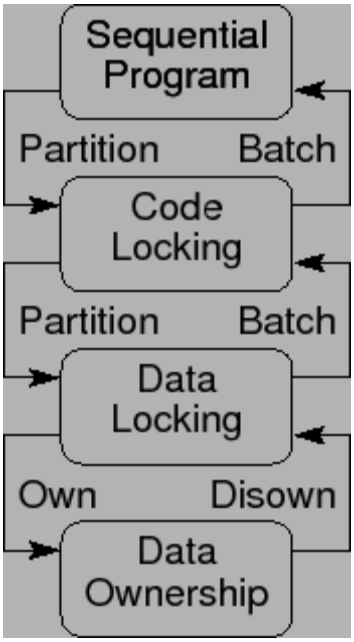


图 5.12：设计模式与锁粒度

图 5.12 是不同程度同步粒度的图形表示。每一种同步粒度都用一节内容来描述。下面几节主要关注锁，不过其他几种同步方式也有类似的粒度问题。

5.3.1. 串行程序

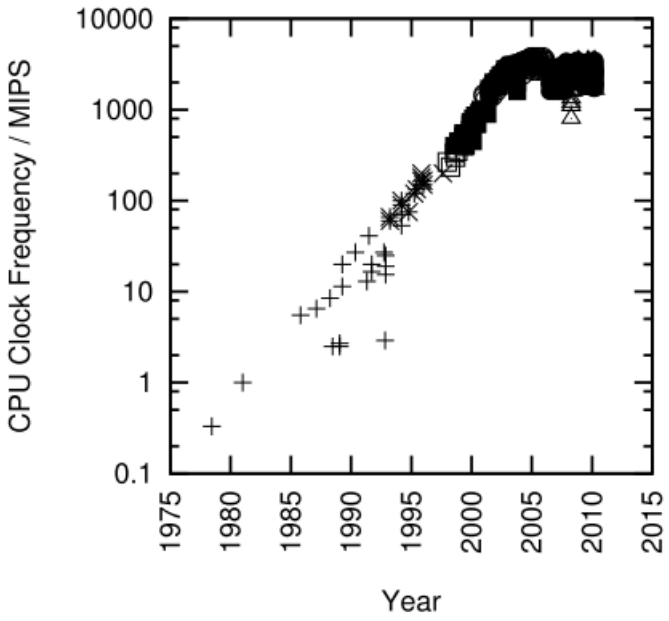


图 5.13：Intel 处理器的 MIPS/时钟频率变化趋势

如果程序在单处理器上运行的足够快，并且不与其他进程、线程或者中断处理程序发生交互，那么您可以将代码中所有的同步原语删掉，远离它们所带来的开销和复杂性。好多年前曾有人争论摩尔定律最终会让所有程序都变得如此。但是，随着 2003 年以来 Intel CPU 的 CPU MIPS 和时钟频率增长速度的停止，见图 5.13，此后要增加性能，就必须提高程序的并行化程度。关于是否这种趋势会导致一块芯片上集成几千个 CPU 的争论不会很快停息，但是考虑到 Paul 是在一台双核笔记本上敲下这句话的，SMP 的寿命极有可能比你我都长。另一个需要注意的地方是以太网的带宽持续增长，如图 5.14 所示。这种增长会导致多线程服务器的产生，这样才能有效处理通信载荷。

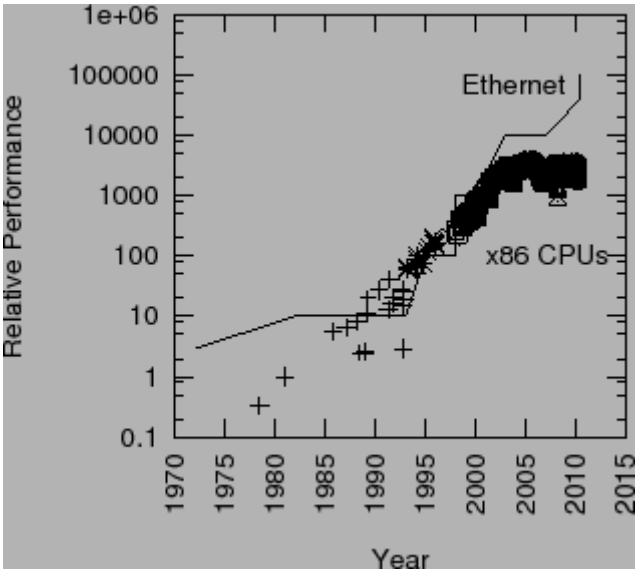


图 5.14；以太网带宽 v.s. Intel x86 处理器的性能

请注意，这并不意味这您应该在每个程序中使用多线程方式编程。我再一次说明，如果一个程序在单处理器上运行的很好，那么您就从 SMP 同步原语的开销和复杂性中解脱出来吧。图 5.15 中哈希表查找代码的简单之美强调了这一点。

```
1  struct  hash_table
2  {
3     long  nbuckets;
4     struct  node  **buckets;
5  };
6
7  typedef  struct  node  {
8     unsigned  long  key;
9     struct  node  *next;
10  }  node_t;
11
12  int  hash_search(struct  hash_table  *h,  long  key)
13  {
14     struct  node  *cur;
15
16     cur  =  h->buckets[key  %  h->nbuckets];
17     while  (cur  !=  NULL)  {
18        if  (cur->key  >=  key)  {
19           return  (cur->key  ==  key);
20        }
```

```
21        cur = cur->next;
22    }
23    return 0;
24 }
```

图5.15：串行版的哈希表搜索算法

5.3.2. 代码锁

代码锁是最简单的设计，只使用全局锁。在已有的程序上使用代码锁，可以很容易的让程序可以在多处理器上运行。如果程序只有一个共享资源，那么代码锁的性能是最优的。但是，许多较大且复杂的程序会在临界区上执行许多次，这就让代码锁的扩展性大大受限。

```
1  spinlock_t hash_lock;
2
3  struct hash_table
4  {
5    long nbuckets;
6    struct node **buckets;
7  };
8
9  typedef struct node {
10    unsigned long key;
11    struct node *next;
12 } node_t;
13
14 int hash_search(struct hash_table *h, long key)
15 {
16    struct node *cur;
17    int retval;
18
19    spin_lock(&hash_lock);
20    cur = h->buckets[key % h->nbuckets];
21    while (cur != NULL) {
22       if (cur->key >= key) {
23          retval = (cur->key == key);
24          spin_unlock(&hash_lock);
```

```
25          return  retval;
26        }
27        cur  =  cur->next;
28    }
29    spin_unlock(&hash_lock);
30    return  0;
31  }
```

图 5.16：基于代码锁的哈希表搜索算法

因此，您最好在只有一小段执行时间在临界区程序，或者对扩展性要求不高的程序上使用代码锁。这种情况下，代码锁可以让程序相对简单，和单线程版本类似，如图 5.16 所示。但是，和图 5.15 相比，hash_search()从简单的一行 return 变成了 3 行语句，因为在返回前需要释放锁。

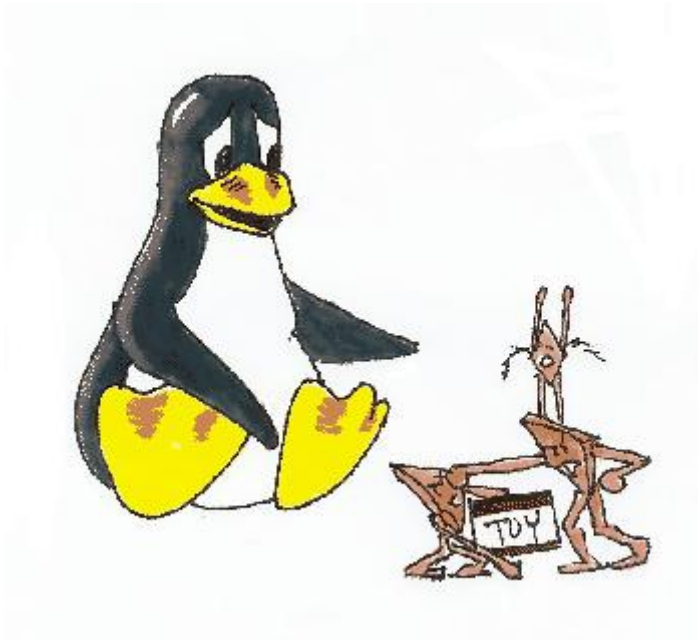


图 5.17：锁竞争

并且，代码锁尤其容易引起“锁竞争”，一种多个 CPU 并发访问同一把锁的情况。照顾一群小孩子（或者像小孩子一样的老人）的 SMP 程序员肯定能马上意识到某样东西只有一个的危险，如图 5.17 所示。

该问题的一种解决办法是下节描述的“数据锁”。

5.3.3. 数据锁

```
1  struct  hash_table
2  {
3     long  nbuckets;
```

```
4     struct bucket **buckets;
5  };
6
7  struct bucket {
8     spinlock_t bucket_lock;
9     node_t *list_head;
10 };
11
12 typedef struct node {
13    unsigned long key;
14    struct node *next;
15 } node_t;
16
17 int hash_search(struct hash_table *h, long key)
18 {
19    struct bucket *bp;
20    struct node *cur;
21    int retval;
22
23    bp = h->buckets[key % h->nbuckets];
24    spin_lock(&bp->bucket_lock);
25    cur = bp->list_head;
26    while (cur != NULL) {
27       if (cur->key >= key) {
28          retval = (cur->key == key);
29          spin_unlock(&bp->hash_lock);
30          return retval;
31       }
32       cur = cur->next;
33    }
34    spin_unlock(&bp->hash_lock);
35    return 0;
36 }
```

图 5.18：基于数据锁的哈希表搜索算法

许多数据结构都可以分割，数据结构的每个部分带有一把自己的锁。这样虽

然每个部分一次只能执行一个临界区，但是数据结构的各个部分形成的临界区就可以并行执行了。在锁竞争必须降低时，和同步开销不是主要局限时，可以使用数据锁。数据锁通过将一块过大的临界区分散到各个小的临界区来减少锁竞争，比如，维护哈希表中的 per-hash-bucket 临界区，如图 5.18 所示。不过这种扩展性的增强带来的是复杂性的提高，增加了额外的数据结构 struct bucket。

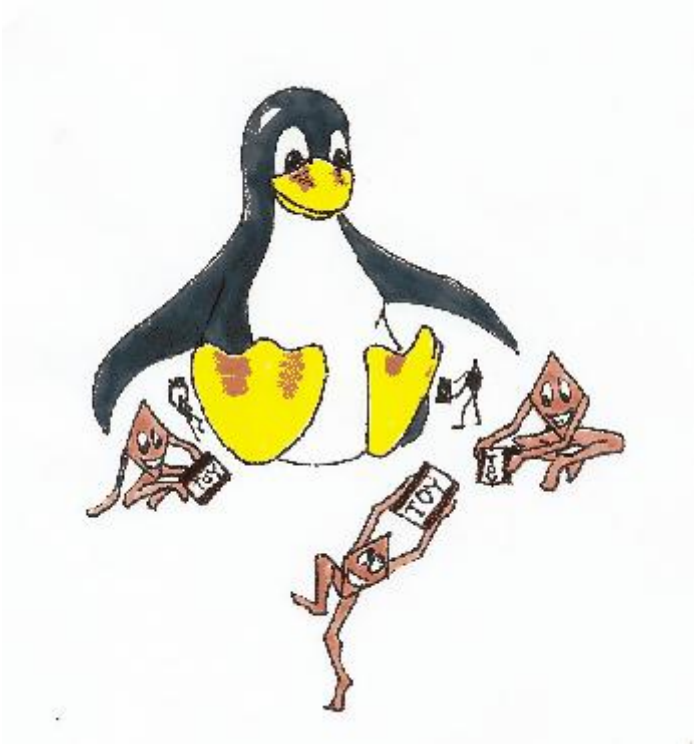


图 5.19：数据锁

和图 5.17 中所示的紧张局面不同，数据锁带来了和谐，见图 5.19——在并行程序中，这总是意味着性能和可扩展性的提升。基于这种原因，Sequent 在它的 DYNIX 和 DYNIX/ptx 操作系统中大量使用了数据锁[BK85][Inm85][Gar90][Dov90][MD92][MG92][MS93]。

不过，那些照顾过小孩子的人可以证明，再细心的照料也不能保证一切风平浪静。同样的情况也适用于 SMP 程序。比如，Linux 内核维护了一种文件和目录的缓存（叫做“dcache”）。该缓存中的每个条目都有一把自己的锁，但是对应根目录的条目和它的直接后代相较于其他条目更容易被遍历到。这将导致许多 CPU 竞争这些热门条目的锁，就像图 5.20 中所示的情景。

图 5.20：数据锁出现问题

在许多情况下，可以设计算法来减少数据冲突的次数，某些情况下甚至可以完全消灭冲突（像 Linux 内核中的 dcache 一样[MSS04]）。数据锁通常用于分割像哈希表一样的数据结构，也适用于每个条目用某个数据结构的实例表示这种情况。2.6.17 内核的 task list 就是后者的例子，每个任务结构都有一把自己的 proc_lock 锁。

在动态分配结构中，数据锁的关键挑战是如何保证在已经获取锁的情况下结构本身是否存在。图 5.18 中的代码通过将锁放入静态分配并且永不释放的哈希桶，解决了上述挑战。但是，这种手法不适用于哈希表大小可变的情况，所以锁也需要动态分配。在这种情况，还需要一些手段来阻止哈希桶在锁被获取后这段时间内释放。

小问题 5.11：当结构的锁被获取时，如何防止结构被释放呢？

5.3.4. 数据所有权

数据所有权方法按照线程或者 CPU 的个数分割数据结构，这样每个线程/CPU 都可以在不需任何同步开销的情况下访问属于它的子集。但是如果线程 A 希望访问另一个线程 B 的数据，那么线程 A 是无法直接做到这一点的。取而代之的是，线程 A 需要先与线程 B 通信，这样线程 B 以线程 A 的名义执行操作，或者，另一种方法，将数据迁移到线程 A 上来。

数据所有权看起来很神秘，但是却应用得十分频繁。

1. 任何只能被一个 CPU 或者一个线程访问的变量（C 或者 C++中的 auto

变量）都属于这个 CPU 或者这个进程。

2. 用户接口的实例拥有对应的用户上下文。这在与并行数据库引擎交互的应用程序中十分常见，这让并行引擎看起来就像顺序程序一样。这样的应用程序拥有用户接口和他当前的操作。显式的并行化只在数据库引擎内部可见。
3. 参数模拟通常授予每个线程一段特定的参数区间，以此达到某种程度的并行化。

如果共享比较多，线程或者 CPU 见的通信会带来较大的复杂性和通信开销。不仅如此，如果使用的最多的数据正好被一个 CPU 拥有，那么这个 CPU 就成为了“热点”，有时就会导致图 5.20 中的类似情况。不过，在不需要共享的情况下，数据所有权可以达到理想性能，代码也可以像图 5.15 中所示的顺序程序例子一样简单。最坏情况通常被称为“尴尬的并行化”，而最好情况，则像图 5.19 中所示一样。

另一个数据所有权的重要实例发生在数据是只读时，这种情况下，所有线程可以通过复制来“拥有”数据。

5.3.5. 锁粒度与性能

本节以一种数学上的同步效率的视角，将视线投向锁粒度和性能。对数学不敢兴趣的读者可以跳过本节。

本节的方法是用一种关于同步机制效率的粗略队列模型，该机制只操作一个共享的全局变量，基于 M/M/1 队列。M/M/1 队列。

（本节未翻译）

...

5.4. 并行快速路径

细粒度的设计一般要比粗粒度的设计复杂。在许多情况，绝大部分开销只由一小部分代码产生[Knu73]。所以为什么不把精力放在这一小块代码上。

这就是并行快速路径设计模式背后的想法，尽可能地并行化常见情况下的代码路径，同时不产生并行化整个算法所带来的复杂性。您必须理解这一点，不只算法需要并行化，算法所属的工作负载也要并行化。构建这种并行快速路径，需要极大的创造性和设计上的努力。

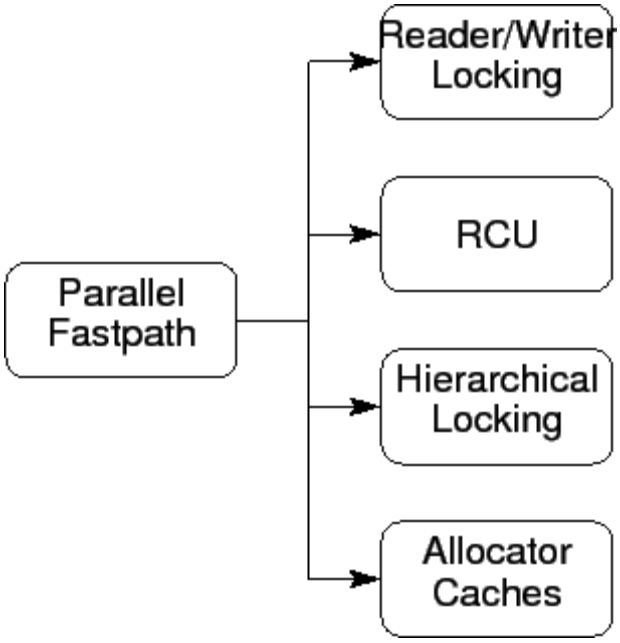


图 5.23：并行快速路径的设计模式

并行快速路径结合了两种以上的设计模式（快速路径和其他的），因此也成为了一种模板设计模式。下列并行快速路径的实例结合了其他设计模式，见图 5.23。

1. 读写锁（5.4.1 节将描述）。
2. Read-Copy-Update（RCU），多数作为读写锁的高性能代替者使用，将在第 8.3 节介绍，本章中不会对其进行深入讨论。
3. 层级锁（[Mck96]），第 5.4.2 节将进行介绍。
4. 资源分配器缓存（[McK96][MS93]）。第 5.4.3 节将进行介绍。

5.4.1. 读写锁

```
1  rwlock_t  hash_lock;
2
3  struct  hash_table
4  {
5     long  nbuckets;
6     struct  node  **buckets;
7  };
8
9  typedef  struct  node  {
10     unsigned  long  key;
11     struct  node  *next;
```

```
12  }  node_t;
13
14  int  hash_search(struct  hash_table  *h,  long  key)
15  {
16     struct  node  *cur;
17     int  retval;
18
19     read_lock(&hash_lock);
20     cur  =  h->buckets[key  %  h->nbuckets];
21     while  (cur  !=  NULL)  {
22        if  (cur->key  >=  key)  {
23           retval  =  (cur->key  ==  key);
24           read_unlock(&hash_lock);
25           return  retval;
26        }
27        cur  =  cur->next;
28     }
29     read_unlock(&hash_lock);
30     return  0;
31  }
```

图5.24：基于读写锁的哈希表搜索算法

如果同步开销可以忽略不计（比如程序使用了粗粒度的并行化），并且只有一小段临界区修改数据，那么让多个读者并行处理可以显著地增加可扩展性。写者与读者互斥，写者和另一写者也互斥。图 5.24 显示了哈希搜索如何用读写锁实现。

读写锁是非对称锁的一种简单实例。Snaman[ST87]描述了一种叹为观止的非对称锁，带有 6 个模式，在许多集群系统中使用。第 6 章进一步介绍了普通锁和读写锁的许多详细信息。

5.4.2. 层级锁

```
1  struct  hash_table
2  {
3     long  nbuckets;
4     struct  bucket  **buckets;
5  };
```

```
6
7  struct  bucket  {
8     spinlock_t  bucket_lock;
9     node_t  *list_head;
10 };
11
12 typedef  struct  node  {
13    spinlock_t  node_lock;
14    unsigned  long  key;
15    struct  node  *next;
16 }  node_t;
17
18 int  hash_search(struct  hash_table  *h,  long  key)
19 {
20    struct  bucket  *bp;
21    struct  node  *cur;
22    int  retval;
23
24    bp  =  h->buckets[key  %  h->nbuckets];
25    spin_lock(&bp->bucket_lock);
26    cur  =  bp->list_head;
27    while  (cur  !=  NULL)  {
28       if  (cur->key  >=  key)  {
29          spin_lock(&cur->node_lock);
30          spin_unlock(&bp->bucket_lock);
31          retval  =  (cur->key  ==  key);
32          spin_unlock(&cur->node_lock);
33          return  retval;
34       }
35       cur  =  cur->next;
36    }
37    spin_unlock(&bp->bucket_lock);
38    return  0;
39 }
```

图5.25：基于层级锁的哈希表搜索算法

层级锁背后的想法是尽可能长时间的持有一把粗粒度的锁，在此期间持有细粒度的锁。图 5.25 显示了我们的哈希表搜索如何采用层级锁的方式实现，不过这也显示了该方法的重大弱点：我们付出了获取第二把锁的开销，可以我们只持有它一小段时间。在这种情况下，简单的数据锁方法则更简单且性能又佳。

小问题 5.14： 哪种情况下使用层级锁最好？

5.4.3. 资源分配器缓存

本节展示了一种简明扼要的并行内存分配器，用于分配固定大小的内存。更多信息请见[MG92][MS93][BA01][MSK01]等书，或者参见 Linux 内核[Tor03]。

5.4.3.1. 并行资源分配问题

并行内存分配器锁面临的基本问题是在普通情况下快速地进行内存分配和释放，和在不理想情况下有效的分布内存之间的矛盾。

假设有一个直接了当的数据所有权类型的程序——该程序简单地将内存按照 CPU 数划分，这样每个 CPU 都有自己的一份。比如，该系统有 2 个 CPU 和 2G 内存（和我正在敲字的这台电脑一样）。我们可以为每个 CPU 分配 1G 内存，这样每个 CPU 都可以访问属于自己的那一半内存，无须加锁，以及不必关心锁带来的复杂性和开销。不幸的是，这种简单的模型存在问题，如果有一种算法，让 CPU0 分配所有内存，让 CPU1 释放所有内存，在在生产者-消费者算法中是可能的，这样模型就失效了。

另一个极端，代码锁，则受到大量锁竞争和通信开销的影响。

5.4.3.2. 资源分配的并行快速路径

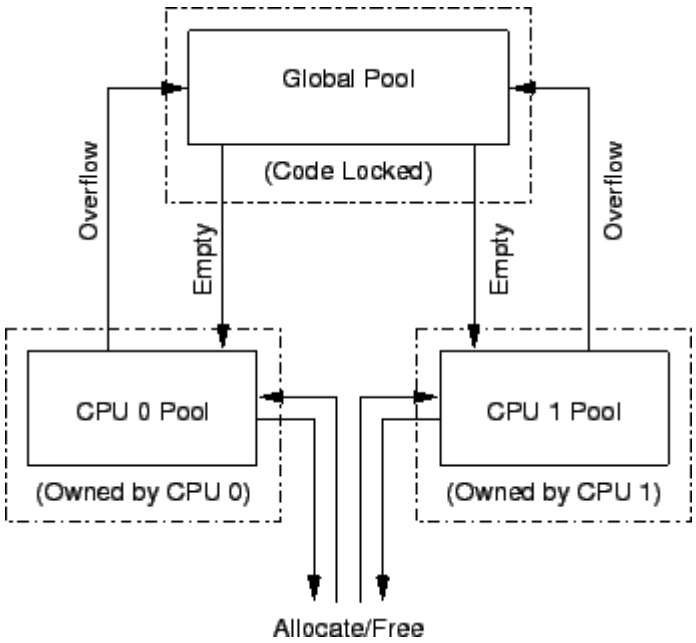


图 5.26：分配器缓存概要图

常见的解决方案让每个 CPU 拥有一块规模适中的内存块缓存，以此作为快速路径，同时提供一块较大的共享内存池分配额外的内存块，该内存池用代码锁保护。为了防止任何 CPU 独占内存块，我们给每个 CPU 的缓存可以容纳的内存块大小做一限制。在双核系统中，内存块的数据流如图 5.26 所示：当某个 CPU 的缓存池已满时，该 CPU 释放的内存块将传送到全局缓存池中，类似地，当 CPU 缓存池为空时，该 CPU 所要分配的内存块是从全局缓存池中取出来的。

5.4.3.3. 数据结构

```
1  #define  TARGET_POOL_SIZE  3
2  #define  GLOBAL_POOL_SIZE  40
3
4  struct  globalmempool  {
5     spinlock_t  mutex;
6     int  cur;
7     struct  memblock  *pool[GLOBAL_POOL_SIZE];
8  }  globalmem;
9
10  struct  percpumempool  {
```

```
11      int  cur;
12      struct  memblock  *pool[2  *  TARGET_POOL_SIZE];
13  };
14
15  DEFINE_PER_THREAD(struct  percpumempool,
percpumem);
```

图 5.27：分配器缓存数据结构

图 5.27 是一个“玩具式”缓存分配器的数据结构。图 5.26 的全局缓存池由 globalmem 实现，类型为 strcut globalmempool，两个 CPU 的缓存池是由每 CPU 变量 percpumem 实现，类型为 struct percpumempool。这两个数据结构都有一个 pool 字段，作为指向内存块的指针数组，从下标 0 开始向上填充。这样，如果 globalmem.pool[3]为 NULL，那么从下标 4 开始的数组成员都为 NULL。cur 字段包含 pool 数组中下标最大的非空元素，为-1 表示所有 pool 数组的成员为空。从 globalmem.pool[0]到 globalmem.pool[globalmem.cur]的所有成员必须非空，而剩下的成员必须为空。

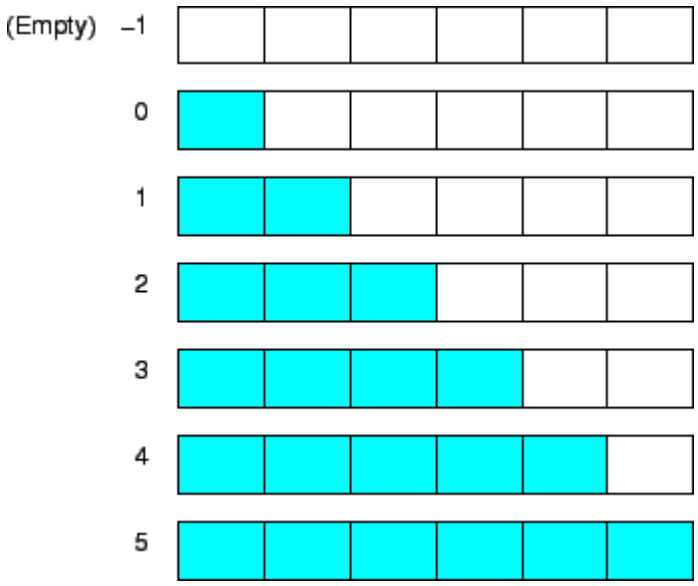


图 5.28：分配器缓冲池概要图

对 pool 数据结构的操作见图 5.28，图中的六个格子代表 pool 字段中的数据指针，他们前面的数字代表 cur 字段。深色的格子代表非空的指针，浅色的格子代表 NULL 指针。虽然有些让人迷惑，但是请注意该数据结构有一个重要的性质：cur 字段总是比非空指针的个数少一个。

5.4.3.4. 分配函数

```
1  struct  memblock  *memblock_alloc(void)
```

```
2  {
3     int  i;
4     struct  memblock  *p;
5     struct  percpumempool  *pcpp;
6
7     pcpp  =  &__get_thread_var(percpumem);
8     if  (pcpp->cur  <  0)  {
9        spin_lock(&globalmem.mutex);
10        for  (i  =  0;  i  <  TARGET_POOL_SIZE  &&
11           globalmem.cur  >=  0;  i++)  {
12           pcpp->pool[i]  =
                        globalmem.pool[globalmem.cur];
13           globalmem.pool[globalmem.cur--]  =  NULL;
14        }
15        pcpp->cur  =  i  -  1;
16        spin_unlock(&globalmem.mutex);
17     }
18     if  (pcpp->cur  >=  0)  {
19        p  =  pcpp->pool[pcpp->cur];
20        pcpp->pool[pcpp->cur--]  =  NULL;
21        return  p;
22     }
23     return  NULL;
24  }
```

图5.29：分配器缓存的分配函数

在图 5.29 中可以见到分配函数 memblock_alloc()。第 7 行获得当前线程的每线程缓存池，第 8 行检查缓存池是否为空。

如果为空，第 9-16 行尝试从全局缓存池中取出内存块填满每线程缓存池，第 9 行获取自旋锁，第 16 行释放自旋锁。第 10-14 行从全局缓存池中取出内存块，移至每线程缓存池，直到每线程缓存池达到目标大小（半满）或者全局缓存池耗尽为止，第 15 行将每线程缓存池的 cur 字段设置为合适大小。

接下来，第 18 行检查每线程缓存池是否还是为空，如果不是，第 19-21 行取出一个内存块返回。如果为空，第 23 返回内存耗尽的不幸消息。

5.4.3.5. 释放函数

```
1  void  memblock_free(struct  memblock  *p)
2  {
3     int  i;
4     struct  percpumempool  *pcpp;
5
6     pcpp  =  &__get_thread_var(percpumem);
7     if  (pcpp->cur  >=  2  *  TARGET_POOL_SIZE  -  1)
      {
8        spin_lock(&globalmem.mutex);
9        for  (i  =  pcpp->cur;
               i  >=  TARGET_POOL_SIZE;  i--)
          {
10           globalmem.pool[++globalmem.cur]  =
                                        pcpp->pool[i];
11           pcpp->pool[i]  =  NULL;
12        }
13        pcpp->cur  =  i;
14        spin_unlock(&globalmem.mutex);
15    }
16    pcpp->pool[++pcpp->cur]  =  p;
17 }
```

图5.30：分配器缓存的释放函数

图 5.30 是内存块的释放函数。第 6 行获取当前线程的缓存池指针，第 7 行检查该缓存池是否已满了。

如果是，第 8-15 行将每线程缓存池一半的内存块释放给全局缓存池，第 8 行和第 14 行获取、释放自旋锁。第 9-12 行实现将内存块从本地移至全局缓存池的循环，第 13 行将每线程缓存池的 cur 字段设置为合适的大小。

接下来，第 16 行将刚刚释放的内存块放入每线程缓存池。

5.4.3.6. 性能

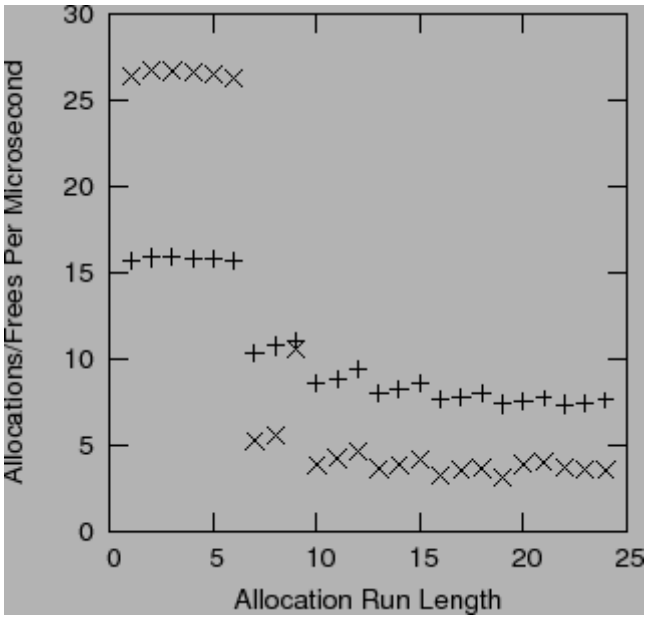


图 5.31：分配器缓存的性能

图 5.31 是粗略的性能数据，在 1Ghz（每个 CPU 4300 bogomips）的双核 Intel x86 处理器上运行，每个 CPU 的缓存至多可以装下 6 个内存块。在这个微型 benchmark 中，每个线程重复分配一组内存块，然后释放，一组内存块的数目以“分配运行长度”的名字显示在 x 轴上。y 轴是每毫秒成功分配/释放的数目——失败的分配不计入内，“+”来自于单线程的运行结果。

从图中可以发现，运行长度在 6 之前拥有线性的扩展性，性能也非常优秀，不过当运行长度大于 6 以后，性能开始变得低下，扩展性甚至为负值。鉴于此，将 TARGET_POOL_SIZE 设置的足够大非常重要，幸运的是在实践中很容易做到这点[MSK01]，尤其是内存容量疯涨的今天。比如，在大多数系统中，将 TARGET_POOL_SIZE 设置为 100 是很合理的，这样可以保证有 99%的时间是使用每线程缓存池分配和释放的。

从图中可知，使用数据所有权思想的例子相较于使用锁的版本，极大地提升了性能。在常规情况中避免使用锁，正是本书不断重复的主题。

小问题 5.15：图 5.31 中存在一个性能提升的模式：增加三个样本中的运行长度，比如 10，11 和 12，为什么不这么做？

小问题 5.16：当运行长度为 19 或更大时，两线程测试开始出现分配失败。如果全局缓存池的大小为 40，每 CPU 缓存池的大小为 3，那么可以出现分配失败的最小分配运行长度是多少？

5.4.3.7. 真实世界中的设计

并行的“玩具”资源分配器非常简单，但是真实世界中的设计在几个方面扩展这个方案。

首先，真实的资源分配器需要处理各种不同的资源大小，在“玩具”中只能分配固定大小。一种比较流行的做法是提供一系列固定大小的资源，恰当地放置以平衡内碎片和外碎片，比如 1980 年代晚期的的 BSD 内存分配器[MK88]。这样做就意味着每种资源大小都要有一个“globalmem”变量，同样对应的锁也要每种一个，因此真实的实现将采用数据锁，而非“玩具”程序中的代码锁。

其次，产品级的系统必须可以改变内存的用途，这意味着这些系统必须能将内存块组合成更大的数据结构，比如页[MS93]。这种组合也需要锁的保护，这种锁也必须是一种资源大小一个的。

第三，组合后的内存必须回到内存管理系统，内存页也必须是从内存管理系统分配的。这一层面所需要的锁将依赖于内存管理系统，但也可以是代码锁。在这一层面中使用代码锁通常是可以容忍的，因为在设计良好的系统中很少触及这一级别[MSK01]。

尽管真实世界的设计要复杂许多，但背后的思想是一样的——并行的快速路径。

表 5.1：真实世界中的并行分配器

等级	锁类型	目的
每线程资源池	数据所有权	高速分配
全局内存块资源池	数据锁	将内存块放在各个线程中
组合	数据锁	将内存块放在页中
系统内存	代码锁	获取、释放系统内存

5.5. 性能总结

@@@ summarize performance of the various options. Forward-reference to the RCU/NBS section.

6. 锁

在过去几十年并发研究领域的出版物中，锁总是扮演着坏人的角色，锁背负的指控包括引起死锁、锁封护（luyang 注：lock convoying，多个同优先级的线程重复竞争同一把锁，此时大量虽然被唤醒而得不到锁的线程被迫进行调度切换，这种频繁的调度切换相当影响系统性能）、饥饿、不公平、data races 以及其他许多并发带来的罪孽。有趣的是，在共享内存并行软件中真正承担重担的是——你猜对了——锁。

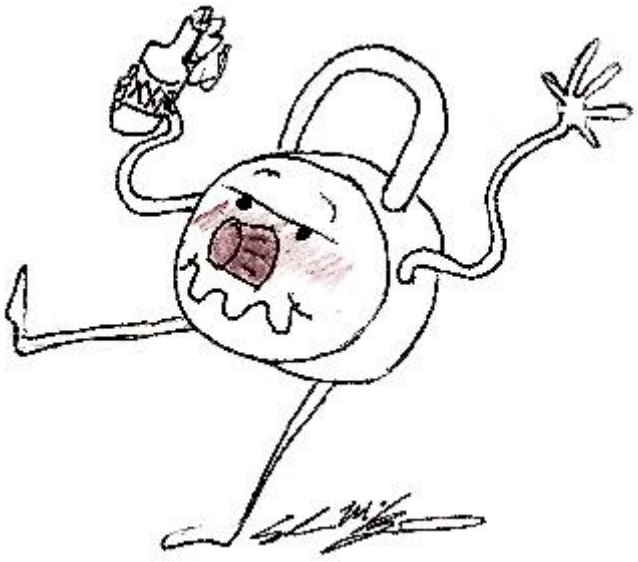


图 6.1：锁：坏人还是懒汉？

这种截然不同的看法源于下面几个原因：

1. 很多因锁产生的问题大都有实用的解决办法，可以在大多数场合工作良好，比如
   a) 使用锁层级避免死锁。
   b) 死锁检测工具，比如 Linux 内核 lockdep 功能[Cor06]。
   c) 对锁友好的数据结构，比如数组、哈希表、基数，第十一章将会讲述这些数据结构。
2. 有些锁的问题只有在竞争程度很高时才会出现，一般只有不良的设计才会让锁竞争如此激烈。
3. 有些锁的问题可以通过其他同步机制配合锁来避免。包括引用计数、统计计数、不阻塞的简单数据结构，还有 RCU。

4. 在不久之前，几乎所有的共享内存并行程序都是闭源的，所以多数研究者很难知道业界的实践解决方案。
5. 所有美好的故事都需要一个坏人，锁在研究文献中扮演坏小子的角色已经有着悠久而光荣的历史了。

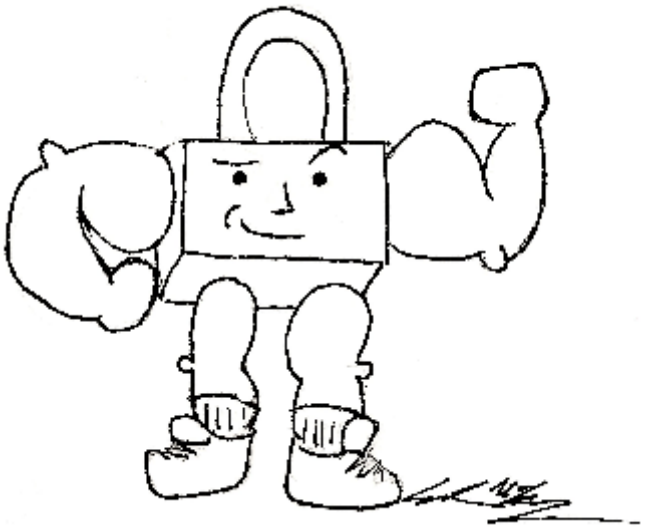


图 6.2：锁：驮马还是英雄？

本章将给读者一个概述的认识，了解如何避免锁所带来的问题。

6.1. 生存（staying alive）

由于锁被控诉造成死锁和饥饿，因此对于共享内存并行程序的开发者来说，最重要的考虑之一是保持程序的运行。下面的章节将涵盖死锁、活锁、饥饿、不公平和低效率。

6.1.1. 死锁

死锁发生在一组线程持有至少一把锁时又等待该组线程中某个成员释放它持有的另一把锁时。

如果缺乏外界干预，死锁会一直持续。除了持有锁的线程释放，没有线程可以获取到该锁，但是持有锁的线程在等待获取该锁的线程释放其他锁之前，又无法释放该锁。

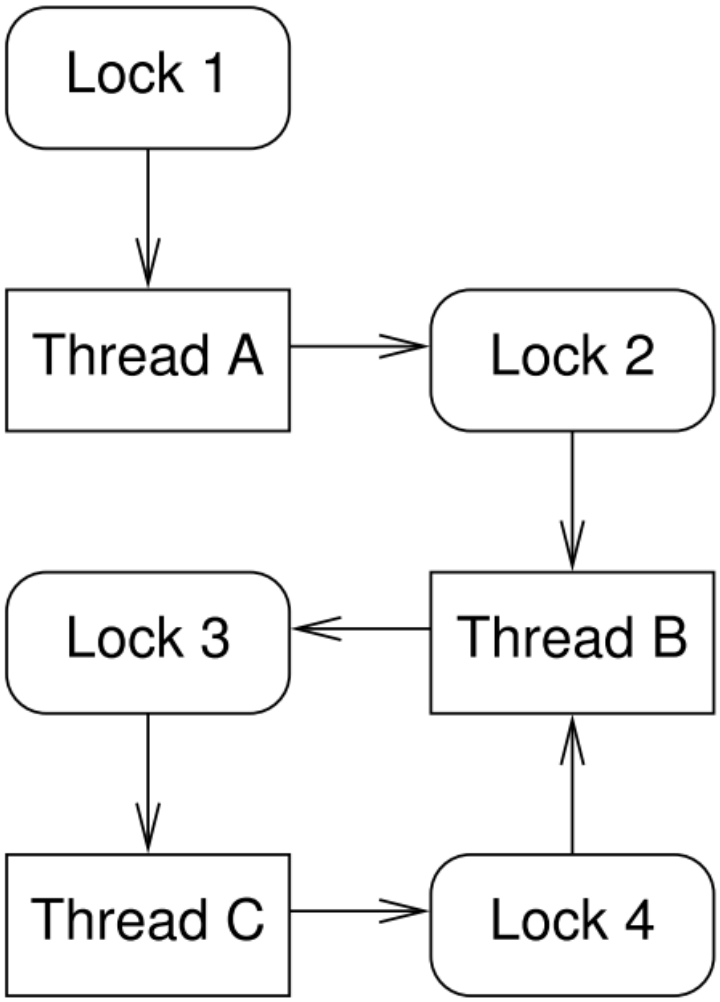


图 6.3：死锁循环

我们可以用有向图来表示死锁，节点代表锁和线程，如图 6.3 所示。从锁指向线程的箭头表示线程持有了该锁，比如，线程 B 持有锁 2 和锁 4。从线程到锁的箭头表示线程在等待这把锁，比如，线程 B 等待锁 3 释放。

死锁场景必然包含至少一个以上的死锁循环。在图 6.3 中，死锁循环是线程 B、锁 3、线程 C、锁 4，然后又回到线程 B。

小问题 6.1： 但是死锁的定义只说了每个线程持有至少一把锁，然后等待同样的线程释放持有的另一把锁。你怎么知道这里有一个循环？

虽然有一些软件环境，比如数据库系统，可以修复已有的死锁，但是这种方法要么需要杀掉其中一个线程，要么强制从某个线程中偷走一把锁。杀线程和强制偷锁对于事务交易是可以的，但是对内核和应用程序这种层次的锁来说问题多多。

因此，内核和应用程序需要避免死锁。避免死锁主要有三种方法，等级锁，条件锁，一次一把锁的设计。

等级锁为锁编号，禁止乱序获取锁。在图 6.3 中我们可以用数字为锁编号，

这样如果线程已经获取来编号相同或者更高的锁，就不允许再获取这把锁。线程B侵犯了锁的等级，因为它在持有锁4时又试图获取锁3，因此导致死锁的发生。

再一次强调，使用等级锁要为锁编号，禁止乱序获取锁。在大型程序中，最好用工具来检查锁的等级[Cor06]。

```
1  spin_lock(&lock2);
2  layer_2_processing(pkt);
3  nextlayer  =  layer_1(pkt);
4  spin_lock(&nextlayer->lock1);
5  layer_1_processing(pkt);
6  spin_unlock(&lock2);
7  spin_unlock(&nextlayer->lock1);
```

图 6.4：协议分层与死锁

但是假设锁没有合理的等级。这在真实世界完全会发生，比如，在分层的网络协议栈中，报文流是双向的。当报文从一层发送到另一层时，锁完全有可能是被其他层持有的。考虑到报文可以在协议栈中上下传送，图 6.4 有一个死锁的秘诀。当报文在协议栈中从上往下发送时，必须逆序获取下一层的锁。而报文在协议栈中从下往上发送时，是按顺序获取锁，图 6.4 中第 4 行的获取锁操作将导致死锁。

```
1  retry:
2     spin_lock(&lock2);
3     layer_2_processing(pkt);
4     nextlayer  =  layer_1(pkt);
5     if  (!spin_trylock(&nextlayer->lock1))  {
6        spin_unlock(&lock2);
7        spin_lock(&nextlayer->lock1);
8        spin_lock((&lock2);
9        if  (layer_1(pkt)  !=  nextlayer)  {
10           spin_unlock(&nextlayer->lock1);
11           spin_unlock((&lock2);
12           goto  retry;
13        }
14     }
15     layer_1_processing(pkt);
16  spin_unlock(&lock2);
17  spin_unlock(&nextlayer->lock1);
```

图 6.5：通过条件锁来避免死锁

在本例中避免死锁的办法是强加一套锁的等级，但是在有需要时又可以有条件地乱序获取锁，如图 6.5 所示。与图 6.4 中无条件的获取层 1 的锁不同，第 5 行用 spin_trylock()有条件地获取锁。该原语在锁可用时立即获取锁，在锁不可用时不获取锁，返回 0。

如果 spin_trylock()成功，第 15 行进行层 1 的处理工作。否则，第 6 行释放锁，第 7 行和第 8 行用正确的顺序获取锁。不幸的是，系统中可能有多块网络设备（比如 Ethernet 和 WiFi），这样 layer_1()函数必须进行路由选择。这种选择随时都可能改变，特别是系统可移动时。所以第 9 行必须重新检查路由选择，如果发生改变，必须释放锁重新来过。

小问题 6.2： 图 6.4 到 6.5 的转变能否推广到其他场景？

小问题 6.3：为了避免死锁，图 6.5 带来的复杂性也是值得的吧？

在某些情况下，避免锁嵌套是可以的，这样也就避免了死锁。但是，肯定还有一些手段，可以 ensure that the needed data structures remain in existence during the time that neither lock is held。第 6.3 节讨论了其中一种手段，剩下的在第八章叙述。

6.1.2. 活锁

虽然条件锁是一种有效的避免死锁机制，但是有可能被滥用。看看图 6.6 中优美的对称例子吧。该例中的美遮掩了一个丑陋的活锁。为了发现它，考虑以下事件顺序：

1. 第 4 行线程 1 获取锁 1，然后调用 do_one_thing()。
2. 第 18 行线程 2 获取锁 2，然后调用 do_a_third_thing()。
3. 线程 1 试图获取锁 2，由于线程 2 已经持有而失败。
4. 线程 2 试图获取锁 1，由于线程 1 已经持有而失败。
5. 线程 1 释放锁 1，然后跳转到 retry。
6. 线程 2 释放锁 2，然后跳转到 retry。
7. 上述过程不断重复，活锁华丽登场。

```
1  void  thread1(void)
2  {
3  retry:
4     spin_lock(&lock1);
5     do_one_thing();
6     if  (!spin_trylock(&lock2))  {
7        spin_unlock(&lock1);
```

```
8         goto  retry;
9     }
10     do_another_thing();
11     spin_unlock(&lock2);
12     spin_unlock(&lock1);
13  }
14
15  void  thread2(void)
16  {
17  retry:
18     spin_lock(&lock2);
19     do_a_third_thing();
20     if  (!spin_trylock(&lock1))  {
21        spin_unlock(&lock2);
22        goto  retry;
23     }
24     do_a_fourth_thing();
25     spin_unlock(&lock1);
26     spin_unlock(&lock2);
27  }
```

图 6.6：滥用条件锁

小问题 6.4： 图 6.6 中的活锁该如何避免？

饥饿与活锁十分类似。换句话说，活锁是饥饿的一种极端形式，所有线程都饥饿了。

6.1.3. 不公平

N/A

6.1.4. 低效率

N/A

6.2. 锁的类型

锁有多少种类型，说出来一定让你大吃一惊，远超过本小节能描述的范围之

外。下列章节讨论了互斥锁（第 6.2.1 节）、读写锁（第 6.2.2 节）和 multi-role 锁（第 6.2.3 节）。

6.2.1. 互斥锁

N/A

6.2.2. 读写锁

N/A

6.2.3. Beyond Reader-Writer Locks

VAXCluster 六态锁。

6.3. 基于锁的存在担保（existence guarantee）

并行编程中的一种关键挑战是提供存在担保（existence guarantee）[GKAS99]，这样才能正确解决这个问题：尝试删除一个对象的同时又有其他人尝试并发地访问该对象。存在担保对于数据元素被该元素中的锁或者引用计数保护这种情况极其有用。无法提供这种担保的代码容易埋藏隐含的竞态，如图 6.7 所示。

```
1  int  delete(int  key)
2  {
3     int  b;
4     struct  element  *p;
5
6     b  =  hashfunction(key);
7     p  =  hashtable[b];
8     if  (p  ==  NULL  ||  p->key  !=  key)
9        return  0;
10     spin_lock(&p->lock);
11     hashtable[b]  =  NULL;
12     spin_unlock(&p->lock);
13     kfree(p);
14     return  1;
15  }
```

图 6.7：没有存在担保的每数据元素锁

小问题 6.5： 如果图 6.7 第 8 行中我们想删除的元素不是列表中第一个元素怎么办？

小问题 6.6： 图 6.7 会出现什么竞态条件？

```
1  int  delete(int  key)
2  {
3     int  b;
4     struct  element  *p;
5     spinlock_t  *sp;
6
7     b  =  hashfunction(key);
8     sp  =  &locktable[b];
9     spin_lock(sp);
10     p  =  hashtable[b];
11     if  (p  ==  NULL  ||  p->key  !=  key)  {
12        spin_unlock(sp);
13        return  0;
14     }
15     hashtable[b]  =  NULL;
16     spin_unlock(sp);
17     kfree(p);
18     return  1;
19  }
```

图 6.8：带有基于锁的存在担保的每数据元素锁

解决本例问题的办法是使用一个全局锁的哈希集合，这样每个哈希桶都有自己的锁，如图 6.8 所示。该方法允许在获取指向数据元素的指针（第 10 行）之前获取合适的锁（第 9 行）。虽然该方法在数据包含在单个可分割的数据结构比如图中所示的哈希表时工作的不错，但是如果有某个数据元素可以是多个哈希表的成员，或者更复杂的数据结构比如树和图时，就会有问题了。这些问题还是可以解决的，事实上，这些解决办法组成来基于锁的软件事务性内存实现[ST95][DSS06]。第八章将描述提供存在担保的简单方法。

7. 数据所有者

每 CPU/任务/进程/线程数据。

Function shipping vs. data shipping。

大问题：本地处理有多少 vs 全局处理有多少？多频繁，代价多高，... 最好是集中还是分化？

map/reduce 的关系？消息传递！

这些问题显示来耦合数据所有权相较其他方法的好处。比如，work-stealing 调度器。也许内存分配也可以算进来，虽然它现在的地位还不错。

8. 延迟处理

延后工作的策略可能在人类出现之前就存在了，但直到最近几十年，人们才认识到该策略在简化并行算法的价值[KL80]，[Mas92]。通用的并行编程延后工作方法包括排队、引用计数和 RCU。

8.1. 屏障

HPC 风格的屏障。

8.2. 引用计数

引用计数跟踪一个对象被引用的次数，防止对象过早的被释放。虽然这是一种概念上很简单的技术，但是在细节中隐藏着许多魔鬼。毕竟，如果对象不太会提前释放，那么就不需要引用计数了。但是如果对象容易被提前释放，那么如何阻止对象在获取引用计数过程中被提前释放？

对这个问题有几个可能的答案，包括：

1. 在操作引用计数时必须持有一把处于对象之外的锁。可供选择的锁类型很多，大多数都可以满足要求。
2. 使用不为 0 的引用计数创建对象，只有在当前引用计数不为 0 时才能获取新的引用计数。一旦获取，引用可以传递给其他实体。
3. 为对象提供存在担保，这样在任何有实体尝试获取引用的时刻都无法释放对象。存在担保通常是由自动垃圾收集器来提供，并且您在第 8.3 节还会看到，RCU 也会提供存在担保。
4. 为对象提供类型安全的存在担保，当获取到引用时将会执行附加的类型检查。类型安全的存在担保可以由专用内存分配器提供，也可以由 Linux 内核中的 SLAB_DESTROY_BY_RCU 特性提供，如第 8.3 节所示。

当然，任何提供存在担保的机制，根据其定义实际也提供类型安全的保证。所以本节将后两种答案合并放在 RCU 一类，这样我们就有三种保护引用获取的类型了，即锁、引用计数和 RCU。

小问题 8.1：为什么不用简单的比较并交换操作来实现引用获取呢？这样可以只在引用计数不为 0 时获取引用。

考虑到引用计数问题的关键是对引用获取和释放对象之间的同步，我们有九种可能的机制组合，如表 8.1 所示。表中将引用计数机制归为以下几个大类：

表 8.1：引用计数和同步机制

获取同步	释放同步		
	锁	引用计数	RCU
锁	-	CAM	CA
引用计数	A	AM	A
RCU	CA	MCA	CA

1. 简单计数，不使用原子操作，内存屏障或者对齐限制（“-”）。
2. 不使用内存屏障的原子计数（“A”）。
3. 原子计数，只在释放时使用内存屏障（“AM”）。
4. 原子计数，在获取时用原子操作检查，在释放时使用内存屏障（“CAM”）。
5. 原子计数，在获取时用原子操作检查（“CA”）。
6. 原子计数，在获取时用原子操作检查，在获取时还使用内存屏障（“MCA”）。

但是，由于 Linux 内核中所有的返回值的原子操作都包含内存屏障，所有释放操作也包含内存屏障。因此，类型“CA”和“MCA”与“CAM”相等，这样就只剩前四种类型：“-”，“A”，“AM”，“CAM”。第 8.2.2 节列出了支持引用计数的 Linux 原语。稍后的章节将给出一种优化，可以改进引用获取和释放十分频繁，而很少需要检查引用是否为 0 这一情况下的性能。

8.2.1. 引用计数类型的实现

第 8.2.1.1 节描述了由锁保护的简单计数（“-”），第 8.2.1.2 节描述了不带内存屏障的原子计数（“A”），第 8.2.1.3 节描述了获取时使用内存屏障的原子计数（“AM”），第 8.2.1.4 节描述了检查和释放时使用内存屏障的原子计数（“CAM”）。

8.2.1.1. 简单计数

```
1  struct  sref  {
2     int  refcount;
3  };
4
5  void  sref_init(struct  sref  *sref)
6  {
```

```
7     sref->refcount  =  1;
8  }
9
10  void  sref_get(struct  sref  *sref)
11  {
12     sref->refcount++;
13  }
14
15  int  sref_put(struct  sref  *sref,
16 void  (*release)(struct  sref  *sref))
17  {
18     WARN_ON(release  ==  NULL);
19     WARN_ON(release  ==  (void  (*)(struct  sref
*))kfree);
20
21     if  (--sref->refcount  ==  0)  {
22        release(sref);
23        return  1;
24     }
25     return  0;
26  }
```

图 8.1：简单引用计数的 API

简单计数，既不用原子操作也不用内存屏障，可以用于在获取和释放引用计数时都用同一把锁保护的情况。在这种情况下，引用计数是可以以非原子操作方式读写的，因为锁提供了必要的互斥保护、内存屏障、原子指令和阻止编译器优化。这种方法适用于锁在保护引用计数之外还保护其他操作的情况，这样也使得引用一个对象必须得等锁（被其他地方）释放后再持有。图 8.1 展示了简单的 API，用来实现简单非原子引用计数——虽然简单引用计数几乎都是 open-coded。

8.2.1.2. 原子计数

```
1  struct  kref  {
2     atomic_t  refcount;
3  };
4
```

```
5  void  kref_init(struct  kref  *kref)
6  {
7     atomic_set(&kref->refcount,1);
8  }
9
10  void  kref_get(struct  kref  *kref)
11  {
12     WARN_ON(!atomic_read(&kref->refcount));
13     atomic_inc(&kref->refcount);
14  }
15
16  int  kref_put(struct  kref  *kref,
17 void  (*release)(struct  kref  *kref))
18  {
19     WARN_ON(release  ==  NULL);
20     WARN_ON(release  ==  (void  (*)(struct  kref
*))kfree);
21
22     if  ((atomic_read(&kref->refcount)  ==  1)  ||
23 (atomic_dec_and_test(&kref->refcount)))  {
24         release(kref);
25         return  1;
26     }
27     return  0;
28  }
```

图 8.2：Linux 内核的 kref API

简单原子计数用于这种情况，任何 CPU 必须先持有一个引用才能获取引用。这是用在当单个 CPU 创建一个对象供自己使用时，同时也允许其他 CPU、任务、定时器处理函数或者 CPU 后来产生的 I/O completion 处理函数访问该对象。CPU 在将对象传递给其他实体手上前，必须先以该实体的名义获取一个新的引用。在 Linux 内核中，kref 原语就是用于这种引用计数的，如图 8.2 所示。

因为锁无法保护所有引用计数操作，所以需要原子计数，这意味着可能会有两种不同的 CPU 并发地操纵引用计数。如果使用普通的增/减函数，一对 CPU 可以同时获取引用计数，假设它们都获取到了计数值“3”。如果它们都增加各自的值，那么都就得到计数值“4”，然后将值写回引用计数中。但是引用计数的新

值本该是 5，这样就丢失了其中一次增加。因此，计数器增加和计数器减少时都必须使用原子操作。

如果释放引用计数由锁或者 RCU 保护，那么就不再需要内存屏障了（以及禁用编译器优化），并且锁也可以防止一对释放操作同时执行。如果是 RCU，清理必须延后直到所有当前 RCU 读端的临界区执行完毕，RCU 基础设施会提供所有需要的内存屏障和禁止编译优化。因此，如果两个 CPU 同时释放了最后两个引用，实际的清理工作将延后到所有 CPU 退出它们 RCU 读端的临界区后才会开始。

小问题 8.2： 为什么这种情况不需要保护呢：一个 CPU 释放了最后一个引用后，另一个 CPU 获取对象的引用？

kref 结构自身包括一个原子变量，如图 8.2 中的第 1、2 行所示。第 5 到第 8 行的 kref_init()函数将计数器初始化为 1。这里要注意，atomic_set()原语只是一个简单的赋值操作，它的名字来源于操作的数据类型 atomic_t，而不是指原子操作。kref_init()函数必须在对象创建过程中调用，调用点必须在该对象对其他 CPU 可见之前。

第 10-14 行的 kref_get()函数无条件地原子增加计数器的值。虽然 atomic_inc()原语并不在所有平台上显式阻止编译器优化，但是由于 kref 原语是在单独的模块中，并且 Linux 内核的编译过程不做任何跨模块的优化，因此最后达到的效果是一样的。

第 16-28 行的 kref_put()函数，第 22 行检查计数器的值是否为 1（此时不允许并发的 kref_get()），或者第 23 行原子地减少计数器的值后该值为 0。在两种情况下，kref_put()调用了指定的 release 函数，并返回 1，告知调用者已经执行了清理操作。否则，kref_put()返回 0。

小问题 8.3：如果图 8.2 中第 22 行的检查结果为假，那么怎样才能让第 23 行的检查结果为真呢？

小问题 8.4：图 8.2 中第 22 行的检查计数器值是否为 1 的非原子操作安全吗？

8.2.1.3. 带释放内存屏障的原子计数

Linux 内核的网络层采用了这种风格的引用，用于在报文路由跟踪目的缓存。实际的实现要更复杂一点儿；本节将关注 struct dst_entry 引用计数是如何满足这种用例的，如图 8.3 所示。

```
1 static inline
2 struct dst_entry * dst_clone(struct dst_entry *
dst)
```

```
3  {
4     if  (dst)
5        atomic_inc(&dst->__refcnt);
6     return  dst;
7  }
8
9  static  inline
10  void  dst_release(struct  dst_entry  *  dst)
11  {
12     if  (dst)  {
13        WARN_ON(atomic_read(&dst->__refcnt)  <  1);
14        smp_mb__before_atomic_dec();
15        atomic_dec(&dst->__refcnt);
16     }
17  }
```

图 8.3：Linux 内核的 dst_clone API

如果调用者已经持有一个 dst_entry 的引用，那么可以使用 dst_clone()原语，该原语会获取另一个引用，然后传递给内核中的其他实体。因为调用者已经持有了一个引用，dst_clone()不需要再执行任何内存屏障。将 dst_entry 传递给其他实体的行为是否需要内存屏障，要视情况而定，不过如果需要内存屏障，那么内存屏障已经嵌入在传递 dst_entry 的过程中了。

dst_release()原语可以在任何环境中调用，调用者可能在调用 dst_release()的上一条语句获取 dst_entry 结构的元素的引用。因此在第 14 行上，dst_release()原语包含了一个内存屏障，阻止编译器和 CPU 的乱序执行。

请注意，调用 dst_clone()和 dst_release()的程序员不需要关心内存屏障，只需要了解使用这两个原语的规则就够了。

8.2.1.4. 带检查和释放内存屏障的原子计数

引用计数的获取和释放可以并发执行这一事实增加了引用计数的复杂性。假设某次引用计数的释放操作发现引用计数的新值为 0，这表明它现在可以安全的清除被引用的对象了。此时我们肯定不希望在清理工作进行时又发生一次引用计数的获取操作，所以获取操作必须包含一个检查当前引用值是否为 0 的检查。该检查必须是原子自增的一部分，如下所示。

小问题 8.5：为什么检查引用计数是否为 0 的操作不能是一个简答的"if-then"

语句呢，在“then”部分使用原子增加？

Linux 内核的 fget()和 fput()原语属于这种风格的引用计数。图 8.4 是经过简化后的版本。

```
1  struct  file  *fget(unsigned  int  fd)
2  {
3     struct  file  *file;
4     struct  files_struct  *files  =  current->files;
5
6     rcu_read_lock();
7     file  =  fcheck_files(files,  fd);
8     if  (file)  {
9        if  (!atomic_inc_not_zero(&file->f_count))  {
10            rcu_read_unlock();
11            return  NULL;
12        }
13    }
14    rcu_read_unlock();
15    return  file;
16 }
17
18 struct  file  *
19 fcheck_files(struct  files_struct  *files,
                unsigned  int  fd)
20 {
21    struct  file  *  file  =  NULL;
22    struct  fdtable  *fdt  =
                  rcu_dereference((files)->fdt);
23
24    if  (fd  <  fdt->max_fds)
25       file  =  rcu_dereference(fdt->fd[fd]);
26    return  file;
27 }
28
29 void  fput(struct  file  *file)
30 {
```

```
31     if  (atomic_dec_and_test(&file->f_count))
32         call_rcu(&file->f_u.fu_rcuhead,
                    file_free_rcu);
33  }
34
35  static  void  file_free_rcu(struct  rcu_head  *head)
36  {
37     struct  file  *f;
38
39     f  =  container_of(head,  struct  file,
                          f_u.fu_rcuhead);
40     kmem_cache_free(filp_cachep,  f);
41  }
```

图 8.4：Linux 内核 fget/fput API

第 4 行的 fget()取出一个指向当前进程的文件描述符表的指针，该表可能在多个进程间共享。第 6 行调用 rcu_read_lock()，进入 RCU 读端临界区。后续任何 call_rcu()原语调用的回调函数将延后到对应的 rcu_read_unlock()完成后执行（本例中的第 10 行或者第 14 行）。第 7 行根据参数 fd 指定的文件描述符，查找对应的 struct file 结构，文件描述符的内容稍后再讲。如果指定的文件描述符存在一个对应的已打开文件，那么第 9 行尝试原子地获取一个引用计数。如果第 9 行的操作失败，那么第 10-11 行退出 RCU 读写端临界区，返回失败。如果第 9 行的操作成功，那么第 14-15 行退出读写端临界区，返回一个指向 struct file 的指针。

fcheck_files()原语是 fget()的辅助函数。该函数使用 rcu_dereference()原语来安全地获取受 RCU 保护的指针，用于之后的解引用（这将在如 DEC Alpha 之类的 CPU 上产生一个内存屏障，在这种机器上数据依赖并不让内存顺序执行？？）。第 22 行使用 rcu_dereference()来获取指向任务当前的文件描述符表的指针，第 24 行检查是否指定的文件描述符在该表范围之内。如果在，那么第 25 行获取该 struct file 的指针，然后调用 rcu_dereference()原语。第 26 行返回 struct file 的指针，如果第 24 行的检查失败，那么这里返回 NULL。

fput()原语释放一个 struct file 的引用。第 31 行原子地减少引用计数，如果自减后的值为 0，那么第 32 行调用 call_rcu()原语来释放 struct file（通过 call_rcu()第二个参数指定的 file_free_rcu()函数），不过这只在当前所有执行 RCU 读端临界区的代码执行完毕才会发生。等待当前所有执行 RCU 读端临界区的代码执行完毕的时间被称为“grace period”（优雅周期）。请注意，atomic_dec_and_test()

原语中包含一个内存屏障。在本例中该屏障并非必须，因为 struct file 只有在所有 RCU 读端临界区完成后才能销毁，但在 Linux 中，根据定义所有会返回值的原子操作都需要包含内存屏障。

一旦优雅周期完毕，第 39 行 file_free_rcu()函数获取 struct file 的指针，第 40 行释放该指针。

本方法也用于 Linux 虚拟内存系统中，请见针对 page 结构的 get_page_unless_zero()和 put_page_testzero()函数，以及针对内存映射的 try_to_unuse()和 mmput()函数。

8.2.2. 支持引用计数的 Linux 原语

在上述例子中使用的 Linux 内核原语见下表：

- atomic_t：可供原子操作的 32 位类型定义。
- void atomic_dec(atomic_t *var)：不需要内存屏障或者阻止编译器优化的原子自减引用计数操作。
- int atomic_dec_and_test(atomic_t *var)：原子减少引用计数，如果结果为 0 则返回 true。需要内存屏障并且阻止编译器优化，否则可能让引用计数在原语外改变。
- void atomic_inc(atomic_t *var)：原子增加引用计数，不需要内存屏障或者禁用编译器优化。
- int atomic_inc_not_zero(atomic_t *var)：原子增加引用计数，如果结果不为 0，那么在增加后返回 true。该操作会产生内存屏障，并禁止编译器优化，否则引用会在原语外改变。
- int atomic_read(atomic_t *var)：返回引用计数的整数值。该函数不是原子操作，不需要内存屏障，也不需要禁止编译器优化。
- void atomic_set(atomic_t *var，int val)：将引用计数的值设置为 val。该函数不是原子操作，不需要内存屏障，也不需要禁止编译器优化。
- void call_rcu(struct rcu_head *head, void (*func)(struct rcu_head *head))：在当前所有执行 RCU 读端临界区完成后调用 func(head)，不过 call_rcu()原语是立即返回的。请注意，head 通常是受 RCU 保护的数据结构的一个字段，func 通常是释放该数据结构的函数。从调用 call_rcu()到调用 func 之间的时间间隔被称为“优雅周期”。任何包含一个优雅周期的时间间隔本身就是一个优雅周期。
- type *container_of(p, type, f)：给出指针 p，指向类型为 type 的数据结构中的字段 f，返回指向数据结构的指针。

- void rcu_read_lock(void)：标记一个 RCU 读端临界区的开始。
- void rcu_read_unlock(void)：标记一个 RCU 读端临界区的结束。RCU 读端临界区可以嵌套。
- void smp_mb__before_atomic_dec(void)：只有在该平台的 atomic_dec() 原语没有产生内存屏障，禁止编译器的乱序优化时才有用，执行上面的操作。
- struct rcu_head：用于 RCU 基础框架的数据结构，用来跟踪等待优雅周期的对象。通常作为受 RCU 保护的数据结构中的一个字段。

8.2.3. 计数器优化

在经常增加和减少计数，但很少检查计数是否为 0 的场合里，像第四章讨论的那样，维护一个每 CPU 或者每任务计数器很有用。见附录 D.1 中的例子，将每 CPU 计数器应用于 RCU 上。该方法可以避免在增加或减少计数函数中使用原子操作或者内存屏障，但还是要禁止编译器的乱序优化。另外，像 synchronize_srcu()这样的原语，检查总的引用计数是否为 0 的速度十分缓慢。这使得该方法不适合用于频繁获取和释放引用计数的场合，不过对于极少检查引用计数是否为 0 的场合还是适合的。

8.3. Read-Copy Update（RCU）

8.3.1. RCU 基础

本节作者：Paul E. McKenney 和 Jonathan Walpole。

读——拷贝——更新（RCU）是一种同步机制，2002 年 10 月引入 Linux 内核。RCU 允许读操作可以与更新操作并发执行，这一点提升了程序的可扩展性。常规的互斥锁让并发线程互斥执行，并不关心该线程是读者还是写者，而读写锁在没有写者时允许并发的读者，相比于这些常规锁操作，RCU 在维护对象的多个版本时确保读操作保持一致，同时保证只有所有当前读端临界区都执行完毕后才释放对象。RCU 定义并使用了高效并且易于扩展的机制，用来发布和读取对象的新版本，还用于延后旧版本对象的垃圾收集工作。这些机制恰当地在读端和更新端分布工作，让读端非常快速。在某些场合下（比如非抢占式内核里），RCU 读端的函数完全是零开销。

小问题 8.6：顺序锁不也可以让读者和写者并发执行么？

看到这里，读者通常会问，“究竟 RCU 是什么？”，或者是另一个问题“RCU

怎么工作？”（还有些比较少见的情形，读者会断定 RCU 不可能工作）。本节致力于从一种基本的视角回答上述问题，稍后的章节将从用户使用和 API 的视角重新看待这些问题。最后一节会给出一个表。

RCU 由三种基础机制构成，第一个机制用于插入，第二个用于删除，第三个用于让读者可以不受并发的插入和删除干扰。第 8.3.1.1 节描述了发布——订阅机制，用于插入。第 8.3.1.2 节描述了如何等待已有的 RCU 读者来启动删除。第 8.3.1.3 节讨论了如何维护新近更新对象的多个版本，允许并发的插入和删除操作。最后第 8.3.1.4 节对 RCU 的基础进行总结。

8.3.1.1. 发布——订阅机制

RCU 的一个关键特性是可以安全的扫描数据，即使数据此时正被修改。RCU 通过一种发布——订阅机制达成了并发的数据插入。举个例子，假设初始值为 NULl 的全局指针 gp 现在被赋值指向一个刚分配并初始化的数据结构。请见图 8.5 所示的代码片段（还有一些适当的锁操作）。

```
1  struct  foo  {
2     int  a;
3     int  b;
4     int  c;
5  };
6  struct  foo  *gp  =  NULL;
7
8  /*  .  .  .  */
9
10  p  =  kmalloc(sizeof(*p),  GFP_KERNEL);
11  p->a  =  1;
12  p->b  =  2;
13  p->c  =  3;
14  gp  =  p;
```

图 8.5：“发布”的数据结构（不安全）

不幸的是，这块代码无法保证编译器和 CPU 会按照顺序执行最后四条赋值语句。如果对 gp 的赋值发生在初始化 p 的各字段之前，那么并发的读者会读到未初始化的值。这里需要内存屏障来保证事情按顺序发生，可是内存屏障又向来是以难用而闻名的。所以这里我们用一句 rcu_assign_pointer()原语将内存屏障封装起来，让其拥有发布的语义。那么最后四行代码将是这样：

```
1  p->a  =  1;
2  p->b  =  2;
3  p->c  =  3;
4  rcu_assign_pointer(gp,  p);
```

rcu_assign_pointer()“发布”一个新结构，强制让编译器和 CPU 在为 p 的各字段赋值后再去为 gp 赋值。

不过，只保证更新者的执行顺序并不够，因为读者也需要保证读取顺序。请看下面这个例子中的代码：

```
1  p  =  gp;
2  if  (p  !=  NULL)  {
3     do_something_with(p->a,  p->b,  p->c);
4  }
```

这块代码看起来好像不会受到乱序执行的影响，可惜事与愿违，在 DEC Alpha CPU[McK05a，McK05b]机器上，还有启用编译器 value-speculation 优化时，会让 p->a，p->b 和 p->c 的值在 p 赋值之前被读取（luyang：不管你信不信，反正我是信了）。也许在启动编译器的 value-speculation 优化时比较容易观察到这一情形，此时编译器会先猜测 p->a、p->b、p->c 的值，然后再去读取 p 的实际值来检查编译器的猜测是否正确。这种类型的优化十分激进，甚至有点疯狂了，但是这确实发生在 profile-driven 的优化中。

显然，我们必须在编译器和 CPU 层面阻止这种危险的优化。rcu_dereference()原语用了各种内存屏障指令和编译器指令来达到这一目的。

```
1  rcu_read_lock();
2  p  =  rcu_dereference(gp);
3  if  (p  !=  NULL)  {
4     do_something_with(p->a,  p->b,  p->c);
5  }
6  rcu_read_unlock();
```

rcu_dereference()原语用一种“订阅”的办法获取指定指针的值。保证后续的解引用操作可以看见在对应的“发布”操作（rcu_assign_pointer）前进行的初始化。rcu_read_lock()和 rcu_read_unlock()是肯定需要的：这对原语定义了 RCU 读端的临界区。第 8.3.1.2 节将会解释它们的意图，不过请注意，这对原语既不会自旋或者阻塞，也不会阻止 list_add_rcu()的并发执行。事实上，在没有配置 CONFIG_PREEMPT 的内核里，这对原语就是空函数。

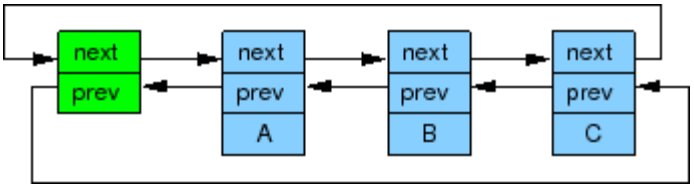


图 8.6：Linux 的循环链表

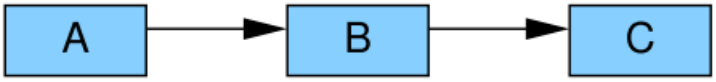


图 8.7：简化表示的 Linux 链表

虽然理论上 rcu_assign_pointer()和 rcu_derederence()可以用于构造任何能想象到的受 RCU 保护的数据结构，但是实践中常常只用于上层的构造。因此 rcu_assign_pointer()和 rcu_dereference()原语是嵌入在特殊的 RCU 变体——即 Linux 操纵链表的 API 中。Linux 有两种双链表的变体，循环链表 struct list_head 和散列表 struct hlist_head/struct hlist_node。前一种如图 8.6 所示，绿格子代表链表头，蓝格子代表链表元素。这种表示方法比较麻烦，所以图 8.7 中给出一种简化表示方法。

```
1  struct  foo  {
2     struct  list_head  *list;
3     int  a;
4     int  b;
5     int  c;
6  };
7  LIST_HEAD(head);
8
9  /*  .  .  .  */
10
11  p  =  kmalloc(sizeof(*p),  GFP_KERNEL);
12  p->a  =  1;
13  p->b  =  2;
14  p->c  =  3;
15  list_add_rcu(&p->list,  &head);
```

图 8.8：RCU 发布链表

图 8.8 是对链表采用指针发布的例子。

第 15 行必须用某些同步机制（最常见的是各种锁）来保护，防止多核 list_add()实例并发执行。不过，同步并不能阻止 list_add()的实例与 RCU 的读者并发执行。

订阅一个受 RCU 保护的链表的代码，很直接：

```
1  rcu_read_lock();
2  list_for_each_entry_rcu(p,  head,  list)  {
3    do_something_with(p->a,  p->b,  p->c);
4  }
5  rcu_read_unlock();
```

list_add_rcu()原语向指定的链表发布了一项条目，保证对应的list_for_each_entry_rcu()可以订阅到同一项条目。

小问题 8.7：如果 list_for_each_entry_rcu()刚好与 list_add_rcu()并发执行时，list_for_each_entry_rcu()为什么没有出现段错误？

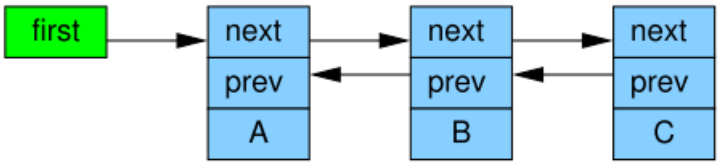


图 8.9：Linux 的线性链表

Linux 的其他双链表、散列表都是线性链表，这意味着它的头结点只需要一个指针，而不是向循环链表那样需要两个，如图 8.9 所示。因此散列表的使用可以减少哈希表的 hash bucket 数组一半的内存消耗。和前面一样，这种表示法太麻烦了，散列表也用和链表一样的简化表达方式，见图 8.7。

```
1  struct  foo  {
2    struct  hlist_node  *list;
3    int  a;
4    int  b;
5    int  c;
6  };
7  HLIST_HEAD(head);
8
9  /*  .  .  .  */
10
11  p  =  kmalloc(sizeof(*p),  GFP_KERNEL);
12  p->a  =  1;
13  p->b  =  2;
14  p->c  =  3;
15  hlist_add_head_rcu(&p->list,  &head);
```

图 8.10：RCU 发布哈希链表

向受 RCU 保护的散列表发布新元素和向循环链表的操作十分类似，见图

8.10。

和之前一样，第 15 行必须用某种同步机制比如锁来保护。

订阅受 RCU 保护的散列表和订阅循环链表没什么区别：

```
1  rcu_read_lock();
2  hlist_for_each_entry_rcu(p,  q,  head,  list)  {
3     do_something_with(p->a,  p->b,  p->c);
4  }
5  rcu_read_unlock();
```

小问题 8.8：为什么我们要给 hlist_for_each_entry_rcu()传递两个指针？list_for_each_entry_rcu()都只需要一个。

表 8.2 是 RCU 的发布和订阅原语，另外还有一个取消发布原语。

表 8.2：RCU 的发布与订阅原语

类别	发布	取消发布	订阅
指针	rcu_assign_pointer()	rcu_assign_pointer(…，NULL)	rcu_dereference()
链表	list_add_rcu() list_add_tail_rcu() list_replace_rcu()	list_del_rcu()	list_for_each_entry_rcu()
哈希链表	hlist_add_after_rcu() hlist_add_before_rcu() hlist_add_head_rcu() Hlist_replace_rcu()	hlist_del_rcu()	hlist_for_each_entry_rcu()

请注意，list_replace_rcu()、list_del_rcu()、hlist_replace_rcu()和 hlist_del_rcu()这些 API 引入了一点复杂性。何时才能安全的释放刚被替换或者删除的数据元素？我们怎么能知道何时所有读者释放了他们对数据元素的引用？

这些问题将在随后的小节里得到解答。

8.3.1.2. 等待已有的 RCU 读者执行完毕

从最基本的角度来说，RCU 就是一种等待事物结束的方式。当然，有很多其他的方式可以用来等待事物结束，比如引用计数，读写锁，事件等等。RCU 的最伟大之处在于它可以等待（比如）20000 种不同的事物，而无需显式地去跟踪它们中的每一个，也无需去担心对性能的影响，对扩展性的限制，复杂的死锁场景，还有内存泄漏带来的危害等等使用显式跟踪手段会出现的问题。

在 RCU 的例子中，被等待的事物称为“RCU 读端临界区”。RCU 读端临界

区从 rcu_read_lock()原语开始，到对应的 rcu_read_unlock()原语结束。RCU 读端临界区可以嵌套，也可以包含一大块代码，只要这其中的代码不会阻塞或者睡眠（虽然有一种叫 SRCU[McK06]的特殊 RCU 类型允许代码在 SRCU 读端临界区中睡眠）。如果您遵守这些约定，那么您可以使用 RCU 去等待任何代码的完成。

RCU 通过间接地确定这些事物何时完成，才完成了这样的丰功伟绩，附录 D 中详细地描述了具体方法。

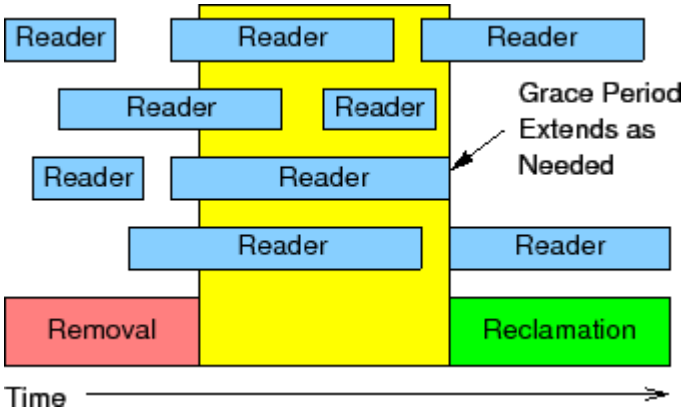


图 8.11：读者和 RCU 优雅周期

尤其如图 8.11 所示，RCU 是一种等待已有的 RCU 读端临界区执行完毕的方法，这里的执行完毕也包括在临界区里执行的内存操作。不过请注意，在某个优雅周期开始后才启动的 RCU 读端临界区会扩展到该优雅周期的结尾处。

下列伪代码展示了使用 RCU 等待读者的基本算法：

1. 作出改变，比如替换链表中的一个元素。
2. 等待所有已有的 RCU 读端临界区执行完毕（比如使用 synchronize_rcu()原语）。这里要注意的是后续的 RCU 读端临界区无法获取刚刚删除元素的引用。
3. 清理，比如释放刚才被替换的元素。

```
1  struct  foo  {
2     struct  list_head  *list;
3     int  a;
4     int  b;
5     int  c;
6  };
7  LIST_HEAD(head);
8
9  /*  .  .  .  */
10
11  p  =  search(head,  key);
```

```
12 if (p == NULL) {
13    /* Take appropriate action, unlock, and
         return. */
14 }
15 q = kmalloc(sizeof(*p), GFP_KERNEL);
16 *q = *p;
17 q->b = 2;
18 q->c = 3;
19 list_replace_rcu(&p->list, &q->list);
20 synchronize_rcu();
21 kfree(p);
```

图8.12：标准RCU替换示例

图 8.12 所示的代码片段演示了这一过程，其中字段 a 作为搜索的关键字。

第 19、20 和 21 行实现了刚才提到的三个步骤。第 16-19 行正如 RCU 其名（读——拷贝——更新）：在允许并发“读”的同时，第 16 行“拷贝”，第 17 到 19 行“更新”。

synchronize_rcu()原语第一眼看上去有点神秘。毕竟它需要等待所有 RCU 读端临界区完成，并且和我们之前看到的一样，rcu_read_lock()和 rcu_read_unlock()原语确定 RCU 读端临界区，如果是在非抢占式内核中，这对原语甚至都是空函数。

这里有一个小花招，由于 rcu_read_lock()和 rcu_read_unlock()划定的 RCU 经典读端临界区不允许阻塞或者睡眠。因此，当某个 CPU 执行上下文切换时，我们能够保证任何之前的 RCU 读端临界区都已经完成了。这就意味着只要每个 CPU 已经执行至少一次上下文切换，那么“所有”之前的 RCU 读端临界区肯定都已经执行完毕了，这也就意味着 synchronize_rcu()可以安全的返回了。

因此，经典的 RCU synchronize_rcu()可以简单理解成下面的语句（请见随后的第 8.3.4 节“玩具”RCU 的实现）：

```
1 for_each_online_cpu(cpu)
2   run_on(cpu);
```

这里的 run_on()将当前线程切换到指定 CPU 上，这会强制该 CPU 发生一次上下文切换。这样通过 for_each_online_cpu()的循环，每个 CPU 都会强制发生上下文切换，从而保证了所有之前的 RCU 读端临界区都已经按照要求执行完毕。虽然这个简单方法在跨越 RCU 读端临界区时禁止抢占的内核上有效，换句话说，就是对 non-CONFIG_PREEMPT 和 CONFIG_PREEMPT 内核有效，但是对 CONFIG_PREEMPT_RT（-rt）实时内核不起作用。所以，实时 RCU 使用了一种

基于引用计数的方法[McK07a]。

当然，实际在 Linux 内核中的实现要复杂的多，因为还需要处理中断、NMI、CPU 热插拔和其他产品级内核需要面对的问题，而且还要保证良好的性能和可扩展性。实时 RCU 实现还必须提供良好的实时响应能力，这就比靠禁止抢占的实现（比如上面那两行代码）复杂多了。

虽然对 synchronize_rcu()的实现在概念上有个简单认识很有用，但是还存在其他问题。比如，在遍历正在被并发更新的链表时，RCU 读者到底看见了什么？这个问题将在下一节予以回答。

8.3.1.3. 维护最近被更新对象的多个版本

本节将展示 RCU 如何维护链表的多个版本，供并发的读者访问。本节通过两个例子来说明在读者还处于 RCU 读端临界区时，被读者引用的数据元素如何保持完整性。第一个例子展示了链表元素的删除，第二个例子展示了链表元素的替换。

例子 1：在删除过程中维护多个版本

在开始“删除”这个例子以前，我们要把图 8.12 的第 11 到 21 行修改成下面的样子：

```
1  p  =  search(head,  key);
2  if  (p  !=  NULL)  {
3     list_del_rcu(&p->list);
4     synchronize_rcu();
5     kfree(p);
6  }
```

这段代码用图 8.13 显示的方式更新链表。每个元素中的三个数字分别代表字段 a、b、c 的值。红色的元素表示 RCU 读者此时正持有该元素的引用。请注意，我们为了让图更清楚，忽略了后向指针和从尾指向头的指针。

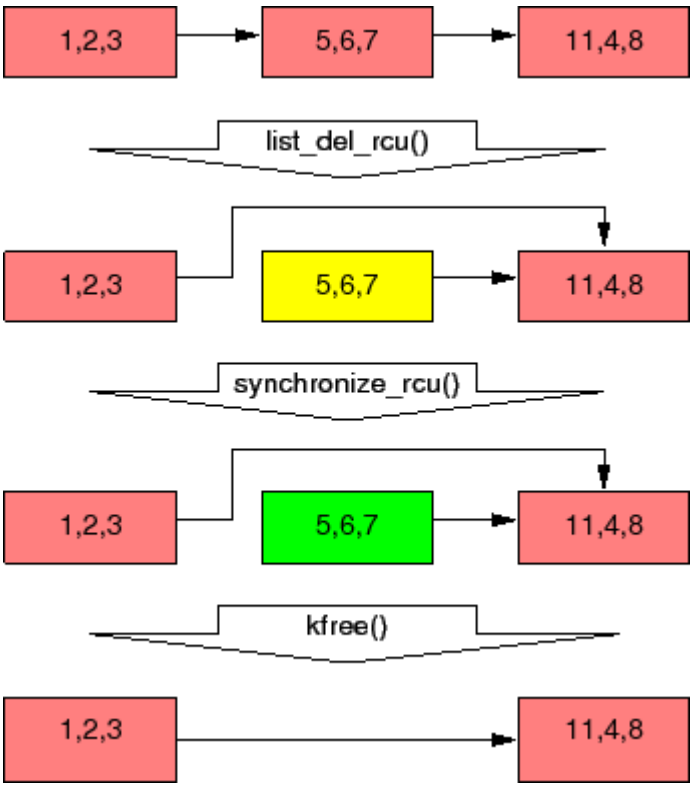


图 8.13：RCU 从链表中删除元素

等第 3 行的 list_del_rcu()执行完毕后，“5、6、7”元素从链表中被删除，如图 8.13 的第二行所示。因为读者不直接与更新者同步，所以读者可能还在并发地扫描链表。这些并发的读者有可能看见，也有可能看不见刚刚被删除的元素，这取决于扫描的时机。不过，刚好在取出指向被删除元素指针后被延迟的读者（比如，由于中断、ECC 内存错误或者配置了 CONFIG_PREEMPT_RT 内核中的抢占），就有可能在删除后还看见链表元素的旧值。因此，我们此时有两个版本的链表，一个有元素“5、6、7”，另一个没有。元素“5、6、7”用黄色标注，表明老读者可能还在引用它，但是新读者已经无法获得它的引用。

请注意，读者不允许在退出 RCU 读端临界区后还维护元素“5、6、7”的引用。因此，一旦第 4 行的 synchronize_rcu()执行完毕，所有已有的读者都要保证已经执行完，不能再有读者引用该元素，如图 8.13 中第三排的绿色部分。这样我们又回到了唯一版本的链表。

此时，元素“5、6、7”可以安全被释放了，如图 8.13 的最后一排所示。这样我们就完成了元素“5、6、7”的删除。本节后面的部分将描述元素的替换。

例子 2：在替换过程中维护多个版本

在开始“替换”例子之前，先给大家看看图 8.12 中所示的最后几行代码：

```
1  q  =  kmalloc(sizeof(*p),  GFP_KERNEL);
2  *q  =  *p;
3  q->b  =  2;
```

```
4  q->c  =  3;
5  list_replace_rcu(&p->list,  &q->list);
6  synchronize_rcu();
7  kfree(p);
```

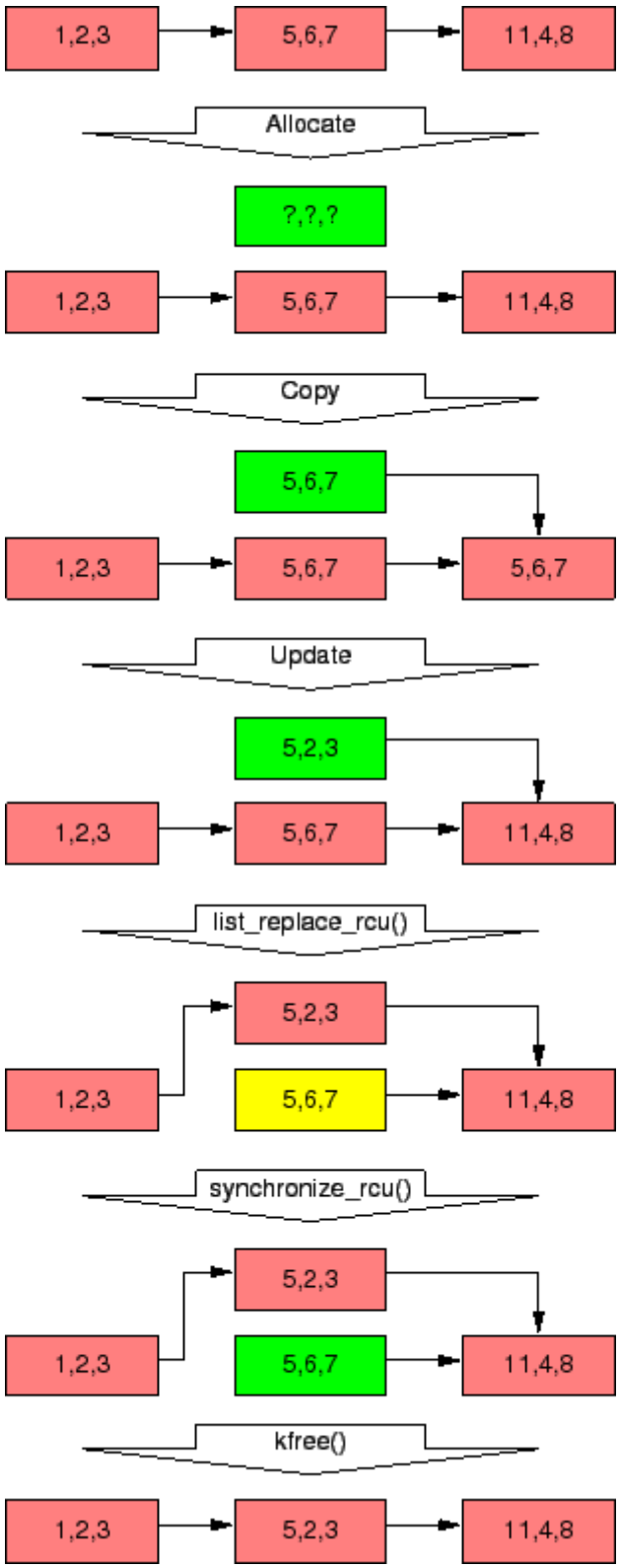


图8.14：RCU 从链表中替换元素

链表的初始状态包括指针 p 都和“删除”例子中一样，如图 8.14 的第一排所示。

和前面一样，每个元素中的三个数字分别代表字段 a、b、c。红色的元素表示读者可能正在引用，并且因为读者不直接与更新者同步，所以读者有可能与整个替换过程并发执行。请注意我们为了图表的清晰，再一次忽略了后向指针和从尾指向头的指针。

下面的文件描述了元素“5、2、3”如何替换元素“5、6、7”的过程，任何特定读者可能看见这两个值其中一个。

第 1 行用 kmalloc()分配了要替换的元素，如图 8.14 第二排所示。此时，没有读者持有刚分配的元素的引用（用绿色表示），并且该元素是未初始化的（用问号表示）。

第 2 行将旧元素复制给新元素，如图 8.14 中第三排所示。新元素此时还不能被读者访问，但是已经初始化了。

第 3 行将 q->b 的值更新为 2，第 4 行将 q->c 的值更新为 3，如图 8.14 中第四排所示。

现在，第 5 行开始替换，这样新元素终于对读者可见了，因此颜色也变成了红色，如图 8.14 第五排所示。此时，链表就有两个版本了。已经存在的老读者可能看到元素“5,6,7”（现在颜色是黄色的），而新读者将会看见元素“5,2,3”。不过这里可以保证任何读者都能看到一个完好的链表。

随着第 6 行 synchronize_rcu()的返回，优雅周期结束，所有在 list_replace_rcu()之前开始的读者都已经完成。特别是任何可能持有元素“5,6,7”引用的的读者保证已经退出了它们的 RCU 读端临界区，不能继续持有引用。因此，不再有任何读者持有旧数据的引用，如图 8.14 中第六排绿色部分所示。这样我们又回到了单一版本的链表，只是用新元素替换了旧元素。

等第 7 行的 kfree()完成后，链表就成为了图 8.14 最后一排的样子。

不过尽管 RCU 是因替换的例子而得名的，但是 RCU 在内核中的主要用途还是第 8.3.1.3 节中简单的删除例子一样。

讨论

上述这些例子假设整个更新操作都持有了一把互斥锁，这意味着任意时刻最多只会有两个版本的链表。

小问题 8.9：如果要您修改删除的例子，允许多于两个版本的链表被激活，您该怎么做呢？

小问题 8.10：在任意时刻一个链表能有多少个 RCU 版本被激活？

这个事件序列显示了 RCU 更新如何使用多个版本，在有读者并发的情况下安全地执行改变。当然，有些算法无法优雅地处理多个版本。有些技术在 RCU

中采用了这些算法[McK04]，但是这超过了本节的范围。

8.3.1.4. RCU 基础总结

本节描述了 RCU 算法的三个基本组件：

1. 添加新数据的发布——订阅机制。
2. 等待已有 RCU 读者结束的方法，以及
3. 维护多个版本数据的准则，允许在不影响或者延迟其他并发 RCU 读者的前提下改变数据。

小问题 8.11：rcu_read_lock()和 rcu_read_unlock()既没有自旋也没有阻塞，RCU 的更新者怎么会让 RCU 读者等待？

这三个 RCU 组件使得数据可以在有并发读者时被改写，通过不同方式的组合，这三种组件可以实现各种基于 RCU 算法的变体，我们在下面的小节将介绍其中一部份。

8.3.2. RCU 用法

本节将从使用 RCU 的视角，以及使用哪种 RCU 的角度来回答上一节的问题“什么是 RCU？”因为 RCU 最常用的目的是替换已有的机制，所以我们首先看看 RCU 与这些机制之间的关系，如表 8.3 所示。在表 8.3 所列章节之后，第 8.3.2.8 节进行了一番总结。

表 8.3：RCU 用法

RCU 可以替代的机制	小节
读写锁	8.3.2.1
受限制的引用计数机制	8.3.2.2
Bulk 引用计数机制	8.3.2.3
穷人版的垃圾回收器	8.3.2.4
存在担保	8.3.2.5
类型安全的内存	8.3.2.6
等待事物结束	8.3.2.7

8.3.2.1. RCU 是读写锁的替代者

也许在 Linux 内核中 RCU 最常见的用途就是在读占大多数时间的情况下替换读写锁了。可是在一开始我并没有想到 RCU 的这个用途，事实上在上世纪九

十年代初期，我在实现通用 RCU 实现之前选择实现了一种类似 brlock 的东西。我为 brlock 原型想象的每个用途都被 RCU 实现过了。事实上，在 brlock 原型第一次使用时 RCU 已经用了不止三年了。兄弟们，我是不是看起来很傻！

RCU 和读写锁最关键的相似之处在于两者都有可以并行执行的读端临界区。事实上，在某些情况下，完全可以从机制上用对应的读写锁 API 来替换 RCU 的 API。不过，等等，这样做有什么必要？

RCU 的优点在于性能、没有死锁，还有实时的延迟。当然 RCU 也有一点点缺点，比如读者与更新者并发执行，比如低优先级 RCU 读者可以阻塞正等待优雅周期完毕的高优先级线程，还比如优雅周期的延迟可以有好几毫秒。这些优点和缺点在后面的小节中进行讨论。

性能

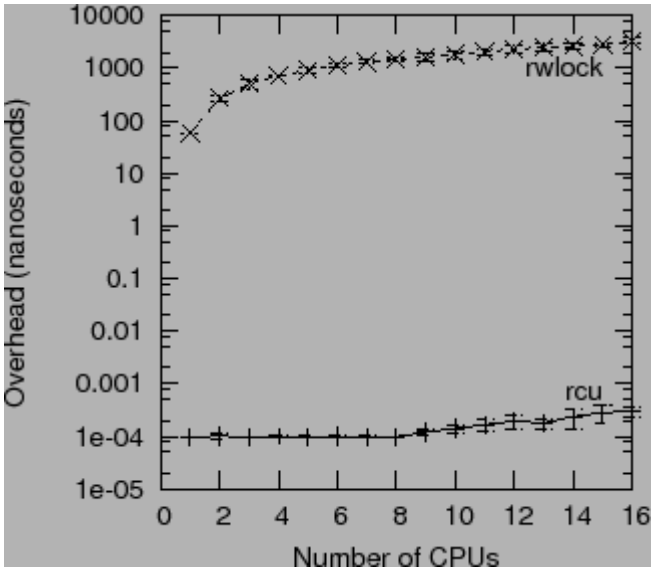


图 8.15：RCU 相较于读写锁的读端性能优势

RCU 相较于读写锁的读端性能优势，见图 8.15。

小问题 8.12：我 X!你想让我相信 RCU 在 3GHz 的时钟周期也就是超过 300 皮秒（译注：百亿分之一秒，10 的负 12 次方）时只有 100 飞秒（译注：百兆分之一秒，10 的负 15 次方）的开销？

请注意，在单个 CPU 上读写锁比 RCU 慢一个数量级，在 16 个 CPU 上读写锁比 RCU 要慢三个数量级。与此相反，RCU 就扩展的很好。在上面两个例子中，错误曲线都几乎是水平的。

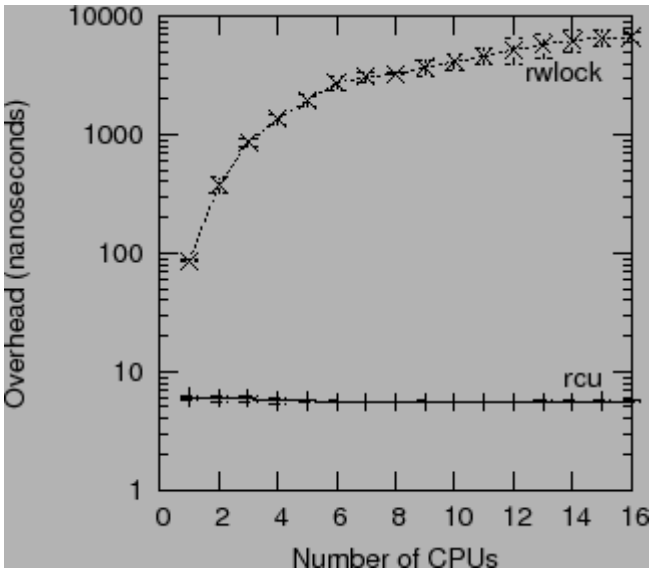


图8.16：可抢占RCU相较于读写锁的性能优势

更温和的视角来自CONFIG_PREEMPT内核，虽然RCU仍然超过了读写锁一到三个数量级，如图8.16所示。请注意读写锁在CPU数目很多时的陡峭曲线。RCU的错误曲线几乎是一条水平线。

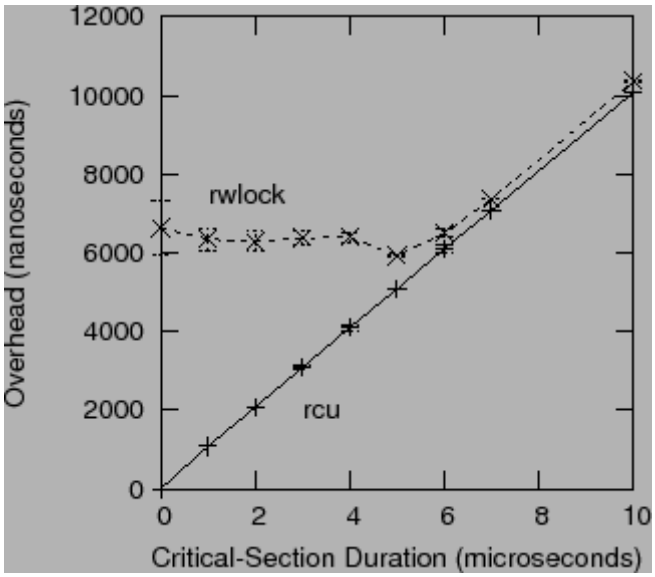


图8.17：RCU与读写锁在临界区增长时的性能比较

当然，在图8.16中，由于不现实的零临界区长度，读写锁的低性能被夸大了。随着临界区的增长，RCU的性能优势也不再显著，在图8.17的16个CPU的系统里，y轴代表读端原语的总开销，x轴代表临界区长度。

小问题8.13：为什么rwlock的开销和变化率随着临界区开销的增长而下降？

但是，考虑到很多系统调用（以及它们所包含的RCU读端临界区）都能在几毫秒内完成，所以这个结果对RCU是有利的。

另外，下一节将会讨论，RCU读端原语基本上是不会死锁的。

免于死锁

虽然 RCU 在多数为读的工作负荷下提供了显著的性能优势，但是使用 RCU 的主要目标却是它可以免于读端死锁的特性。这种免于死锁的能力来源于 RCU 的读端原语不阻塞、不自旋，甚至不会向后跳转，所以 RCU 读端原语的执行时间是确定的。这使得 RCU 读端原语不可能组成死锁循环。

小问题 8.14：RCU 读端原语免于死锁的能力有例外吗？如果有，哪种事件序列会造成死锁？

RCU 读端免于死锁的能力带来了一个有趣的后果，RCU 读者可以无条件地升级为 RCU 更新者。在读写锁中尝试这种升级则会造成死锁。进行 RCU 读者到更新者提升的代码片段如下所示：

```
1  rcu_read_lock();
2  list_for_each_entry_rcu(p,  &head,  list_field)  {
3    do_something_with(p);
4    if  (need_update(p))  {
5       spin_lock(my_lock);
6       do_update(p);
7       spin_unlock(&my_lock);
8    }
9  }
10  rcu_read_unlock();
```

请注意，do_update()是在锁的保护下执行，也是在 RCU 读端的保护下执行。

RCU 免于死锁的特性带来的另一个有趣后果是 RCU 不会受很多优先级反转问题影响。比如，低优先级的 RCU 读者无法阻止高优先级的 RCU 更新者获取更新端锁。类似地，低优先级的更新者也无法阻止高优先级的 RCU 读者进入 RCU 读端临界区。

实时延迟

因为 RCU 读端原语既不自旋也不阻塞，所以这些原语有着极佳的实时延迟。另外，如之前所说，这也就意味着这些原语不会受与 RCU 读端原语和锁有关的优先级反转影响。

但是，RCU 还是会受到更隐晦的优先级反转问题影响，比如，在等待 RCU 优雅周期结束时阻塞的高优先级进程，会被-rt 内核的低优先级 RCU 读者阻塞。这可以用 RCU 优先级提升[McK07d, GMTW08]解决。

RCU 读者与更新者并发执行

因为 RCU 读者既不自旋也不阻塞，还因为 RCU 更新者没有任何类似回滚（rollback）或者中止（abort）的语义，所以 RCU 读者和更新者必然可以并发执行。这意味着 RCU 读者有可能访问旧数据，还有可能发现数据不一致，无论 0

这两个问题中的哪一个都有可能让读写锁卷土重来。

不过，在令人吃惊的大量情景中，数据不一致和旧数据都不是问题。网络路由表是一个经典例子。因为路由的更新可能要花相当长一段时间才能到达指定系统（几秒甚至几分钟），所以系统可能会在更新到来后的一段时间内仍然将报文发到错误的地址去。通常在几毫秒内将报文发送到错误地址并不算什么问题。并且，因为 RCU 的更新可以在无需等待 RCU 读者执行完毕的情况下发生，所以 RCU 读者可能会比读写锁的读者更早地看见更新后的路由表，如图 8.18 所示。

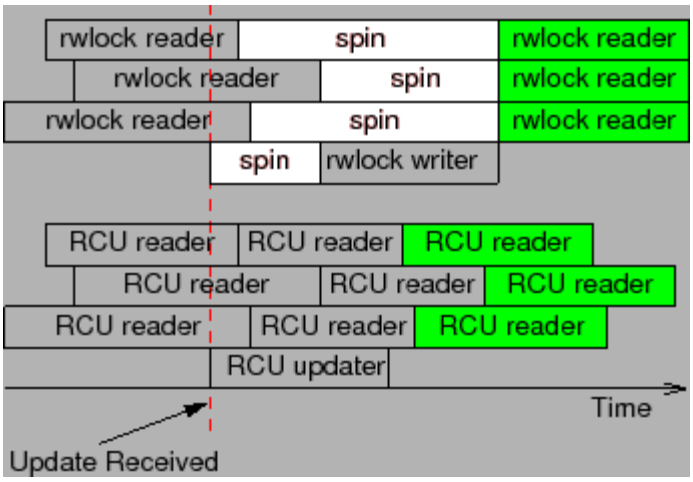


图 8.18：RCU 与读写锁在响应时间上的比较

一旦收到更新，rwlock 的写者在最后一个读者完成之前不能继续执行，后续的读者在写者更新完毕之前也不能去读。不过，这一点也保证了后续的读者可以看见最新的值，如图中绿色的部分。相反，RCU 读者和更新者相互不会阻塞，这就允许 RCU 读者可以更快地看见更新后的值。当然，因为读者和更新者的执行重叠了一部分，所以所有 RCU 读者都“可能”看见更新后的值，包括图中三个在更新者之前就已开始的 RCU 读者。然而，再一次强调，只有绿色的 RCU 读者才能“保证”看到更新后的值，如图 8.18 所示。

简单地说，读写锁和 RCU 提供了不同的保证。在读写锁中，任何在写者之后开始的读者都“保证”能看到新值，而在写者正在自旋时开始的读者有可能看见新值，也有可能看见旧值，这取决于读写锁实现中的读者/写者哪一个优先。与之相反，在 RCU 中，在更新者完成后才开始的读者都“保证”能看见新值，在更新者开始后才完成的读者有可能看见新值，也有可能看见旧值，这取决于具体的时机。

这里面的关键点是，虽然限定在计算机系统这一范围读写锁保证了一致性，但是这种一致性是以增加外部世界的不一致作为代价的。换句话说，读写锁以外部世界的旧数据作为代价，获取了内部的一致性。

然而，在限定系统中还是存在无法容忍的不一致和旧数据。幸运的是，有很多种办法可以避免这种不一致和旧数据[McK04, ACMS03]，有一些方法是基于第

8.2 节提到的引用计数。

低优先级 RCU 读者可以阻塞高优先级的回收者

在随后的章节将会逐一介绍实时 RCU[GMTW08]（见附录 D.4）、SRCU[McK06]（见附录 D.1）以及 QRCU[McK07f]（见附录 E.6）， 在它们中抢占的读者会打断正在进行中的优雅周期，即使高优先级的任务因为等待优雅周期完成而阻塞也是如此。实时 RCU 可以通过用 call_rcu()替换 synchronize_rcu()来避免此问题，或者采用 RCU 优先级提升来避免[McK07d, GMTW08]，不过该方法在 2008 年初还处于实验状态。虽然有必要讨论 SRCU 和 QRCU 的优先级提升，可是现在实时领域还没有明确的需求。

延续好几个毫秒的 RCU 优雅周期

除了 QRCU 和第 8.3.4 节提到的几个“玩具”RCU 实现，RCU 优雅周期会延续好几个毫秒。虽然有一些手段可以消除这样长的延迟带来的损害，比如使用在可能时使用异步接口（call_rcu()和 call_rcu_bh()），但是根据拇指定律，这也是 RCU 使用在读数据占多数的情景的主要原因。

读写锁与 RCU 代码对比

在最好的情况下，将读写锁转换成 RCU 非常简单，如图 8.19 到 8.21 所示，这些都来自 Wikipedia[MPA+06]。

```
1 struct el {
2   struct list_head lp;
3   long key;
4   spinlock_t mutex;
5   int data;
6   /* Other data fields */
7 };
8 DEFINE_RWLOCK(listmutex);
9 LIST_HEAD(head);
```

```
1 struct el {
2   struct list_head lp;
3   long key;
4   spinlock_t mutex;
5   int data;
6   /* Other data fields */
7 };
8 DEFINE_SPINLOCK(listmutex);
9 LIST_HEAD(head);
```

图 8.19：将读写锁转换为 RCU：数据

```
1 int search(long key, int *result)
2 {
3   struct el *p;
4
5   read_lock(&listmutex);
6   list_for_each_entry(p, &head, lp) {
7     if (p->key == key) {
8       *result = p->data;
9       read_unlock(&listmutex);
10      return 1;
11    }
12  }
13  read_unlock(&listmutex);
14  return 0;
15 }
```

```
1 int search(long key, int *result)
2 {
3   struct el *p;
4
5   rcu_read_lock();
6   list_for_each_entry_rcu(p, &head, lp) {
7     if (p->key == key) {
8       *result = p->data;
9       rcu_read_unlock();
10      return 1;
11    }
12  }
13  rcu_read_unlock();
14  return 0;
15 }
```

图 8.20：将读写锁转换为 RCU：搜索

```
 1 int delete(long key)
 2 {
 3   struct el *p;
 4
 5   write_lock(&listmutex);
 6   list_for_each_entry(p, &head, lp) {
 7     if (p->key == key) {
 8       list_del(&p->lp);
 9       write_unlock(&listmutex);
10       kfree(p);
11       return 1;
12     }
13   }
14   write_unlock(&listmutex);
15   return 0;
16 }
```

```
 1 int delete(long key)
 2 {
 3   struct el *p;
 4
 5   spin_lock(&listmutex);
 6   list_for_each_entry(p, &head, lp) {
 7     if (p->key == key) {
 8       list_del_rcu(&p->lp);
 9       spin_unlock(&listmutex);
10       synchronize_rcu();
11       kfree(p);
12       return 1;
13     }
14   }
15   spin_unlock(&listmutex);
16   return 0;
17 }
```

图8.21：将读写锁转换成RCU：删除

详细阐述如何用 RCU 替换读写锁已经超出了本书的范围。

8.3.2.2. RCU 是一种受限制的引用计数机制

因为优雅周期不能在 RCU 读端临界区仍在进行时完毕，所以 RCU 读端原语可以像受限的引用计数机制一样使用。比如考虑下面的代码片段：

```
1 rcu_read_lock();    /* acquire reference. */
2 p = rcu_dereference(head);
3 /* do something with p. */
4 rcu_read_unlock();    /* release reference. */
```

rcu_read_lock()原语可以看做是获取对 p 的引用，因为在 rcu_dereference()为 p 赋值之后才开始的优雅周期无法在配对的 rcu_read_unlock()之前完成。这种引用计数机制是受限制的，因为我们不允许在 RCU 读端临界区中阻塞，也不允许将一个任务的 RCU 读端临界区传递给另一个任务。

不管上述的限制，下列代码可以安全地删除 p：

```
1 spin_lock(&mylock);
2 p = head;
3 rcu_assign_pointer(head, NULL);
4 spin_unlock(&mylock);
5 /* Wait for all references to be released. */
6 synchronize_rcu();
7 kfree(p);
```

将 p 赋值为 head 阻止了任何获取将来对 p 的引用的操作，synchronize_rcu()等待所有之前获取的引用释放。

小问题 8.15：但是等等！这和之前思考 RCU 是一种读写锁的替代者的代码

完全一样！发生了什么事？

当然，RCU 也可以与传统的引用计数结合，LKML 中对此有过讨论，第 8.2 节也做出了总结。

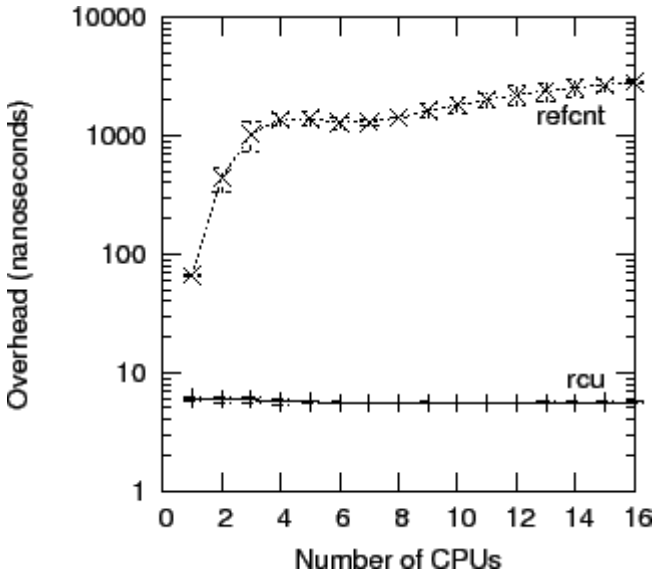


图 8.22：RCU 与引用计数的性能比较

但是何必这么麻烦？我再一次回答，部分原因是性能，如图 8.22 所示，图中再一次显示了在 16 个 3GHz CPU 的 Intel x86 系统中采集的数据。

小问题 8.16：为什么引用计数的开销在 6 个 cpu 左右时有一点下降？

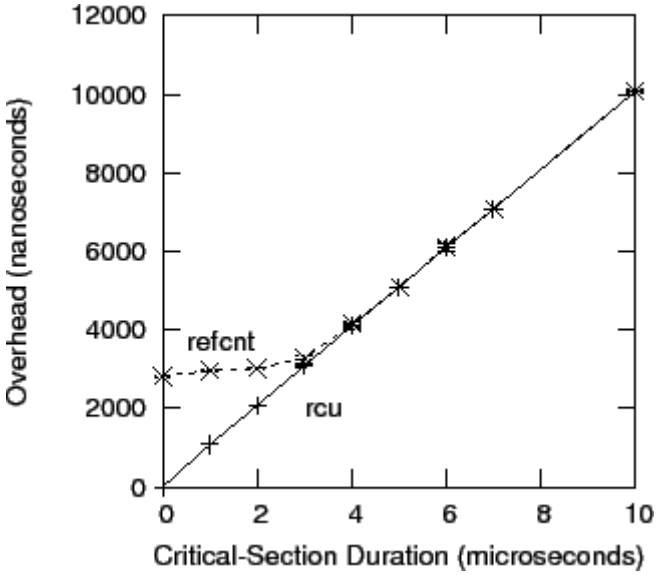


图 8.23：RCU 与引用计数的响应时间比较

并且，和读写锁一样，RCU 的性能优势主要来源于较短的临界区，如图 8.23 中的 16 核系统所示。另外，和读写锁一样，许多系统调用（以及它们包含的任何 RCU 读端临界区）都在几毫秒内完成。

但是，伴随着 RCU 的限制有可能相当麻烦。比如，在许多情况下，在 RCU 读端临界区中禁止睡眠可能与我们的整个目的不符。下一节将从处理该问题的方

法出发，同时涉及在某些情况下如何降低传统引用计数的复杂性。

8.3.2.3. RCU 是一种 bulk 引用计数机制

前面的章节曾经说过，传统的引用计数器通常与某种或者一组数据结构有联系。然而，维护大量不同种类的数据结构的单一全局引用计数，通常会导致包含引用计数的缓存线来回“乒乓”。这种缓存线“乒乓”会严重影响系统性能。

相反，RCU 的轻量级读端原语允许读端极其频繁地调用，却只带来微不足道的性能影响，这使得 RCU 可以作为一种几乎没有惩罚的“bulk reference-counting”机制。可睡眠 RCU（SRCU）[McK06]可以接受单个任务必须在会阻塞的代码中持有一个引用的情况（这句翻译的太差了）。但是这种没有包含特殊的情景，一个任务将引用传递给另一个引用，在开始一次 I/O 时获取引用，然后当对应的 I/O 完成时在中断处理函数里释放该引用。（原则上 SRCU 的实现可以处理这一点，但是在实践中还不清楚这是否是一个好的权衡。）

当然，SRCU 带来了它自己的限制条件，即要传递给对应的 srcu_read_unlock() 的 srcu_read_lock()的返回值，以及硬件中断处理函数或者 NMI/SMI 处理函数不能调用 SRCU 原语。SRCU 的限制会带来多少问题，如何更好地处理这些问题，这一切尚未有结论。

8.3.2.4. RCU 是穷人版的垃圾回收器（garbage collector）

当人们刚开始学习 RCU 时，有种比较少见的感叹是“RCU 有点像垃圾回收器！”。这种感叹有一部分是对的，不过还是会给学习造成误导。

也许思考 RCU 与垃圾自动回收器（GC）之间关系的最好办法是，RCU 类似自动决定回收时机的 GC，但是 RCU 与 GC 有几点不同：（1）程序员必须手动指示何时可以回收指定数据结构，（2）程序员必须手动标识出可以合法持有引用的 RCU 读端临界区。

尽管存在这些差异，两者的相似程度仍然相当得高，就我所知至少有一篇理论分析 RCU 的文献曾经分析过两个的相似度。不仅如此，我所知道的第一种类 RCU 的机制就运用垃圾回收器来处理优雅周期。然后，下面一节提供了一种更好地思考 RCU 的方法。

8.3.2.5. RCU 是一种提供存在担保的方法

Gamsa et al.[GKAS99]讨论了存在担保，并且描述了如何用一种类似 RCU 的

机制提供这种存在担保，第 6.3 节讨论了如何通过锁来提供存在担保还有这样做的不利之处。如果任何受 RCU 保护的数据元素在 RCU 读端临界区中被访问，那么数据元素在 RCU 读端临界区持续期间保证存在。

```
1  int  delete(int  key)
2  {
3     struct  element  *p;
4     int  b;
5     5
6     b  =  hashfunction(key);
7     rcu_read_lock();
8     p  =  rcu_dereference(hashtable[b]);
9     if  (p  ==  NULL  ||  p->key  !=  key)  {
10        rcu_read_unlock();
11        return  0;
12     }
13     spin_lock(&p->lock);
14     if  (hashtable[b]  ==  p  &&  p->key  ==  key)  {
15        rcu_read_unlock();
16        hashtable[b]  =  NULL;
17        spin_unlock(&p->lock);
18        synchronize_rcu();
19        kfree(p);
20        return  1;
21     }
22     spin_unlock(&p->lock);
23     rcu_read_unlock();
24     return  0;
25  }
```

图8.24：带存在担保的每数据元素锁

图 8.24 展示了基于 RCU 的存在担保如何通过从哈希表删除元素的函数来实现每数据元素锁。第 6 行计算哈希函数，第 7 行进入 RCU 读端临界区。如果第 9 行发现哈希表对应的哈希项（bucket）为空，或者数据元素不是我们想要删除的那个，那么第 10 行退出 RCU 读端临界区，第 11 行返回错误。

小问题 8.17：如果图 8.24 中第 9 行链表中的第一个元素不是我们想要删除的元素，那该怎么办？

如果第 9 行判断为 false，第 13 行获取更新端的自旋锁，然后第 14 行检查元素是否还是我们想要的。如果是，第 15 行退出 RCU 读端临界区，第 16 行从哈希表中删除找到的元素，第 17 行释放锁，第 18 行等待所有之前已经存在的 RCU 读端临界区退出，第 19 行释放刚被删除的元素，最后第 20 行返回成功。如果 14 行的判断发现元素不再是我们想要的，那么第 22 行释放锁，第 23 行退出 RCU 读端临界区，第 24 行返回错误以删除该关键字。

小问题 8.18：为什么图 8.24 第 15 行可以在第 17 行释放锁之前就退出 RCU 读端临界区？

小问题 8.19：为什么图 8.24 第 23 行不能在第 22 行释放锁之前就退出 RCU 读端临界区？

细心的读者可能会发现，这个例子只不过是第 8.3.2.7 小节“RCU 是一种等待事物结束的方法”中那个例子的变体。细心的读者还会发现对死锁免疫要比第 6.3 节讨论的基于锁的存在担保更好。

8.3.2.6. RCU 是一种提供类型安全内存的方法

很多无锁算法并不需要数据元素在被 RCU 读端临界区引用时保持完全一致，只要数据元素的类型不变就可以了。换句话说，只要结构类型不变，无锁算法可以允许某个数据元素在被其他对象引用时可以释放并重新分配，但是决不允许类型上的改变。这种“保证”，在学术文献中被称为“类型安全的内存”（type-safe meomry）[GC96]，比前一节提到的存在担保要弱一些，因此处理起来也要困难一些。类型安全的内存算法在 Linux 内核中的应用是 slab 缓存，被 SLAB_DESTROY_BY_RCU 标记专门标识出来的缓存通过 RCU 将已释放的 slab 返回给系统内存。在任何已有的 RCU 读端临界区持续期间，使用 RCU 可以保证所有带有 SLAB_DESTROY_BY_RCU 标记且正在使用的 slab 元素仍然在该 slab 中，类型保持一致。

小问题 8.20：如果在多个线程中，每个线程都有任意长的 RCU 读端临界区，那么在任何时刻系统中至少有一个线程正处于 RCU 读端临界区吗？这不会阻止数据从带有 SLAB_DESTROY_BY_RCU 标记的 slab 回到系统内存吗？不会造成 OOM 吗？

这些算法一般使用了一个验证步骤，用于确定刚刚被引用的数据结构确实是被请求的数据[LS86 第 2.5 节]。这种验证要求数据结构的一部分不能被释放-重分配过程触碰。通常这种有效性检查很难保证不存在隐晦且难解决的 Bug。

因此，虽然基于类型安全的无锁算法在一种很难达到的情景下非常有效，但是你最好还是尽量使用存在担保。毕竟简单总是更好的。

8.3.2.7. RCU 是一种等待事物结束的方式

在 8.3.1 节我们提过 RCU 的一个重要组件是等待 RCU 读者结束的方法。RCU 的强大之处，其中之一就是允许您在等待上千个不同事物结束的同时，又不用显式地去跟踪其中每一个，因此也就无需担心性能下降、扩展限制、复杂的死锁场景、内存泄露等显式跟踪机制自身的问题。

在本节中，我们将展示 synchronize_sched()的读端版本（包括禁止抢占，禁止中断的原语）如何让您实现与不可屏蔽中断（NMI）处理函数的交互，如果用锁来实现，这将极其困难。这种方法被称为“纯 RCU”[McK04]，Linux 内核的多处地方使用了这种方法。

“纯 RCU”设计的基本形式如下：

1. 做出改变，比如，OS 对一个 NMI 做出反应。
2. 等待所有已有的读端临界区完全退出（比如使用 synchronize_sched()原语）。这里的关键之处是后续的 RCU 读端临界区保证可以看见变化发生后的样子。
3. 扫尾工作，比如，返回表明改变成功完成的状态。

本节剩下的部分将用 Linux 内核中的例子做展示。在下面这个例子中，timer_stop()函数使用 synchronize_sched()确保在释放相关资源之前，所有正在处理的 NMI 处理函数已经完成。图 8.25 是对该例简化后的代码。

```
1  struct  profile_buffer  {
2     long  size;
3     atomic_t  entry[0];
4  };
5  static  struct  profile_buffer  *buf  =  NULL;
6
7  void  nmi_profile(unsigned  long  pcvalue)
8  {
9     struct  profile_buffer  *p  =
                        rcu_dereference(buf);
10
11     if  (p  ==  NULL)
12        return;
13     if  (pcvalue  >=  p->size)
14        return;
15     atomic_inc(&p->entry[pcvalue]);
```

```
16  }
17
18  void  nmi_stop(void)
19  {
20     struct  profile_buffer  *p  =  buf;
21
22     if  (p  ==  NULL)
23        return;
24     rcu_assign_pointer(buf,  NULL);
25     synchronize_sched();
26     kfree(p);
27  }
```

图 8.25：用 RCU 等待 NMI 结束

第 1 到 4 行定义了 profile_buffer 结构，包含一个大小和一个变长数组的入口。第 5 行定义了指向 profile_buffer 的指针，这里假设别处对该指针进行了初始化，指向内存的动态分配区。

第 7-16 行定义了 nmi_profile()函数，供 NMI 中断处理函数调用。该函数不会被抢占，也不会被普通的中断处理函数中断，但是，该函数还是会受 cache miss、ECC 错误，以及被同一个核的其他硬件线程抢占时钟周期等因素影响。第 9 行使用 rcu_dereference()原语来获取指向 profile_buffer 的本地指针，这样做是为了确保在 DEC Alpha 上的内存顺序执行，如果当前没有分配 profile_buffer，第 11 行和 12 行退出，如果参数 pcvalue 超出范围，第 13 和 14 行退出。否则，第 15 行增加以参数 pcvalue 为下标的 profile_buffer 项的值。请注意，profile_buffer 结构中的 size 保证了 pcvalue 不会超出 buffer 的范围，即使突然将较大的 buffer 替换成了较小的 buffer 也是如此。

第 18-27 行定义了 nmi_stop()函数，由调用者负责互斥访问（比如持有正确的锁）。第 20 行获取 profile_buffer 的指针，如果没有 buffer，第 22 和 23 行退出。否则，第 24 行将 profile_buffer 的指针置 NULL（使用 rcu_assign_pointer()原语在弱顺序的机器中保证内存顺序访问），第 25 行等待 RCU Sched 的优雅周期结束，尤其是等待所有不可抢占的代码——包括 NMI 中断处理函数——结束。一旦执行到第 26 行，我们就可以保证所有获取到指向旧 buffer 指针的 nmi_profile()实例都已经返回了。现在可以安全释放 buffer 了，这时使用 kfree()原语。

小问题 8.21：假设 nmi_profile()函数可以被抢占。那么怎么做才能让我们的例子正常工作？

简而言之，RCU 让 profile_buffer 动态切换变得更简单（您可以试试原子操

作，或者还可以用锁来折磨下自己）。但是，RCU 通常还是运用在更高的抽象层次上的，正如之前几个小节那样。

8.3.2.8. RCU 用法总结

RCU 的核心只是提供以下功能的 API：

1. 用于添加新数据的发布——订阅机制。
2. 等待已有 RCU 读者结束的方法，以及
3. 维护多版本的准则，使得在有 RCU 读者并发时不会影响或延迟数据更新。

也就是说，在 RCU 之上建造更高抽象级别的架构是可能的，比如前几节列出的读写锁、引用计数和存在担保。更进一步，我对 Linux 社区会继续为 RCU 寻找新用法丝毫不感到怀疑，当然其他的同步原语肯定也是这样。

8.3.3. Linux 内核中的 RCU API

本节以 Linux 内核中的 API 作为视角来看待 RCU。第 8.3.3.1 节列出了 RCU 的等待完成 API，第 8.3.3.2 节列出了 RCU 的发布——订阅和版本维护 API。最后，第 8.3.3.4 节给除了总结性的评论。

8.3.3.1. RCU 有一个等待完成 API 族

对“什么是 RCU”这个问题，最直接的回答是 RCU 是一种在 Linux 内核中使用的 API，表 8.4 和表 8.5 分别总结了不可睡眠 RCU 和可睡眠 RCU 的等待 RCU 读者结束 API，表 8.6 列出了发布/订阅 API。

表 8.4：RCU 的等待事件结束 API

属性	经典 RCU	RCU BH	RCU Sched	实时 RCU
目的	最初版本	防止 DDoS 攻击	等待不可抢占的代码：硬中断、NMI	实时响应
合入内核版本	2.5.43	2.6.9	2.6.12	2.6.26
读端原语	rcu_read_lock()! rcu_read_unlock()!	rcu_read_lock_bh() rcu_read_unlock_bh()	preempt_disable() preempt_enable() (以及类似函数)	rcu_read_lock() rcu_read_unlock()
更新端原语（同步）	synchronize_rcu() synchronize_net()		synchronize_sched()	synchronize_rcu() synchronize_net()

属性	经典 RCU	RCU BH	RCU Sched	实时 RCU
更新端原语（异步/回调）	call_rcu()!	call_rcu_bh()	call_rcu_sched()	call_rcu()
更新端原语（等待回调）	rcu_barrier()	rcu_barrier_bh()	rcu_barrier_sched()	rcu_barrier()
类型安全的内存	SLAB_DESTROY_BY_RCU			SLAB_DESTROY_BY_RCU
读端限制	不能阻塞	不能开中断	不能阻塞	只能抢占或者获取锁
读端开销	禁止抢占/打开抢占（在 No-preempt 内核中无此开销）	禁止 BH/打开 BH	禁止抢占/打开抢占（在 No-preempt 内核中无此开销）	简单指令 关中断/开中断
异步更新端开销	低于微秒级	低于微秒级		低于微秒级
优雅周期延迟	几十毫秒	几十毫秒	几十毫秒	几十毫秒
Non-preempt-RT 实现	经典 RCU	RCU BH	经典 RCU	可抢占 RCU
preempt-RT 实现	可抢占 RCU	实时 RCU	在所有 CPU 上强制调度	实时 RCU

如果您刚刚接触 RCU，那么您只需要将注意力放到表 8.4 的一列就好，该表中的每一列都总结了 Linux 内核 RCU API 家族中的一个成员。比如，如果您对理解 Linux 内核如何使用 RCU 比较感兴趣，那么“RCU Classic”就是一个不错的起点，因为它应用的最频繁。另一方面，如果您对理解 RCU 自身感兴趣，那么“SRCU”的 API 最简单。当然您随时都可以从其他列开始研究。

如果您已经对 RCU 很熟悉了，这几个表可以看做是有用的参考。

小问题 8.22：为什么表 8.4 的有些格子里带有一个惊叹号（“!”）？

“RCU Classic”一列对应的是 RCU 的原始实现，rcu_read_lock()和 rcu_read_unlock()原语划分出 RCU 读端临界区，可以嵌套使用。对应的同步的更新端原语 synchronize_rcu()和它的兄弟 synchronize_net()都是等待当前正在执行的 RCU 读端临界区退出。等待时间被称为“优雅周期”。异步的更新端原语 call_rcu()在后续的优雅周期结束后调用由参数指定的函数。比如，call_rcu(p, f); 在优雅周期结束后执行 RCU 回调 f(p)。有一些场景，比如使用 call_rcu()卸载 Linux 内核模块时，需要等待所有未完成的 RCU 回调执行完毕[McK07e]。此时需要 rcu_barrier()原语做到这一点。请注意，在附录 D.2 和 D.3 中描述的最近的层级 RCU[McK08a]实现也归于“RCU classic”语义中。

最后，RCU 还可以用于提供类型安全的内存[GC96]，见 8.3.2.6 节。在 RCU

的语境中，类型安全的内存保证了给定数据元素在 RCU 读端临界区正在访问它时不会改变类型。想要利用基于 RCU 的类型安全的内存，要将 SLAB_DESTROY_BY_RCU 传递给 kmem_cache_create()。有一点很重要，SLAB_DESTROY_BY_RCU 不会阻止 kmem_cache_alloc()立即重新分配刚被 kmem_cache_free()释放的内存！事实上，由 rcu_dereference 返回的受 SLAB_DESTROY_BY_RCU 标记保护的数据结构可能会释放——重新分配任意次，甚至在 rcu_read_lock()保护下也是如此。与之相反，SLAB_DESTROY_BY_RCU 可以阻止 kmem_cache_free()在 RCU 优雅周期结束之前返回数据结构的完全释放的 slab 给系统。一句话，虽然数据元素可能被释放——重新分配 N 次，但是它的类型是保持不变的。

小问题 8.23：您如何防止无限期阻塞的 synchronize_rcu()调用产生的超大 RCU 读端临界区？

小问题 8.24：synchronize_rcu() API 等待所有已经存在的中断处理函数完成，对吗？

在“RCU BH”一栏，rcu_read_lock_bh()和 rcu_read_unlock_bh()划分出了 RCU 读端临界区，而 call_rcu_bh()在后续的优雅周期结束后调用参数执行的函数。请注意 RCU BH 没有同步的 synchronize_rcu_bh()接口，不过如果有需要，添加一个也很容易。

小问题 8.25：您如果混合使用原语会发生什么？比如您用 rcu_read_lock()和 rcu_read_unlock()划分出 RCU 读端临界区，但是之后又用 call_rcu_bh()来发起 RCU 回调？

小问题 8.26：硬件中断处理函数可以看成是在隐式的 rcu_read_lock_bh()保护之下，对吗？

在“RCU Sched”一栏，所有禁止抢占的 API 都创建了一个 RCU 读端临界区，synchronize_sched()等待对应的 RCU 优雅周期结束。该 RCU API 族是在 Linux 2.6.12 加入内核的，它将旧的 synchronize_kernel() API 分成现在的 synchronize_rcu()（经典 RCU）和 synchronize_sched()（RCU Sched）。请注意 RCU Sched 原本没有异步的 call_rcu_sched()接口，不过在 2.6.26 时添加了一个。本着 Linux 社区极简主义的哲学，所有这些 API 都是按需增加的。

小问题 8.27：如果您混合使用 RCU Classic 和 RCU Sched 会发生什么？

小问题 8.28：总体来说，您不能依靠 synchronize_sched()来等待所有已有的中断处理函数结束，对吗？

“Realtime RCU”这栏有着和 RCU Classic 一样的 API，唯一的区别是 RCU 读端临界区可以被抢占，也可以在获取自旋锁时阻塞。Realtime RCU 的设计请见[McK07a]。

小问题 8.29：为什么 SRCU 和 QRCU 缺少异步的 call_srcu()或者 call_qrcu()接口？

表 8.5：可睡眠 RCU 的等待完成 API

属性	SRCU	QRCU
目的	可睡眠的读者	可睡眠的读者和快速的优雅周期
合入内核版本	2.6.19	
读端原语	srcu_read_lock() srcu_read_unlock()	qrcu_read_lock() qrcu_read_unlock()
更新端原语（同步）	synchronize_srcu()	synchronize_qrcu()
更新端原语（异步/回调）	N/A	N/A
更新端原语（等待回调）	N/A	N/A
类型安全的内存		
读端限制	不可使用 synchronize_srcu()	不可使用 synchronize_qrcu()
读端开销	简单指令、开关抢占	对共享变量的原子自增和自减
异步更新端开销	N/A	N/A
优雅周期延迟	几十毫秒	无读者时几十纳秒
Non-preempt-RT 实现	SRCU	N/A
Preempt-RT 实现	SRCU	N/A

表 8.5 中的“SRCU”栏列出了一种专门的 RCU API，允许在 RCU 读端临界区中睡眠（细节详见附录 D.1）。当然，在 SRCU 读端临界区中使用 synchronize_srcu()会导致自我死锁，所以应该被避免。SRCU 与之前提到的 RCU 实现有一点不同，调用者为每个 SRCU 分配一个 srcu_struct。这种做法是为了防止 SRCU 读端临界区阻塞在其他不相关的 synchronize_srcu()调用上。另外，在这种 RCU 的变体中，srcu_read_lock()返回的值必须传给对应的 srcu_read_unlock()。

“QRCU”栏列出了和 SRCU 有着一样 API 的 RCU 实现，但是专门针对没有读者时的极短优雅周期做了优化，详见[McK07f]。和 SRCU 一样，在 QRCU 读端临界区使用 synchronize_qrcu()会导致自我死锁，所以应该避免。虽然 QRCU 现在还没有被 Linux 内核接收，但是值得一提的是，QRCU 是唯一一种敢宣称拥有低于毫秒级的优雅周期延迟的内核级 RCU 实现。

小问题 8.30：在哪种情况下可以在 SRCU 读端临界区中安全的使用 synchronize_srcu()？

Linux 内核现在拥有的各种 RCU 实现数目让人惊叹。要减少这个数据还是有些希望的，证据在于最近的 Linux 内核最多只有三种 RCU 实现、四个 API（因为 RCU Classic 和 Realtime RCU 共享同样的 API）。不过，和减少锁的 API 一样，进一步的减少 RCU 实现的 API 还需要小心的检查和分析。

不同的 RCU API 可以根据它们的 RCU 读端临界区必须提供的 forward-progress 保证和它们的范围来区分，如下：

1. RCU BH：读端临界区必须保证除了 NMI 和 IRQ 处理函数以外的 forward progress，但不包括软中断。RCU BH 是全局的。
2. RCU Sched：读端临界区必须保证除了 NMI 和 IRQ 处理函数以外的 forward progress，包括软中断。RCU Sched 是全局的。
3. RCU（classic 和 realtime）：读端临界区必须保证除了 NMI、IRQ、软中断处理函数和高优先级实时任务（在 realtime RCU 中）以外的 forward progress，包括软中断。RCU 是全局的。
4. SRCU 和 QRCU：读端临界区不保证 forward progress，除非某些任务在等待对应的优雅周期完成，此时这些读端临界区应该在不超过数秒内完成（也可能更快[2]）。SRCU 和 QRCU 的范围由各自对应的 srcu_struct 或者 qrcu_struct 定义。

换句话说，SRCU 和 QRCU 以允许开发者限定它们的范围来弥补它们非常弱的 forward-progress 保证。

8.3.3.2. RCU 拥有发布——订阅和版本维护 API

幸运的是，下表列出的 RCU 发布——订阅和版本维护原语适用于之前讨论的所有 RCU 变体。这种共性让我们在某些情况下可以复用很多代码，这一点限制了本来会出现的 API 数目的增长。RCU 发布——订阅 API 的本意是消除其他 API 中的内存屏障，这样在无需精通 Linux 支持的超过 20 种 CPU 架构各自的内存顺序模型的情况下[Spr01]，Linux 内核开发者可以轻松使用 RCU。

表 8.6：RCU 的发布——订阅和版本维护 API

类型	原语	合入内核版本	开销
遍历链表	List_for_each_entry_rcu()	2.5.59	简单指令（alpha 上还有内存屏障）

[2] 感谢 James Bottomley 劝说我采用这种构想，而不是简单地说不提供任何 forward progress 保证。

更新链表	List_add_rcu()	2.5.44	内存屏障
	List_add_tail_rcu()	2.5.44	内存屏障
	List_del_rcu()	2.5.44	简单指令
	List_replace_rcu()	2.6.9	内存屏障
	List_splice_init_rcu()	2.6.21	优雅周期延迟
遍历哈希链表	Hlist_for_each_entry_rcu()	2.6.8	简单指令（alpha 上还有内存屏障）
	Hlist_add_after_rcu()	2.6.14	内存屏障
	Hlist_add_before_rcu()	2.6.14	内存屏障
	Hlist_add_head_rcu()	2.5.64	内存屏障
	Hlist_del_rcu()	2.5.64	简单指令
	Hlist_replace_rcu()	2.6.15	内存屏障
遍历指针	Rcu_dereference()	2.6.9	简单指令（alpha 上还有内存屏障）
更新指针	Rcu_assign_pointer()	2.6.10	内存屏障

表中的第一类 API 作用于 Linux 的 struct list_head 循环双链表。list_for_each_entry_rcu()原语以类型安全的方式遍历受 RCU 保护的链表，同时加强在遍历链表时并发插入新数据元素情况下的内存顺序。在非 Alpha 的平台上，该原语相较于 list_for_each_entry()原语不产生或者只带来极低的性能惩罚。list_add_rcu()、list_add_tail_rcu()和 list_replace_rcu()原语都是对非 RCU 版本的模拟，但是在弱顺序的机器上回带来额外的内存屏障开销。list_del_rcu()原语同样是非 RCU 版本的模拟，但是奇怪的是它要比非 RCU 版本快一点点，这是由于 list_del_rcu()只污染 prev 指针，而 list_del()会同时污染 prev 和 next 指针。最后，list_splice_init_rcu()原语和它的非 RCU 版本类似，但是会带来整个优雅周期的延迟。这个优雅周期的目的是让 RCU 读者在完全与链表头部脱离联系之前结束它们对源链表的遍历——如果做不到这一点，RCU 读者甚至会结束它们的遍历。

小问题 8.31：为什么 list_del_rcu()没有同时污染 next 和 prev 指针？

表中的第二类 API 直接作用于 Linux 的 struct hlist_head 线性散列表。Struct hlist_head 比 struct list_head 高级一点的地方是前者只需要一个单指针的链表头部，这在大型哈希表中将节省大量内存。表中的 struct hlist_head 原语与非 RCU 版本的关系同 struct list_head 原语的类似关系一样。

表中的最后一类 API 直接作用于指针，这对创建受 RCU 保护的非链表数据元素是非有用，比如受 RCU 保护的数组和树。Rcu_assign_pointer()原语确保在

弱序机器上，任何在给指针赋值之前进行的初始化都将按照顺序执行。同样地，rcu_dereferece()原语确保后续指针解引用的代码可以在 Alpha CPU 上看见对应的 rcu_assign_pointer()之前进行的初始化的结果。在非 Alpha CPU 上，rcu_dereference()记录哪些被解引用的指针是受 RCU 保护的。

小问题 8.32：通常来说，任何属于 rcu_dereference()的指针必须一直用 rcu_assign_pointer()更新。这个规则有例外吗？

小问题 8.33：这些遍历和更新原语可以在所有 RCU API 家族成员中使用，这有什么负面影响吗？

8.3.3.3. RCU 的 API 可以用在何处？

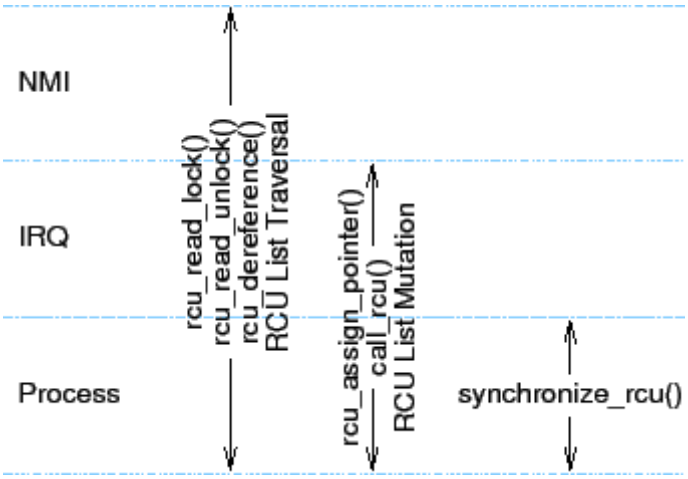


图 8.26：RCU API 使用限制

图 8.26 显示了哪些 RCU API 可以用于哪些内核环境。RCU 读端原语可以用于任何环境，包括 NMI；RCU mutation 和异步优雅周期原语可以用于除了 NMI 以外的任何环境；RCU 同步优雅周期原语只能用于进程上下文。RCU 的遍历链表原语包括 list_for_each_entry_rcu()、hlist_for_each_entry_rcu()等等。同样地，RCU 的 list-mutation 原语包括 list_add_rcu()、hlist_del_rcu()等等。

请注意其它 RCU API 族的原语是可以替换的，比如 srcu_read_lock()也可以用在 rcu_read_lock()能用的上下文。

8.3.3.4. 那么，RCU 究竟是什么？

RCU 的核心不过是一种支持对插入操作的发布和订阅、等待所有 RCU 读者完成、维护多个版本的 API。也就是说，完全可以在 RCU 之上构造抽象级别更高的模型，比如读写锁、引用计数、存在担保等在前面列出的模型。更进一步，我对 Linux 社区会继续为 RCU 寻找新用法丝毫不感到怀疑，当然其他的同步原

语肯定也是这样。

当然，对 RCU 更复杂的看法还包括所有您拿这些 API 能做的所有事。

但是，对很多人来说，想要完整的观察 RCU，就需要一个 RCU 的例子实现。因此下一节将给出一系列复杂性和能力都不断增加的玩具式 RCU 实现。

8.3.4. “玩具式”的 RCU 实现

本节设计的“玩具式”RCU 实现不是为了强调高性能、实用性或者用于什么产品上，而主要是为了强调概念的清晰度。不过，您需要对第 1、2、3、4 和 8 章有一个通透的理解，然后才能说可以轻松理解下面这些玩具实现。

本节以解决存在担保问题的视角，提出了一系列复杂度递增的 RCU 实现。第 8.3.4.1 节给出了基于简单锁的基础 RCU 实现，而从第 8.3.4.3 到底 8.3.4.9 节给出了一组基于锁、引用计数、free-running 计数的简单 RCU 实现。最后，第 8.3.4.10 节对本节进行总结并列出理想的 RCU 实现应该具有的特性。

8.3.4.1. 基于锁的 RCU

也许最简单的RCU实现就是用锁了，如图8.27所示（rcu_lock.h和rcu_lock.c）。在本节的实现中，rcu_read_lock()获取一把全局自旋锁，rcu_read_unlock()释放锁，而 synchronize_rcu()获取自旋锁然后再释放。

```
1  static  void  rcu_read_lock(void)
2  {
3     spin_lock(&rcu_gp_lock);
4  }
5
6  static  void  rcu_read_unlock(void)
7  {
8     spin_unlock(&rcu_gp_lock);
9  }
10
11  void  synchronize_rcu(void)
12  {
13     spin_lock(&rcu_gp_lock);
14     spin_unlock(&rcu_gp_lock);
15  }
```

图 8.27：基于锁的 RCU 实现

因为 synchronize_rcu()只有在获取锁（然后释放）以后才会返回，所以在所有之前发生的 RCU 读端临界区完成前，synchronize_rcu()是不会返回的，因此这符合 RCU 的语义。当然，一个读端临界区同时只能有一个 RCU 读者进入，这基本上可以说是和 RCU 的目的相反了。另外，rcu_read_lock()和 rcu_read_unlock()中的锁操作开销是极大的，读端的开销从 Power5 单核 CPU 上的 100 纳秒到 64 核系统上的 17 毫秒不等。更糟的是，使用同一把锁使得 rcu_read_lock()会进入死锁循环。此外，因为没有用递归锁，所以 RCU 读端临界区不能嵌套。最后一点，原则上并发的 RCU 更新操作可以共享一个公共的优雅周期，但是该实现将优雅周期串行化了，因此无法共享优雅周期。

小问题 8.34：为什么图 8.27 的 RCU 实现里的死锁情景不会出现其他 RCU 实现中？

小问题 8.35：为什么图 8.27 的 RCU 实现不直接用读写锁？这样 RCU 读者就可以处理并发了。

很难想象这种实现能用在任何一个产品中，但是这种实现有一点好处：可以用在几乎所有的用户态程序上。不仅如此，类似的使用每 CPU 锁或者读写锁的实现还曾经用于 Linux 2.4 内核中。

下一节将介绍每 CPU 锁方法的修改版：每线程锁实现。

8.3.4.2. 基于每线程锁的 RCU

图 8.28（rcu_lock_percpu.h 和 rcu_lock_percpu.c）显示了一种基于每线程锁的实现。rcu_read_lock()和 rcu_read_unlock()分别获取和释放当前线程的锁。synchronize_rcu()函数按照次序逐一获取和释放每个线程的锁。这样，所有在 synchronize_rcu()开始时就已经执行的 RCU 读端临界区，必须在 synchronize_rcu()结束前返回。

```
1  static  void  rcu_read_lock(void)
2  {
3     spin_lock(&__get_thread_var(rcu_gp_lock));
4  }
5
6  static  void  rcu_read_unlock(void)
7  {
8     spin_unlock(&__get_thread_var(rcu_gp_lock));
9  }
```

```
10
11  void  synchronize_rcu(void)
12  {
13     int  t;
14
15     for_each_running_thread(t)  {
16        spin_lock(&per_thread(rcu_gp_lock,  t));
17        spin_unlock(&per_thread(rcu_gp_lock,  t));
18     }
19  }
```

图8.28：基于锁的每线程RCU实现

本节实现的优点在于允许并发的 RCU 读者，同时避免了使用单个全局锁可能造成的死锁。不仅如此，读端开销虽然高达大概 140 纳秒，但是不管 CPU 数目为多少，始终保持在 140 纳秒。不过，更新端的开销则在从 Power5 单核上的 600 纳秒到 64 核系统上的超过 100 毫秒不等。

小问题 8.36：如果在图 8.28 的第 15-18 行里，先获取所有锁，然后再释放所有锁，这样是不是更清晰一点呢？毕竟如果这样的话，在没有读者的时刻里代码流程会简化很多。

小问题 8.37：图 8.28 中的实现能够避免死锁吗？如果能，为什么能？如果不能，为什么不能？

小问题 8.38：假如图 8.28 中的 RCU 算法只使用广泛应用的原语，比如 POSIX 线程，会不会更好呢？

本方法在某些情况下是很有效的，尤其是类似的方法曾在 Linux 2.4 内核中使用[MM00]。

之后要提到的基于计数器的 RCU 实现克服了基于锁实现的某些缺点。

8.3.4.3. 基于计数器的简单 RCU 实现

```
1  atomic_t  rcu_refcnt;
2
3  static  void  rcu_read_lock(void)
4  {
5     atomic_inc(&rcu_refcnt);
6     smp_mb();
7  }
```

```
8
9  static  void  rcu_read_unlock(void)
10 {
11    smp_mb();
12    atomic_dec(&rcu_refcnt);
13 }
14
15 void  synchronize_rcu(void)
16 {
17    smp_mb();
18    while  (atomic_read(&rcu_refcnt)  !=  0)  {
19       poll(NULL,  0,  10);
20    }
21    smp_mb();
22 }
```

图8.29：使用单个全局引用计数器的RCU 实现

图 8.29 是一种稍微复杂一点的 RCU 实现（rcu_rcg.h 和 rcu_rcg.c）。本方法在第 1 行定义了一个全局引用计数器 rcu_refcnt。rcu_read_lock()原语自动增加计数，然后执行一个内存屏障，确保在原子自增之后才进入 RCU 读端临界区。同样地，rcu_read_unlock()先执行一个内存屏障，划定 RCU 读端临界区的结束点，然后再原子自减计数器。synchronize_rcu()原语不停自旋，等待引用计数器的值变为 0，语句前后用内存屏障保护正确的顺序。第 19 行的 poll()只是纯粹的延时，从纯 RCU 语义的角度上看是可以省略的。等 synchronize_rcu()返回后，所有之前发生的 RCU 读端临界区都已经完成了。

通过和 8.3.4.1 节中基于锁的实现相比，我们欣喜地发现本节这种实现可以让读者并发进入 RCU 读端临界区。和 8.3.4.2 节中基于每线程锁的实现相比，我们又欣喜地发现本节的实现可以让 RCU 读端临界区嵌套。另外，rcu_read_lock()原语不会进入死锁循环，因为它既不自旋也不阻塞。

小问题 8.39：但是如果您在调用 synchronize_rcu()时持有一把锁，然后又在 RCU 读端临界区中获取同一把锁，会发生什么呢？

但是，这个实现还是存在一些严重的缺点。首先，rcu_read_lock()和 rcu_read_unlock()中的原子操作开销是非常大的，读端开销从 Power5 单核 CPU 上的 100 纳秒到 64 核系统上的 40 毫秒不等。这意味着 RCU 读端临界区必须非常得长，才能够满足真实的读端并发请求。但是在另一方面，当没有读者时，优雅周期只有差不多 40 纳秒，这比 Linux 内核中的产品级实现要快上很多个数量

级。

小问题 8.40：假如 synchronize_rcu()包含了一个 10 毫秒的延时，优雅周期怎么可能只要 40 纳秒呢？

其次，如果存在多个并发的 rcu_read_lock()和 rcu_read_unlock()操作，对 rcu_refcnt 的内存访问竞争将会十分激烈（luyang:还记得之前提过的缓存线乒乓么？）。以上这两个缺点极大地影响 RCU 的目标，即提供一种读端低开销的同步原语。

最后，在很长的读端临界区中的大量 RCU 读者甚至会让 synchronize_rcu()无法完成，因为全局计数器可能永远不为 0。这会导致 RCU 更新端的饥饿，这一点在产品级应用里肯定是不可接受的。

小问题 8.41：在图 8.29 的实现里，为什么当 synchronize_rcu()等待时间过长了以后，不能简单地让 rcu_read_lock()暂停一会儿呢？这种做法不能防止 synchronize_rcu()饥饿吗？

通过上述内容，很难想象本节的实现可以在产品级应用中使用，虽然它比基于锁的实现更有这方面的潜力，比如，作为一种高负荷调试环境中的 RCU 实现。下一节我们将介绍一种对写者更有利的引用计数 RCU 变体。

8.3.4.4. 不会让更新者饥饿的引用计数 RCU

图 8.31（rcu_rcgp.h）展示了一种 RCU 实现的读端原语，使用一对引用计数器（rcu_refcnt[]），通过一个全局索引（rcu_idx）从这对计数器中选出一个计数器，一个每线程的嵌套计数器 rcu_nesting，一个每线程的全局索引快照（rcu_read_idx），以及一个全局锁（rcu_gp_lock），图 8.30 给出了上述定义。

```
1  DEFINE_SPINLOCK(rcu_gp_lock);
2  atomic_t  rcu_refcnt[2];
3  atomic_t  rcu_idx;
4  DEFINE_PER_THREAD(int,  rcu_nesting);
5  DEFINE_PER_THREAD(int,  rcu_read_idx);
```

图 8.30：RCU 全局引用计数对的数据定义

rcu_read_lock()原语自动增加由 rcu_idx 标出的 rcu_refcnt[]成员的值，然后将索引保存在每线程变量 rcu_read_idx 中。rcu_read_unlock()原语自动减少对应的 rcu_read_lock()增加的那个计数器的值。不过，因为 rcu_idx 每个线程只能设置为 rcu_idx 设置一个值，所以还需要一些手段才能允许嵌套。方法是用每线程的 rcu_nesting 变量跟踪嵌套。

```
1  static  void  rcu_read_lock(void)
```

```
2  {
3     int  i;
4     int  n;
5
6     n  =  __get_thread_var(rcu_nesting);
7     if  (n  ==  0)  {
8        i  =  atomic_read(&rcu_idx);
9        __get_thread_var(rcu_read_idx)  =  i;
10        atomic_inc(&rcu_refcnt[i]);
11    }
12    __get_thread_var(rcu_nesting)  =  n  +  1;
13    smp_mb();
14 }
15
16 static  void  rcu_read_unlock(void)
17 {
18    int  i;
19    int  n;
20
21    smp_mb();
22    n  =  __get_thread_var(rcu_nesting);
23    if  (n  ==  1)  {
24 i  =  __get_thread_var(rcu_read_idx);
25       atomic_dec(&rcu_refcnt[i]);
26    }
27    __get_thread_var(rcu_nesting)  =  n  -  1;
28 }
```

图8.31：使用全局引用计数对的RCU 读端原语

为了让这种方法能够工作，图 8.31 rcu_read_lock()函数的第 6 行获取了当前线程的 rcu_nesting，如果第 7 行的检查发现当前处于最外层的 rcu_read_lock()，那么第 8-10 行获取变量 rcu_idx 的当前值，将其存到当前线程的 rcu_read_idx 中，然后增加被 rcu_idx 选中的 rcu_refcnt 元素的值。第 12 行不管现在的 rcu_nesting 值是多少，直接对其加 1。第 13 行执行一个内存屏障，确保 RCU 读端临界区不会在 rcu_read_lock()之前开始。

同样地，rcu_read_unlock()函数在第 21 行也执行一个内存屏障，确保 RCU

读端临界区不会在 rcu_read_unlock()代码之后还没结束。第 22 行获取当前线程的 rcu_nesting，如果第 23 行的检查发现当前处于最外层的 rcu_read_unlock()，那么第 24-25 行获取当前线程的 rcu_read_idx（由最外层的 rcu_read_lock()保存）并且自动减少被 rcu_read_idx 选择的 rcu_refcnt 元素。无论当前嵌套了多少层，第 27 行都直接减少本线程的 rcu_nesting 值。

```
1  void  synchronize_rcu(void)
2  {
3     int  i;
4
5     smp_mb();
6     spin_lock(&rcu_gp_lock);
7     i  =  atomic_read(&rcu_idx);
8     atomic_set(&rcu_idx,  !i);
9     smp_mb();
10     while  (atomic_read(&rcu_refcnt[i])  !=  0)  {
11        poll(NULL,  0,  10);
12     }
13     smp_mb();
14     atomic_set(&rcu_idx,  i);
15     smp_mb();
16     while  (atomic_read(&rcu_refcnt[!i])  !=  0)  {
17        poll(NULL,  0,  10);
18     }
19     spin_unlock(&rcu_gp_lock);
20     smp_mb();
21 }
```

图 8.32：使用全局引用计数对的 RCU 更新端原语

图 8.32（rcu_rcpg.c）实现了对应的 synchronize_rcu()实现。第 6 行和第 19 行获取并释放 rcu_gp_lock，因为这样可以防止多于一个的并发 synchronize_rcu()实例。第 7-8 行分别获取 rcu_idx 的值和对其取反，这样后续的 rcu_read_lock()实例将使用和之前的实例不同的 rcu_idx 值。然后第 10-12 行等待之前的由 rcu_idx 选出的元素变成 0，第 9 行的内存屏障是为了保证对 rcu_idx 的检查不会被优化到对 rcu_idx 取反操作之前。第 13-18 行重复这一过程，第 20 行的内存屏障是为了保证所有后续的回收操作不会被优化到对 rcu_refcnt 的检查之前执行。

小问题 8.42：为什么图 8.32 中 synchronize_rcu()第 5 行的内存屏障后面又跟

了一个获取自旋锁操作？

小问题 8.43：为什么图 8.32 的计数器要检查两次？难道检查一次还不够吗？

本节的实现避免了图 8.29 的简单计数器实现可能发生的更新端饥饿问题。

不过这种实现仍然存在一些严重确定。首先，rcu_read_lock()和 rcu_read_unlock()中的原子操作开销很大。事实上，它们比图 8.29 中的单计数器要复杂很多，读端原语的开销从 Power5 单核处理器上的 150 纳秒到 64 核处理器上的 40 毫秒不等。更新端 synchronize_rcu()原语的开销也变大了，从 Power5 单核 CPU 中的 200 纳秒到 64 核处理器中的 40 毫秒不等。这意味着 RCU 读端临界区必须非常得长，才能够满足真实的读端并发请求。

其次，如果存在很多并发的 rcu_read_lock()和 rcu_read_unlock()操作，那么对 rcu_refcnt 的内存访问竞争将会十分激烈，这将导致耗费巨大的 cache miss。这一点进一步延长了提供并发读端访问所需要的 RCU 读端临界区持续时间。这两个缺点在很多情况下都影响了 RCU 的目标。

第三，需要检查 rcu_idx 两次这一点为更新操作增加了开销，尤其是线程数目很多时。

最后，尽管原则上并发的 RCU 更新可以共用一个公共优雅周期，但是本节的实现串行化了优雅周期，使得这种共享无法进行。

小问题 8.44：既然原子自增和原子自减的开销巨大，为什么不在图 8.31 的第 10 行使用非原子自增，在第 25 行使用非原子自减呢？

尽管有这样那样的缺点，这种 RCU 的变体还是可以运用在小型的多核系统上，也许可以作为一种 memory-conserving 实现，用于维护与更复杂实现之间的 API 兼容性。但是，这种方法在 CPU 增多时可扩展性不佳。

下一节介绍了另一种基于引用计数机制的 RCU 变体，该方法极大地改善了读端性能和可扩展性。

8.3.4.5. 可扩展的基于计数器 RCU 实现

```
1  DEFINE_SPINLOCK(rcu_gp_lock);
2  DEFINE_PER_THREAD(int  [2],  rcu_refcnt);
3  atomic_t  rcu_idx;
4  DEFINE_PER_THREAD(int,  rcu_nesting);
5  DEFINE_PER_THREAD(int,  rcu_read_idx);
```

图 8.33：RCU 每线程引用计数对的数据定义

```
1  static  void  rcu_read_lock(void)
```

```
2  {
3    int  i;
4    int  n;
5
6    n  =  __get_thread_var(rcu_nesting);
7    if  (n  ==  0)  {
8       i  =  atomic_read(&rcu_idx);
9       __get_thread_var(rcu_read_idx)  =  i;
10       __get_thread_var(rcu_refcnt)[i]++;
11    }
12    __get_thread_var(rcu_nesting)  =  n  +  1;
13    smp_mb();
14 }
15
16 static  void  rcu_read_unlock(void)
17 {
18    int  i;
19    int  n;
20
21    smp_mb();
22    n  =  __get_thread_var(rcu_nesting);
23    if  (n  ==  1)  {
24       i  =  __get_thread_var(rcu_read_idx);
25       __get_thread_var(rcu_refcnt)[i]--;
26    }
27    __get_thread_var(rcu_nesting)  =  n  -  1;
28 }
```

图8.34：使用每线程引用计数对的RCU 读端原语

图 8.34（rcu_rcpl.h）是一种使用每线程引用计数的 RCU 实现的读端原语。本实现与图 8.31 中的实现十分类似，唯一的区别在于 rcu_refcnt 成了一个每线程变量（见图 8.33），所以 rcu_read_lock()和 rcu_read_unlock()原语不用再执行原子操作。

小问题 8.45: 别忽悠了!我在 rcu_read_lock()里看见 atomic_read()原语了!!!为什么你想假装 rcu_read_lock()里没有原子操作？？？

```
1  static  void  flip_counter_and_wait(int  i)
```

```
2  {
3    int  t;
4
5    atomic_set(&rcu_idx,  !i);
6    smp_mb();
7    for_each_thread(t)  {
8       while  (per_thread(rcu_refcnt,  t)[i]  !=  0)
{
9          poll(NULL,  0,  10);
10       }
11    }
12    smp_mb();
13 }
14
15 void  synchronize_rcu(void)
16 {
17    int  i;
18
19    smp_mb();
20    spin_lock(&rcu_gp_lock);
21    i  =  atomic_read(&rcu_idx);
22    flip_counter_and_wait(i);
23    flip_counter_and_wait(!i);
24    spin_unlock(&rcu_gp_lock);
25    smp_mb();
26 }
```

图8.35：使用每线程引用计数对的RCU 更新端原语

图 8.35（rcu_rcpl.c）是 synchronize_rcu()的实现，还有一个辅助函数 flip_counter_and_wait()。synchronize_rcu()函数和图 8.32 中的基本一样，除了原来的重复检查计数器过程被替换成了第 22-23 行的辅助函数。

新的 flip_counter_and_wait()函数在第 5 行更新 rcu_idx 变量，第 6 行执行内存屏障，然后第 7-11 行循环检查每个线程对应的 rcu_refcnt 元素，等待该值变为 0。一旦所有元素都变为 0，第 12 行执行另一个内存屏障，然后返回。

本 RCU 实现对软件环境有所要求，（1）能够声明每线程变量，（2）每个线程都可以访问其他线程的每线程变量，（3）能够遍历所有线程。绝大多数软件环

境都满足上述要求，但是通常对线程数的上限有所限制。更复杂的实现可以避开这种限制，比如，使用可扩展的哈希表。这种实现能够动态地跟踪线程，比如，在线程第一次调用 rcu_read_lock()时将线程加入哈希表。

小问题 8.46：好极了，如果我有 N 个线程，那么我要等待 2N 个 10 毫秒（每个 flip_counter_and_wait()调用消耗的时间，假设我们每个线程只等待一次）。我们难道不能让优雅周期再块一点完成吗？

不过本实现还有一些缺点。首先，需要检查 rcu_idx 两次，这为更新端带来一些开销，特别是线程数很多时。

其次，synchronize_rcu()必须检查的变量数随着线程增多而线性增长，这给线程数很多的应用程序带来的一定开销。

第三，和之前一样，虽然原则上并发的 RCU 更新可以共用一个公共优雅周期，但是本节的实现串行化了优雅周期，使得这种共享无法进行。

最后，本节曾经提到的软件环境需求，在某些环境下每线程变量和遍历线程可能存在问题。

读端原语的扩展性非常好，不管是在单核系统还是 64 核系统都只需要 115 纳秒左右。Synchronize_rcu()原语的扩展性不佳，开销在单核 Power5 系统上的 1 毫秒到 64 核系统上的 200 毫秒不等。总体来说，本节的方法可以算是一种基础的产品级用户态 RCU 实现了。

下一节将介绍一种能够让并发的 RCU 更新更有效的算法。

8.3.4.6. 可扩展的基于计数器 RCU 实现，可以共享优雅周期

```
1  DEFINE_SPINLOCK(rcu_gp_lock);
2  DEFINE_PER_THREAD(int  [2],  rcu_refcnt);
3  long  rcu_idx;
4  DEFINE_PER_THREAD(int,  rcu_nesting);
5  DEFINE_PER_THREAD(int,  rcu_read_idx);
```

图 8.36：使用每线程引用计数对和共享更新数据的数据定义

和前几节一样，图 8.37（rcu_rcpls.h）是一种使用每线程引用计数 RCU 实现的读端原语，但是该实现允许更新端共享优雅周期。本节的实现和图 8.34 中的实现唯一的区别是，rcu_dix 现在是一个 long 型整数，自由的计数（？？？），所以图 8.37 第 8 行用了一个掩码屏蔽了最低位。我们还将 atomic_read()和 atomic_set()改成了 ACCESS_ONCE()。图 8.36 中的数据定义和图 8.33 也很相似，只是 rcu_idx 现在是 long 类型而非之前的 atomic_t 类型。

```
1  static  void  rcu_read_lock(void)
```

```
2  {
3    int  i;
4    int  n;
5
6    n  =  __get_thread_var(rcu_nesting);
7    if  (n  ==  0)  {
8       i  =  ACCESS_ONCE(rcu_idx)  &  0x1;
9       __get_thread_var(rcu_read_idx)  =  i;
10       __get_thread_var(rcu_refcnt)[i]++;
11    }
12    __get_thread_var(rcu_nesting)  =  n  +  1;
13    smp_mb();
14 }
15
16 static  void  rcu_read_unlock(void)
17 {
18    int  i;
19    int  n;
20
21    smp_mb();
22    n  =  __get_thread_var(rcu_nesting);
23    if  (n  ==  1)  {
24       i  =  __get_thread_var(rcu_read_idx);
25       __get_thread_var(rcu_refcnt)[i]--;
26    }
27    __get_thread_var(rcu_nesting)  =  n  -  1;
28 }
```

图8.37：使用每线程引用计数对和共享更新数据的RCU读端原语

```
1  static  void  flip_counter_and_wait(int  ctr)
2  {
3    int  i;
4    int  t;
5
6    ACCESS_ONCE(rcu_idx)  =  ctr  +  1;
```

```
7     i = ctr & 0x1;
8     smp_mb();
9     for_each_thread(t) {
10        while (per_thread(rcu_refcnt, t)[i] != 0)
          {
11            poll(NULL, 0, 10);
12        }
13    }
14    smp_mb();
15 }
16
17 void synchronize_rcu(void)
18 {
19    int ctr;
20    int oldctr;
21
22    smp_mb();
23    oldctr = ACCESS_ONCE(rcu_idx);
24    smp_mb();
25    spin_lock(&rcu_gp_lock);
26    ctr = ACCESS_ONCE(rcu_idx);
27    if (ctr - oldctr >= 3) {
28       spin_unlock(&rcu_gp_lock);
29       smp_mb();
30       return;
31    }
32    flip_counter_and_wait(ctr);
33    if (ctr - oldctr < 2)
34       flip_counter_and_wait(ctr + 1);
35    spin_unlock(&rcu_gp_lock);
36    smp_mb();
37 }
```

图8.38：使用每线程引用计数对的RCU共享更新端原语

图8.38（rcu_rcpls.c）是synchronize_rcu()及其辅助函数flip_counter_and_wait()的实现。和图8.35很相像。flip_counter_and_wait()的区别在于：

1. 第 6 行使用 ACCESS_ONCE()代替了 atomic_set()，用自增替代取反。
2. 新增了第 7 行，将计数器的最低位掩去。

synchronize_rcu()的区别要多一些：

1. 新增了一个局部变量 oldctr，存储第 23 行的获取每线程锁之前的 rcu_idx 值。
2. 第 26 行用 ACCESS_ONCE()代替 atomic_read()。
3. 第 27-30 行检查在锁已获取时，其他线程此时是否在循环检查 3 个以上的计数器，如果是，释放锁，执行一个内存屏障然后返回。在本例中，有两个线程在等待计数器变为 0，所以其他的线程已经做了所有必须的工作了。
4. 在第 33-34 行，在锁已被获取时，人啊过当前检查计数器是否为 0 的线程不足 2 个，那么 flip_counter_and_wait()会被调用两次。另一方面，如果有两个线程，另一个线程已经完成了对计数器的检查，那么只需在有一个就可以了。

在本方法中，如果有任意多个线程并发调用 synchronize_rcu()，一个线程对应一个 CPU，那么最多只有 3 个线程在等待计数器变为 0。

尽管有这些改进，本节的 RCU 实现仍然存在一些缺点。首先，和上一节一样，需要检查 rcu_idx 两次为更新端带来了开销，尤其是线程很多时。

其次，本实现需要每 CPU 变量和遍历所有线程的能力，这在某些软件环境可能是有问题的。

最后，在 32 位机器上，可能由于 rcu_idx 溢出而导致更新端线程被长时间抢占。这可能会导致该线程强制执行不必要的检查计数器操作。但是，即使每个优雅周期只耗费一毫秒，被抢占的线程也可能要等待超过一小时，此时额外执行的检查计数器操作就不是你最关心的了。

和 8.3.4.3 节介绍的实现一样，本实现的读端原语扩展性极佳，不管 CPU 数为多少，开销大概为 115 纳秒。synchronize_rcu()原语的开销仍然昂贵，从 1 毫秒到 15 毫秒不等。然而这比 8.3.4.5 节中大概 200 毫秒的开销已经好多了。所以，尽管存在这些缺点，本节的 RCU 实现还是可以在真实世界中的产品中使用了。

小问题 8.47：所有这些玩具式的 RCU 实现都要么在 rcu_read_lock()和 rcu_read_unlock()中使用了原子操作，要么让 synchronize_rcu()的开销与线程数线性增长。那么究竟在哪种环境下，RCU 的实现既可以让上述三个原语的实现简单，又能拥有 O(1)的开销和延迟呢？

重新看看图 8.37，我们看到了对一个全局变量的访问和对不超过 4 个每线程变量的访问。考虑到在 POSIX 线程中访问每线程变量的开销相对较高，我们可以将三个每线程变量放进单个结构体中，让 rcu_read_lock()和 rcu_read_unlock()

用单个每线程变量存储类来访问各自的每线程变量。不过，下一节将会介绍一种更好的办法，可以减少访问每线程变量的次数到一次。

8.3.4.7. 基于自由增长计数器的 RCU

```
1  DEFINE_SPINLOCK(rcu_gp_lock);
2  long  rcu_gp_ctr  =  0;
3  DEFINE_PER_THREAD(long,  rcu_reader_gp);
4  DEFINE_PER_THREAD(long,  rcu_reader_gp_snap);
```

图8.39：使用自由增长计数器的数据定义

```
1  static  void  rcu_read_lock(void)
2  {
3    __get_thread_var(rcu_reader_gp)  =  rcu_gp_ctr  +
1;
4    smp_mb();
5  }
6
7  static  void  rcu_read_unlock(void)
8  {
9    smp_mb();
10    __get_thread_var(rcu_reader_gp)  =  rcu_gp_ctr;
11  }
12
13  void  synchronize_rcu(void)
14  {
15    int  t;
16
17    smp_mb();
18    spin_lock(&rcu_gp_lock);
19    rcu_gp_ctr  +=  2;
20    smp_mb();
21    for_each_thread(t)  {
22      while  ((per_thread(rcu_reader_gp,  t)  &  0x1)
&&
```

```
23 ((per_thread(rcu_reader_gp, t) -
24       rcu_gp_ctr) < 0)) {
25             poll(NULL, 0, 10);
26         }
27     }
28     spin_unlock(&rcu_gp_lock);
29     smp_mb();
30 }
```

图8.40：使用自由增长计数器的RCU实现

图 8.40（rcu.h 和 rcu.c）是一种基于单个全局 free-running 计数器的 RCU 实现，该计数器只对偶数值进行计数，相关的数据定义见图 8.39。rcu_read_lock()的实现极其简单。第 3 行向全局 free-running 变量 rcu_gp_ctr 加 1，将相加后的奇数值存储在每线程变量 rcu_reader_gp 中。第 4 行执行一个内存屏障，防止后续的 RCU 读端临界区内容“泄漏”。

rcu_read_unlock()实现也很类似。第 9 行执行一个内存屏障，防止前一个 RCU 读端临界区“泄漏”。第 10 行将全局变量 rcu_gp_ctr 的值复制给每线程变量 rcu_reader_gp，将此每线程变量的值变为偶数值，这样当前并发的 synchronize_rcu()实例就知道忽略该每线程变量了。

小问题 8.48：如果任何偶数值都可以让 synchronize_rcu()忽略对应的任务，那么图 8.40 的第 10 行为什么不直接给 rcu_reader_gp 赋值为 0？

synchronize_rcu()会等待所有线程的 rcu_reader_gp 变量变为偶数值。但是，因为 synchronize_rcu()只需要等待“在调用 synchronize_rcu()之前就已存在的”RCU 读端临界区，所以完全可以有更好的方法。第 17 行执行一个内存屏障，防止之前操纵的受 RCU 保护的数据结构被乱序（由编译器或者是 CPU）放到第 17 行之后执行。为了防止多个 synchronize_rcu()实例并发执行，第 18 行获取 rcu_gp_lock 锁（第 28 释放锁）。然后第 19 行给全局变量 rcu_gp_ctr 加 2，遮掩该锁有之前已经存在的 RCU 读端临界区对应的每线程变量 rcu_reader_gp 的值就比 rcu_gp_ctr 与机器字长取模后的值小了。回忆一下，rcu_reader_gp 的值为偶数的线程不在 RCU 读端临界区里，所以第 21-27 行扫描 rcu_reader_gp 的值，直到所有值要么是偶数（第 22 行），要么比全局变量 rcu_gp_ctr 的值大（第 23-24 行）。第 25 行阻塞一小段时间，等待一个之前已经存在的 RCU 读端临界区退出，如果对优雅周期的延迟很敏感的话，也可以用自旋锁来代替。最后，第 29 行的内存屏障保证所有后续的销毁工作不会被乱序到循环之前进行。

小问题 8.49：为什么需要图 8.40 中第 17 和第 29 行的内存屏障？难道第 18 行和第 28 行的锁原语自带的内存屏障还不够吗？

本节方法的读端性能非常好，不管 CPU 数目多少，带来的开销大概是 63 纳秒。更新端的开销稍大，从 Power5 单核的 500 纳秒到 64 核的超过 100 毫秒不等。

小问题 8.50：第 8.3.4.6 节的更新端优化不能用于图 8.40 的实现中吗？

本节实现除了刚才提到的更新端的开销较大以外，还有一些严重缺点。首先，该实现不允许 RCU 读端临界区嵌套，这将是下一节要讨论的话题。其次如果读者在图 8.40 第 3 行获取 rcu_gp_ctr 之后，存储到 rcu_reader_gp 之前被抢占，并且如果 rcu_gp_ctr 计数器的值增长到最大值的一半以上，但没有达到最大值时，那么 synchronize_rcu()将会忽略后续的 RCU 读端临界区。第三也是最后一点，本实现需要软件环境支持每线程变量和对所有线程遍历。

小问题 8.51：图 8.40 第 3 行提到的读者被抢占问题是一个真实问题吗？换句话说，这种导致问题的事件序列可能发生吗？如果不能，为什么不能？如果能，事件序列是什么样的，我们该怎样处理这个问题？

8.3.4.8. 基于 Free-Running 计数器的可嵌套 RCU

```
1  DEFINE_SPINLOCK(rcu_gp_lock);
2  #define  RCU_GP_CTR_SHIFT  7
3  #define  RCU_GP_CTR_BOTTOM_BIT  (1  <<
            RCU_GP_CTR_SHIFT)
4  #define  RCU_GP_CTR_NEST_MASK
           (RCU_GP_CTR_BOTTOM_BIT  -  1)
5  long  rcu_gp_ctr  =  0;
6  DEFINE_PER_THREAD(long,  rcu_reader_gp);
```

图 8.41：基于 free-running 计数器的可嵌套 RCU 的数据定义

```
1  static  void  rcu_read_lock(void)
2  {
3     long  tmp;
4     long  *rrgp;
5
6     rrgp  =  &__get_thread_var(rcu_reader_gp);
7     tmp  =  *rrgp;
8     if  ((tmp  &  RCU_GP_CTR_NEST_MASK)  ==  0)
9        tmp  =  rcu_gp_ctr;
10     tmp++;
```

```
11     *rrgp = tmp;
12     smp_mb();
13  }
14
15  static void rcu_read_unlock(void)
16  {
17     long tmp;
18
19     smp_mb();
20     __get_thread_var(rcu_reader_gp)--;
21  }
22
23  void synchronize_rcu(void)
24  {
25     int t;
26
27     smp_mb();
28     spin_lock(&rcu_gp_lock);
29     rcu_gp_ctr += RCU_GP_CTR_BOTTOM_BIT;
30     smp_mb();
31     for_each_thread(t) {
32        while (rcu_gp_ongoing(t) &&
33                ((per_thread(rcu_reader_gp, t) -
34                  rcu_gp_ctr) < 0)) {
35            poll(NULL, 0, 10);
36        }
37     }
38     spin_unlock(&rcu_gp_lock);
39     smp_mb();
40  }
```

图 8.42：使用 free-running 计数器的可嵌套 RCU 实现

图 8.42（rcu_nest.h 和 rcu_nest.c）是一种基于单个全局 free-running 计数器的 RCU 实现，但是允许 RCU 读端临界区的嵌套。这种嵌套能力是通过让全局变量 rcu_gp_ctr 的最低位记录嵌套次数实现的，定义在图 8.41 中。本节的方法是第 8.3.4.7 节方法的通用版本，保留一个最低位来记录嵌套深度。为了做到这一

点，定义了两个宏，RCU_GP_CTR_NEST_MASK 和 RCU_GP_CTR_BOTTOM_BIT。两个宏之间的关系是：RCU_GP_CTR_NEST_MASK=RCU_GP_CTR_BOTTOM_BIT - 1。RCU_GP_CTR_BOTTOM_BIT 宏是用于记录嵌套那一位之前的一位，RCU_GP_CTR_NEST_MASK 红则覆盖 rcu_gp_ctr 中所有用于记录嵌套的位。显然，这两个宏必须保留足够多的位来记录允许的最大 RCU 读端临界区嵌套深度，在本实现中保留了 7 位，因为允许的最大 RCU 读端临界区嵌套深度为 127，这足够绝大多数应用使用了。

rcu_read_lock()的实现仍然十分简单。第 6 行将指向本线程 rcu_reader_gp 实例的指针放入局部变量 rrgp 中，将代价昂贵的访问 phtread 每线程变量 API 的数目降到最低。第 7 行记录 rcu_reader_gp 的值放入另一个局部变量 tmp 中，第 8 行检查低位字节是否为 0，表明当前的 rcu_read_lock()是最外层的。如果是，第 9 行将全局变量 rcu_gp_ctr 的值存入 tmp，因为第 7 行之前存入的值可能已经过期了。如果不是，第 10 行增加嵌套深度，如果您能记得，存放在计数器的最低 7 位。第 11 行将更新后的计数器值重新放入当前线程的 rcu_reader_gp 实例中，然后，也是最后，第 12 行执行一个内存屏障，防止 RCU 读端临界区泄漏到 rcu_read_lock()之前的代码里。

换句话说，除非当前调用的 rcu_read_lock()是检讨在 RCU 读端临界区中，否则本节实现的 rcu_read_lock()原语会获取全局变量 rcu_gp_ctr 的一个副本，而在嵌套环境中，rcu_read_lock()则去获取 rcu_reader_gp 在当前线程中的实例。在两种情况下，rcu_read_lock()都会增加获取到的值，表明嵌套深度又增加了一层，然后将结果储存到当前线程的 rcu_reader_gp 实例中。

有趣的是，rcu_read_unlock()的实现和第 8.3.4.7 节中的实现一模一样。第 19 行执行一个内存屏障，防止 RCU 读端临界区泄漏到 rcu_read_unlock()之后的代码中去，然后第 20 行减少当前线程的 rcu_reader_gp 实例，这将减少 rcu_reader_gp 最低几位包含的嵌套深度。rcu_read_unlock()原语的调试版本将会在减少嵌套深度之前检查 rcu_reader_gp 的最低几位是否为 0。

synchronize_rcu()的实现与第 8.3.4.7 节十分类似。不过存在两点不同。第一，第 29 行将 RCU_GP_CTR_BOTTOM_BIT 加向全局变量 rcu_gp_ctr，而不是直接加常数 2。第二，第 32 行的比较被剥离成一个函数，检查 RCU_GP_CTR_BOTTOM_BIT 指示的位，而非无条件地检查最低位。

本节方法的读端性能与 8.3.4.7 节中的实现几乎一样，不管 CPU 数目多少，开销大概为 65 纳秒。更新端的开销仍然较大，从 Power5 单核的 600 纳秒到 64 核的超过 100 毫秒。

小问题 8.52：为什么不像上一节那样，直接用一个单独的每线程变量来表示

嵌套深度，反而用复杂的位运算来表示？

除了解决了 RCU 读端临界区嵌套问题以外，本节的实现有着和 8.3.4.7 节的实现一样的缺点。另外，在 32 位系统上，本方法会减少全局变量 rcu_gp_ctr 变量溢出所需的时间。下一节将介绍一种能大大延长溢出所需时间的方法，同时又极大地降低了读端开销。

小问题 8.53：对于图 8.42 的算法，您怎样才能将全局变量 rcu_gp_ctr 溢出的时间延长一倍？

小问题 8.54：对于图 8.42 的算法，溢出是致命的吗？为什么呢？为什么不是？如果是致命的，有什么办法可以解决它？

8.3.4.9. 基于静止状态的 RCU

```
1  DEFINE_SPINLOCK(rcu_gp_lock);
2  long  rcu_gp_ctr  =  0;
3  DEFINE_PER_THREAD(long,  rcu_reader_qs_gp);
```

图 8.43：基于 quiescent-state 的 RCU 的数据定义

```
1  static  void  rcu_read_lock(void)
2  {
3  }
4
5  static  void  rcu_read_unlock(void)
6  {
7  }
8
9  rcu_quiescent_state(void)
10 {
11    smp_mb();
12    __get_thread_var(rcu_reader_qs_gp)  =
13        ACCESS_ONCE(rcu_gp_ctr)  +  1;
14    smp_mb();
15 }
16
17 static  void  rcu_thread_offline(void)
18 {
```

```
19     smp_mb();
20     __get_thread_var(rcu_reader_qs_gp)  =
21         ACCESS_ONCE(rcu_gp_ctr);
22     smp_mb();
23  }
24
25  static  void  rcu_thread_online(void)
26  {
27     rcu_quiescent_state();
28  }
```

图8.44：基于quiescent-state的RCU读端原语

图 8.44（rcu_qs.h）是一种基于静止状态的用户态级 RCU 实现的读端原语，数据定义在图 8.43。从图中第 1-7 行可以看出，rcu_read_lock()和 rcu_read_unlock()原语不做任何事情，就和在 Linux 内核里一样，这种空函数会成为内联函数，然后被编译器优化掉。之所以是空函数，是因为基于静止状态的 RCU 实现用之前提到的静止状态来估计 RCU 读端临界区的长度，这种状态包括图 8.44 第 9-15 行的 rcu_quiescent_state()调用。进入延长过的静止状态（比如当发生阻塞时）的线程可以分别用 thread_offline()和 thread_online() API 来标记延长的静止状态的开始和结尾。这样，thread_online()就成了对 rcu_read_lock()的模仿，thread_offline()就成了对

rcu_read_unlock()的模仿。图 8.44 中第 17-28 行是这两个函数的实现。在 RCU 读端临界区里出现静止状态是不被允许的。

在 rcu_quiescent_state()中，第 11 行执行一个内存屏障，防止在静止状态之前的代码乱序到静止状态之后执行。第 12-13 行获取全局变量 rcu_gp_ctr 的副本，使用 ACCESS_ONCE()来保证编译器不会启用任何优化措施让 rcu_gp_ctr 被读取超过一次。然后对取来的值加 1，储存到每线程变量 rcu_reader_qs_gp 中，这样任何并发的 synchronize_rcu()实例都只会看见奇数值，因此就知道新的 RCU 读端临界区开始了。正在等待老的读端临界区的 synchronize_rcu()实例因此也知道忽略新产生的读端临界区。最后，第 14 行执行一个内存屏障。

小问题 8.55：图 8.44 中第 14 行中多余的内存屏障会不会显著增加 rcu_quiescent_state()的开销？

有些应用程序可能只是偶尔需要用 RCU，但是一旦它们开始用，那一定是到处都在用。这种应用程序可以在开始用 RCU 时调用 rcu_thread_online()，在不再使用 RCU 时调用 rcu_thread_offline()。在调用 rcu_thread_offline()和下一个调用 rcu_thread_online()之间的时间被成为延长的静止状态，在这段时间 RCU 不会

显式地注册静止状态。

rcu_thread_offline()函数直接将每线程变量 rcu_reader_qs_gp 赋值为 rcu_gp_ctr 的当前值，该值是一个偶数。这样所有并发的 synchronize_rcu()实例就知道忽略这个线程。

小问题 8.56：为什么需要图 8.44 第 19 行和第 22 行的内存屏障？

rcu_thread_online()函数直接调用 rcu_quiescent_state()，这也表示延长静止状态的结束。

```
1  void  synchronize_rcu(void)
2  {
3     int  t;
4
5     smp_mb();
6     spin_lock(&rcu_gp_lock);
7     rcu_gp_ctr  +=  2;
8     smp_mb();
9     for_each_thread(t)  {
10        while  (rcu_gp_ongoing(t)  &&
11 ((per_thread(rcu_reader_qs_gp,  t)  -
12 rcu_gp_ctr)  <  0))  {
13 poll(NULL,  0,  10);
14        }
15    }
16    spin_unlock(&rcu_gp_lock);
17    smp_mb();
18 }
```

图 8.45：基于 quiescent-state 的 RCU 更新端原语

图 8.45（rcu_qs.c）是 synchronize_rcu()的实现，和上一节的实现很相像。

本节实现的读端原语块得惊人，调用 rcu_read_lock()和 rcu_read_unlock()的开销一共大概 50 皮秒（10 的负 12 次方秒）。synchronize_rcu()的开销从 Power5 单核上的 600 纳秒到 64 核上的超过 100 毫秒不等。

小问题 8.57：可以确定的是，ca-2008 Power 系统的时钟频率相当高，可是即使是 5GHz 的时钟频率，也不足以让读端原语在 50 皮秒执行完毕！这里究竟发生了什么？

不过，本节的实现要求每个线程要么周期性地调用 rcu_quiescent_state()，要么为延长的静止状态调用 rcu_thread_offline()。周期性调用这些函数的要求在某

些情况下会让实现变得困难，比如某种类型的库函数。

小问题 8.58：为什么在库中实现代码要比图 8.44 和图 8.45 中的 RCU 实现更困难？

小问题 8.59：但是如果您在调用 synchronize_rcu()期间持有一把锁，并且在一个读端临界区里去获取同一把锁，会发生什么？应该是死锁，但是一个没有任何代码的原语是怎么参与进死锁循环的呢？

另外，本节的实现不允许并发的 synchronize_rcu()调用来共享同一个优雅周期。不过，完全可以基于这个 RCU 版本写一个产品级的 RCU 实现了。

8.3.4.10. 关于玩具式 RCU 实现的总结

如果您看到这里了，恭喜您！您现在不仅对 RCU 本身有了更清晰的了解，而且对其所需要的软件和应用环境也更熟悉了。想要更进一步了解 RCU 的读者，请看附录 D，其中包含一些在各种产品中大量采用的 RCU 实现。

之前的章节列出了各种 RCU 原语的理想特性。下面我们将整理一个列表，供有意实现自己的 RCU 实现的读者做参考。

1. 必须有读端原语（比如 rcu_read_lock()和 rcu_read_unlock()）和优雅周期原语（比如 synchronize_rcu()和 call_rcu()），任何在优雅周期开始前就存在的 RCU 读端临界区必须在优雅周期结束前完毕。
2. RCU 读端原语应该有最小的开销。特别是应该避免如 cache miss、原子操作、内存屏障和分支之类的操作。
3. RCU 读端原语应该有 O(1)的时间复杂度，可以用于实时用途。（这意味着读者可以与更新者并发运行。）
4. RCU 读端原语应该在所有上下文中都可以使用（在 Linux 内核中，只有 idle 循环时不能使用 RCU 读端原语）。一个重要的特例是 RCU 读端原语必须可以在 RCU 读端临界区中使用，换句话说，必须允许 RCU 读端临界区嵌套。
5. RCU 读端原语不应该有条件判断，不会返回失败。这个特性十分重要，因为错误检查会增加复杂度，让测试和验证变得更复杂。
6. 除了静止状态以外的任何操作都能在 RCU 读端原语里执行。比如像 I/O 这样的 non-idempotent 操作也该允许。
7. 应该允许在 RCU 读端临界区中执行的同时更新一个受 RCU 保护的数据结构。
8. RCU 读端和更新端的原语应该在内存分配器的设计和实现上独立，换句话说，同样的 RCU 实现应该能在不管数据原语是分配还是释放的同时

保护该数据元素。

9. RCU 优雅周期不应该被在 RCU 读端临界区之外阻塞的线程而阻塞。（但是请注意，大多数基于静止状态的实现破坏了这一愿望。）

小问题 8.60：既然在 RCU 读端临界区中禁止开始优雅周期，那么如何才能在 RCU 读端临界区中更新一个受 RCU 保护的数据结构？

8.3.5. RCU 练习

本节由一系列小问题组成，让您可以尝试运用本书之前提到的各种 RCU 例子。每个小问题的答案都会给出一些提示，后续章节会给出详细的解决办法。rcu_read_lock()、rcu_read_unlock()、rcu_dereference()、rcu_assign_pointer()和 synchronize_rcu()原语足以应付这些练习了。

小问题 8.61：图 4.8（count_end.c）中实现的统计计数器用一把全局锁来保护 read_count()中的累加过程，这对性能和可扩展性影响很大。您该如何用 RCU 改造 read_count()，让其拥有良好的性能和极佳的可扩展性呢？（请注意，read_count()的可扩展性受到统计计数器需要扫描所有线程的计数器的限制。）

小问题 8.62：第 4.5 节给出了一段奇怪的代码，用于统计在可移除设备上发生的 I/O 访问次数。这段代码的快速路径（启动一次 I/O）开销较大，因为需要获取一把读写锁。您该如何用 RCU 来改造这个例子，让其拥有良好的性能和极佳的可扩展性呢？（请注意，一般情况下，I/O 访问代码的性能要比设备移除代码的性能重要。）

9. 使用 RCU

本章展示了如何将 RCU 应用到本书之前讨论的某些例子上去。在某些情况下，使用 RCU 可以带来简单的代码，在另一些情况下，使用 RCU 可以让性能和可扩展性变得更好，还有一些情况下，使用 RCU 可以完美地达成上述两点。

9. 1. RCU 和基于每线程变量的统计计数器

第 4.2.4 节描述了一种统计计数器的实现，性能极佳，基本达到了直接相加的性能（C 语言的++操作符），线性的扩展能力——但是这只是对 inc_count()来说。不幸的是，读出计数值的 read_count()函数需要获取一把全局锁，这就带来了高开销和较差的扩展性。基于锁的实现代码在图 4.8 中。

小问题 9.1：我们究竟为什么要用全局锁？

9.1.1. 设计

使用 RCU 而不是 final_mutex 来保护 read_count()中的线程遍历，是为了获得 read_count()函数的极佳性能和可扩展性，而非是让 inc_count()变得更好。但是，我们并不愿意降低累加过程中的精度。比如，在线程退出时，我们决不能丢失退出线程的统计值，也不能把该统计值累加两次。这种错误会导致最终统计值的不精确，换句话说，这种错误会让统计结果毫无用处。并且事实上 final_mutex 的其中一个作用就是确保不会在 read_count()执行的中间创建或撤销线程。

小问题 9.2：那什么是 read_count()的精确度？

因此，如果我们不用 final_mutex，那么我们需要用其他办法来保证精度度。一种办法是将所有之前退出的线程的计数值之和，以及指向每线程计数器的指针数组放在单个结构之中。这样的结构一旦对 read_count()可见，就一只保持不变，如此来确保 read_count()可以看见一致的数据。

9.1.2. 实现

```
1  struct  countarray  {
2     unsigned  long  total;
3     unsigned  long  *counterp[NR_THREADS];
```

```
4 };
5
6 long __thread counter = 0;
7 struct countarray *countarrayp = NULL;
8 DEFINE_SPINLOCK(final_mutex);
9
10 void inc_count(void)
11 {
12   counter++;
13 }
14
15 long read_count(void)
16 {
17   struct countarray *cap;
18   unsigned long sum;
19   int t;
20
21   rcu_read_lock();
22   cap = rcu_dereference(countarrayp);
23   sum = cap->total;
24   for_each_thread(t)
25     if (cap->counterp[t] != NULL)
26 sum += *cap->counterp[t];
27   rcu_read_unlock();
28   return sum;
29 }
30
31 void count_init(void)
32 {
33   countarrayp = malloc(sizeof(*countarrayp));
34   if (countarrayp == NULL) {
35     fprintf(stderr, "Out of memory\n");
36     exit(-1);
37   }
38   memset(countarrayp, '\0',
```

```
sizeof(*countarrayp));
    39 }
    40
    41 void count_register_thread(void)
    42 {
    43   int idx = smp_thread_id();
    44
    45   spin_lock(&final_mutex);
    46   countarrayp->counterp[idx] = &counter;
    47   spin_unlock(&final_mutex);
    48 }
    49
    50 void count_unregister_thread(int
nthreadsexpected)
    51 {
    52   struct countarray *cap;
    53   struct countarray *capold;
    54   int idx = smp_thread_id();
    55
    56   cap = malloc(sizeof(*countarrayp));
    57   if (cap == NULL) {
    58     fprintf(stderr, "Out of memory\n");
    59     exit(-1);
    60   }
    61   spin_lock(&final_mutex);
    62   *cap = *countarrayp;
    63   cap->total += counter;
    64   cap->counterp[idx] = NULL;
    65   capold = countarrayp;
    66   rcu_assign_pointer(countarrayp, cap);
    67   spin_unlock(&final_mutex);
    68   synchronize_rcu();
    69   free(capold);
    70 }
```

图9.1：RCU 和每线程统计计数器

图 9.1 的第 1-4 行是 conutarray 结构体，包含一个 total 字段，作为之前退出线程的统计值，还有一个 counterp[]指针数组，每个指针指向每个正在运行线程的每线程计数器。his structure allows a given execution of read count() to see a total that is consistent with the indicated set of running threads.

第 6-8 行包含对每线程变量 counter 的定义，对指向当前 countarray 结构的全局指针 counterarrayp 的定义，以及对 final_mutex 自选锁的定义。

第 10-13 行是 inc_count()函数，相较图 4.8 没有任何改变。

第 15-29 行是 read_count()函数，与图 4.8 相比改动很大。第 21-27 行用 rcu_read_lock()和 rcu_read_unlock()替换了获取和释放 final_mutex 的操作。第 22 行使用 rcu_dereference()将当前的 countarray 结构保存到局部变量 cap 中。RCU 的恰当使用可以保证该 conutarray 结构至少在第 27 行的 RCU 读端临界区结束之前一直存在。第 23 行用 cap->total 的值初始化 sum，sum 是之前退出的线程的计数值之和。第 24-26 行将当前正在运行的线程的每线程计数器的值加到 sum 中，然后第 28 行返回 sum。

countarrayp 的初始值由第 31-39 行的 count_init()提供。该函数在创建第一个线程之前调用，它的任何是分配并对结构成员置 0，然后将结构的指针赋给 countarrayp。

第 41-48 行是 count_register_thread()函数，由每个新创建的线程调用。第 43 行获取当前线程的 ID，第 45 行获取 final_mutex 锁，第 46 行将当前线程的计数器赋给 countarrayp 的对应项，第 47 行释放 final_mutex 锁。

小问题 9.3：嘿!!!图 9.1 第 45 行修改了之前存在的 countarray 结构中的一个值！你不是说这个结构一旦被 read_count()可见，就一直保持不变吗？？？

第 50-70 行是 count_ungregister_thread()，由每个线程退出时调用。第 56-60 行分配了一个新的 countarray 结构，第 61 行获取 final_mutex 锁，第 67 行释放。第 62 行将当前 countarray 的值复制给刚创建的 countarray 结构，第 63 行将待退出线程的计数器加到新创建的 countarray 结构的 total 字段上，然后第 64 行将待退出线程的的 counterp[]数组元素置 NULL。第 65 行保留一个指向当前（等会儿就是旧的了）countarray 结构的指针，第 66 行用 rcu_assign_pointer()来用新版本的 countarray 结构替换旧的结构。第 68 行等待优雅周期结束，这样所有并发执行 read_count()并且有旧的 countarray 结构的引用的线程，都可以退出它们的 RCU 读端临界区了，然后就可以丢掉对 countarray 的引用。第 69 行就可以安全的释放旧的 countarray 结构了。

9.1.3. 讨论

小问题 9.4：Wow！图 9.1 有 69 行代码，图 4.8 只有 42 行。值得增加这么多复杂性么？

使用 RCU 可以让待退出线程一直等待，直到其他线程都已经将待退出线程的__thread 变量保存下来之后，待退出线程再退出。这就使得 read_count()函数可以不用锁了，因此也为 inc_count()和 read_count()函数提供了极佳的性能和可扩展性。但是，这种性能和可扩展性是以代码复杂度的增加为代价的。希望编译器和库的作者能够给用户态 RCU 提供对__thread 变量的安全的跨线程访问，这将极大地降低使用__thread 变量的用户所面对的复杂性。

9.2. RCU 和可移除 I/O 设备的计数器

第 4.5 节是一段计算可移除设备上发生的 I/O 访问次数的代码。由于需要获取读写锁，所以这段代码在快速路径（开始 I/O）上开销较大。

本节将介绍如何使用 RCU 避免这种开销。

执行 I/O 的代码与原始代码十分相像，只是用 RCU 读端临界区替换了原始代码中的读写锁：

```
1  rcu_read_lock();
2  if  (removing)  {
3     rcu_read_unlock();
4     cancel_io();
5  }  else  {
6     add_count(1);
7     rcu_read_unlock();
8     do_io();
9     sub_count(1);
10  }
```

RCU 读端原语有着最小的开销，因此可以如希望的那样加速快速路径的执行。

移除设备的更新端代码片段如下：

```
1  spin_lock(&mylock);
2  removing  =  1;
3  sub_count(mybias);
4  spin_unlock(&mylock);
```

```
5  synchronize_rcu();
6  while  (read_count()  !=  0)  {
7    poll(NULL,  0,  1);
8  }
9  remove_device();
```

这里我们用互斥自旋锁替换了读写锁，然后增加了一个 synchronize_rcu()，用于等待所有 RCU 读端临界区退出。因为 synchronize_rcu()的缘故，一旦我们执行到第 6 行，我们就知道所有已有的 I/O 已经被处理过了。

当然，synchronize_rcu()的开销仍然很大，但是考虑到移除设备的事件极少发生，所以这仍然是不错的权衡。

10.验证：调试及分析

11.数据结构

12. 高级同步

12.1. 避免锁

方法列表：RCU，非阻塞同步 (特别简单的形式)，内存屏障，延迟处理。

@@@ Pull deferral stuff back to this section?

12.2. 内存屏障

作者: David Howells 和 Paul McKenney.

人们常常从直觉上认为，软件会按顺序及按逻辑因果关系运行，与一般人相比，黑客常常对此理解得更深刻。当编写、分析及调试使用标准互斥机制（如锁及 RCU）的顺序执行代码、并行代码时，这些直觉是非常有效的。

不幸的是，当代码直接使用基于共享内存的数据结构时，这些直觉是完全错误的（驱动编写者使用 MMIO 寄存器时更信任这样的直觉，但是……）。后面的章节将详细的说明这些直觉错在何处，然后提出内存屏障模型来帮助您避免这些陷阱。

12.2.1 节给出内存序及内存屏障的概况。由于这些背景知识是有用的，因此它将使您意识到您的直觉有一点点问题。这一令人痛苦的任务被放在 12.2.2 节。在这一节，展示了一个看起来正确的代码片段，然而非常不幸的是：在真实的硬件上它会出错。在 12.2.3 节中，展示了一些示例代码：它的变量可以同时表现出多个不同的值。 12.2.4 节给出了内存屏障的基本规则。

12.2.1. 内存序及内存屏障

首先，为什么需要内存屏障？CPUs 不能跟踪它们的自己顺序吗？我们拥有计算机的目的，就是为了对各种事情进行跟踪。难道不是这样吗？

现代计算机的一个难题是：主存不能与 CPU 保持同步-在从内存获取一个变量的同时，现代 CPUs 可以执行上千条指令。因此，CPUs 逐渐使用了大量缓存，如图 12.1 所示。某个 CPU 大量使用的变量将保留在 CPU 的缓存中，允许快速的访问相应的数据。

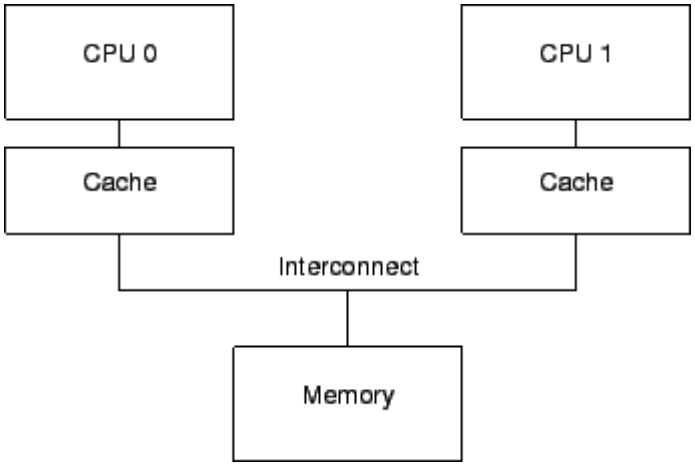


图 12.1：现代计算机系统缓存结构

不幸的是，当一个 CPU 访问不在缓存中的数据时，将导致耗费巨大的“cache miss”来从主存中请求数据。更不幸的是，运行典型的代码会导致大量的 cache misses。为了限制性能下降，CPUs 被设计成在从内存中获取数据的同时执行其他指令。这明显会导致指令和内存引用的乱序执行，并导致严重的混乱，如图 12.2 所示。编译器和同步原语（如锁和 RCU）有责任通过对内存屏障的使用来维护这种用户的直觉（例如，在 LINUX 内核中的 smp_mb()）。这些内存屏障可能是特定的指令比如在 ARM、POWER、 Itanium 和 Alpha 体系中)，也可能由其他其他操作所暗示（例如在 x86 体系中）。

图 12.2：CPUs 可以乱序执行

由于标准的同步原语给人一种按序执行的错觉，你可能希望停止阅读本节并简单的使用这些原语。

但是，如果你需要自己实现同步原语，或者您对理解内存序及内存屏障如何运行有兴趣，就读一下本章吧!

下一节将展示一个与直觉相反的情形。

12.2.2. 如果 B 在 A 后面，并且 C 在 B 后面，为什么 C 不在 A 后面?

内存序和内存屏障可能与直觉有很大差距。例如，考虑图 12.3 中的函数，它并行的执行，并且变量 A、B 和 C 被初始化为 0：

```
1 thread0(void)
2 {
3       A = 1;
4       smp_wmb();
5       B = 1;
6 }
7
8 thread1(void)
9 {
10      while (B != 1)
11              continue;
12      barrier();
13      C = 1;
14 }
15
16 thread2(void)
17 {
18      while (C != 1)
19              continue;
20      smp_mb();
21      assert(A != 0);
22 }
```

图 12.3：Parallel Hardware is Non-Causal

从直觉上来看，thread0() 对 B 的赋值是在对 A 的赋值之后，thread1()在对 C 赋值前，会等待 thread0()对 B 的赋值。 thread2() 在引用 A 前等待 thread1()对 C 的赋值。因此，第 21 行的断言不可能被触发。

请注意，这不是一个仅仅理论上存在的断言：如果在弱序硬件 (一个 1.5GHz 16-CPU POWER 5 系统)上运行这个代码 10M 次的话，将触发 16 次断言。任何想编写内存屏障的人，需要做这些极限测试--虽然正确性证明是有用的，内存屏障反直觉的性质使得我们对这样的证明不敢轻易相信。

问题 12.1：图 12.3 中第 21 行的代码是怎样导致错误的？

问题 12.2：天啊... 我应该如何修复它?

那么您应当做些什么呢？如果可行的话，最好的办法是使用已经存在的原语，这样您可以简单的忽略本节后面的内容。

当然，如果您正在实现同步原语，就没有这么好运了。接下来将讨论，内存序及内存屏障。

12.2.3. 变量可以拥有多个值

认为一个变量将有一个确切的值，这是很自然的想法。不幸的是，是时候对这种想法说“goodbye”了。

要明白这一点，考虑如图 12.4 的代码片段。它被几个 CPUs 并行的执行。第 1 行设置共享变量的值为当前 CPU 的 ID，第 2 行从 gettb()函数对几个值进行初始化，这个函数给出硬件时间计数，它用来在所有 CPUs 之间同步 (不幸的是，并不是所有 CPU 体系都有效!)，第 3-8 行的循环记录变量保持为本 CPU 赋予它的值的时间长度。当然，某一个 CPUs 将是“胜利者”，如果没有第 7-8 行的检测的话，它将永远不会退出。

问题 14.3: 代做片段中所做的什么样的前提条件在真实的硬件上是不存在的？

```
1 state.variable = mycpu;
2 lasttb = oldtb = firsttb = gettb();
3 while (state.variable == mycpu) {
4     lasttb = oldtb;
5     oldtb = gettb();
6     if (lasttb - firsttb > 1000)
7         break;
8 }
```

图 12.4：Software Logic Analyzer

在退出循环前， firsttb 将保存一个时间戳，这是是赋值的时间。lasttb 也保存一个时间戳，它是对共享变量保持最后赋予的值时刻的采样值，如果在进入循环前，共享变量已经变化，那么就等于 firsttb。这允许我们将每个 CPU 的 state.variable 视图分割到一个 532-纳秒的时钟周期中，如图 12.5。这个数据是在一个 1.5GHz POWER5 8 核系统上采集的。每一个核包含一对硬件线程。CPUs 1、2、3 和 4 记录值，而 CPU 0 控制测试。时间戳计数器周期是 5.32ns，这对于我们观察缓存状态来说是足够了。

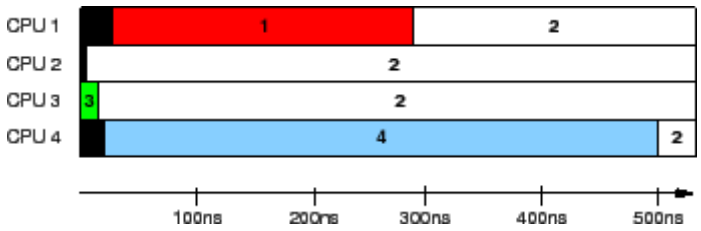


图 12.5：一个变量同时有多个值

每一个水平条表示该 CPU 观察到变量的时间，左边的黑色区域表示相应的 CPU 第一次计数的时间。在最初 5ns 期间，仅仅 CPU 3 拥有变量的值。在接下来的 10ns，CPUs 2 和 3 看到不一致的变量值，但是随后都一致的认为其值是“2”。但是，CPU 1 在整个 300ns 内认为其值是“1”，并且 CPU 4 在整个 500ns 内认为其值是“4”。

问题 14.4：CPUs 怎么会在同一个时刻拥有单个变量的不同视图？

问题 14.5：为 CPUs 2 和 3 这么快就对变量的值达成了一致，而 CPUs 1 和 4 的时间需要这么长?

我们是时候需要告别关于变量值的直觉了。现在需要内存屏障上场了。

12.2.4. 能信任什么东西?

直觉几乎是不可信的。

那么能信任什么呢?

有一些适当简单的规则以允许您用好内存屏障。对于那些希望得到内存屏障底层细节的人来说，至少是从编写可移植代码的角度来说，本节是这些规则的基础。如果您仅仅想知道这些规则是什么，而不想整个的了解所有东西，请直接跳到 12.2.6 节。

内存屏障的详细语义在不同 CPU 之间的变化是很大的，因此可移植代码至少必须遵从最基本的内存屏障语义。

幸运的是，所有 CPUs 有遵从如下规则：

- ✓ 一个特定 CPU 的所有访问操作都与该 CPU 上的程序顺序是一致的。
- ✓ 所有 CPU 对单个变量的访问都与存储这个变量的全局顺序有一些一致性。
- ✓ 内存屏障将成对操作。
- ✓ 互斥锁原语提供这样的手段。

因此，如果你有必要在可移植代码中使用内存屏障，你可以依靠这些属性。每一个属性都将在随后的章节中描述。

12.2.4.1. 自引用是有序的

一个特定 CPU 将象“编程顺序”那样看到它对内存的访问操作。就象 CPU 在同一时刻只执行一条没有乱序的指令一样。对旧的 CPUs 来说，这个限制对二进制兼容来说是必要的。有一些 CPUs 违反这个规则，但是在这些情况下，编译器必须负责确保其顺序（xie.baoyou 注：换句话说，如果您直接编写汇编代码，那么需要小心，这个时候连编程顺序都不能保证）。

另一方面，从编程者的角度来说，CPU 将按照编程序看到对内存的访问操作。

12.2.4.2. 单变量内存一致性

如果一组 CPUs 并发的对单个变量进行存储，那么所有 CPUs 看到的值序列至少在一个全局顺序上是一致的。例如，在图 12.5 所示的访问序列中，CPU 1 看到序列 {1,2}，CPU 2 看到序列{2}，CPU 3 看到序列 {3,2}，CPU 4 看到序列{4,2}。这有一个一致的全局序列{3,1,4,2}（xie.baoyou 注：这代表了这个值在四个 CPU 上的全局赋值顺序）。但是也有五种其他序列，但是这四个值都以“2”结束。

CPU 使用原子操作(如 LINUX 内核的 atomic_inc_return() 原语)，而不是简单存储不同的值，这样观察到的结果将保证是全局一致性的顺序。

12.2.4.3. 成对内存屏障

成对内存屏障提供了有条件的顺序语义（xie.baoyou 注：这句话相当不容易理解，最好是看了后面的内容后再来回味它）。例如，在如下操作中，CPU 1 对 A 的访问绝对早于对 B 的访问。但是，如果 CPU 2 对 B 的访问看到 CPU 1 对 B 的访问，那么 CPU 2 对 A 的访问是确保能够看到 CPU 1 对 A 的访问。虽然某些 CPU 的内存屏障实际上提供更强的保护，但是要做到绝对的顺序保证，可移植代码仅仅只能依赖弱序保证。

CPU 1	CPU 2
access(A);	access(B);
smp_mb();	smp_mb();
access(B);	access(A);

问题 14.6：但是如果内存屏障不能提供绝对的强制顺序，设备驱动者如何可靠的对 MMIO 寄存器执行加载和存储操作？

当然，所谓访问要么是指加载，要么是指存储，二者有不同的属性。表 12.1 展示在一对 CPUs 上可能的存储、加载组合。当然，为了确保操作顺序，必须在每一个操作对之间增加内存屏障。

表 12.1: 内存屏障组合

	CPU 1		CPU 2		De cripti n
	load A)	load(B	load(B)	loa)	Ears to ears.
	load(A)	load(B)	load(B)	store(A)	Only one store.
	load( )	oad(B)	store(B)	load(A)	Only one store.
	load(A)	load(B)	store(B)	store(A)	Pairing 1.
	load(A	sto e(B)	load(B)	load(A)	Only one store.
	load(A)	store(B)	load(B)	store(A)	Pairing 2.
	load(A)	store(B)	store(B)	load(A)	Mouth to mouth, ear to ear.
	load(A)	store(B)	store(B)	store(A)	Pairing 3.
	store(A)	load(B)	load(B)	load(A)	Only one store.
	store(A)	load(B)	load(B)	store(A)	Mouth to mouth, ear to ear.
	store(A)	load(B)	store(B)	load(A)	Ears to mouths.
	store(A)	load(B)	store(B)	store(A)	Stores ``pass in the night".
	store(A)	store(B)	load(B)	load(A)	Pairing 1.
	store(A)	store(B)	load(B)	store(A)	Pairing 3.
	store(A)	store(B)	store(B)	load(A)	Stores ``pass in the night".
	store(A)	store(B)	store(B)	store(A)	Stores ``pass in the night".

12.2.4.4. 成对内存屏障: 可移植的组合

随后的成对操作都是针对表 12.1，并给出可移植代码所需要的所有的内存屏

障组合。

12.2.4.4.1. 第 1 对

在这一对操作中，第一个 CPU 执行一对加载操作，这对操作由内存屏障分开，而第二个 CPU 执行一对存储操作，也用内存屏障分开，如下（A 和 B 都初始化为 0）：

CPU 1	CPU 2
A=1;	Y=B;
smp_mb();	smp_mb();
B=1;	X=A;

在所有 CPUs 都已经完成这些代码后，如果 Y==1，那么必然有 X==1。这种情况下，Y==1 意味着 CPU 2 在内存屏障前的加载已经看到 CPU1 的内存屏障后面的存储。由于内存屏障的成对属性，CPU2 在内存屏障之后的加载必然看到 CPU1 内存屏障之前的存储。

换句话说，如果 Y==0，那么内存屏障的条件不满足。那么，在这种情况下，X 可能是 0 或者 1。

12.2.4.4.2. 第 2 对

每一个 CPU 执行一个加载，后面跟着一个内存屏障，内存屏障后面跟着一个存储，如下(A 和 B 都初始化为 0)：

CPU 1	CPU 2
X=A;	Y=B;
smp_mb();	smp_mb();
B=1;	A=1;

所有 CPU 都执行完以后，如果 X==1，那么必然有 Y==0。在这种情况下，X==1 意味着 CPU 1 在内存屏障之前的加载操作已经看到 CPU2 内存屏障之后的存储操作。由于内存屏障的成对属性，CPU1 在内存屏障后的存储必然也看到 CPU2 在内存屏障之前的加载结果。

换句话说，如果 X==0，那么内存屏障的条件还不存在，因此这种情况下，Y 可能为 0，也可能为 1。

两个 CPU 的代码是对称的，因此两个 CPU 的代码执行后，如果 Y==1 那么必然有 X==0。

12.2.4.4.3. 第 3 对

第一个 CPU 执行一个加载操作，后面跟随一个内存屏障，再后面跟随一个存储操作，而另外一个 CPU 执行一对由内存屏障分开的存储操作，如下(A 和 B 都初始化为 0)：

CPU 1	CPU 2
X=A;	B=2;
smp_mb();	smp_mb();
B=1;	A=1;

当 CPU 都执行完代码后，如果 X==1，那么必然有 B==1。这种情况下，X==1 意味着 CPU 1 在内存屏障之前的加载已经看到 CPU2 内存屏障后面的存储。由于内存屏障是成对的，CPU1 在内存屏障之后的存储也必然看到 CPU2 在内存屏障之前的存储。这意味着 CPU1 对 B 的存储将覆盖 CPU2 对 B 的存储，导致 B==1。

换句话说，如果 X==0，那么内存屏障的条件还不满足，因此在这种情况下，B 可能为 1，也可能为 2。

12.2.4.5. 成对内存屏障: 半可移植组合

表 12.1 后面的操作对可以用于现代硬件，但是在一些 1990s 以前的系统上可能会出问题。但是，它们可以安全的用于自 2000 年以来的主流硬件。

12.2.4.5.1. Ears to Mouths

由于存储不能看到加载的结果 (在此，再一次忽略 MMIO 寄存器)，并不总是能够确定是否满足内存屏障的条件。但是，新的硬件将确保：至少一个加载操作能够看到相应的存储结果。

12.2.4.5.2. Stores ``Pass in the Night''

在下面的例子中，当所有 CPUs 都执行完代码后，您很有可能会认为结果

{A==1,B==2} 不会出现.

CPU 1	CPU 2
A=1;	B=2;
smp_mb();	smp_mb();
B=1;	A=2;

不幸的是, 这样的并不必然适合于 20 世纪的系统。假设包含 A 的缓存行最初由 CPU2 持有，包含 B 的缓存行被 CPU1 持有。那么，在拥有无效队列和存储缓冲的系统中，是有可能出现这样的情况的：前面赋值“pass in the night”，这样随后的赋值实际在第前面的赋值之前发生。这个奇怪 (但是不常见)的效果在附录 C 中解释。

相同的效果也可以在内存屏障位于存储之前的系统中发生，包括“ears to mouths”。

但是，21 世纪的硬件适应了直觉上的按序的操作，允许这种组合被安全的使用。

12.2.4.6. 成对内存屏障: 不可移植组合

表 12.1 中接下来的操作对，内存屏障对它们没有什么效果。

12.2.4.6.1. Ears to Ears

由于加载并不影响内存的状态(在此忽略 MMIO 寄存器)，因此一个加载看到其他加载的结果是不可能的。

12.2.4.6.2. Mouth to Mouth, Ear to Ear

其中一个值仅仅用于加载，另外的值仅仅用于存储。由于(再一次忽略 MMIO 寄存器)加载不可能看到其他操作结果，因此不可能检测到内存屏障的顺序。

12.2.4.6.3. Only One Store

由于仅仅只有一个存储，因此仅仅一个变量允许一个 CPU 看到其他其他 CPU 的访问结果。因此，没有办法检测到由内存屏障提供的顺序。

12.2.4.7. 实现锁操作的语法

假设我们有一个互斥锁(LINUX 内核中的 spinlock_t，pthread 库中的 pthread_mutex_t)，它保护一个变量的值(换句话说，这些变量除了在锁临界区以外，不会被访问)。下面的属性必须存在：

- ✓ 一个特定 CPU 或者线程必须以编程顺序看到自己的所有加载和存储操作。
- ✓ 申请、释放锁必须以全局顺序被看到。
- ✓ 如果一个特定变量在临界区中还没有被存储，那么在临界区中执行的加载操作必须看到上一次存储。

后两个属性之间的不同有一点不好理解：第二点要求申请、释放锁以一种清晰的顺序发生，而第三点要求临界区不“bleed out”。

为什么这些属性是必要的?

假如第一个条件不满足. 那么下面代码中的断言将被触发!

```
a = 1;
b = 1 + a;
assert(b == 2);
```

问题 14.7：为什么 断言 b==2 可能触发?

假设第二个条件不满足。那么下面的代码可能产生内存泄漏!

```
spin_lock(&mylock);
if (p == NULL)
   p = kmalloc(sizeof(*p), GFP_KERNEL);
spin_unlock(&mylock);
```

问题 14.8：为什么这个代码可能产生内存泄漏?

如果第三个条件不满足. 那么下面代码中的计数器可能会倒退。这三个条件是重要的。

```
spin_lock(&mylock);
ctr = ctr + 1;
spin_unlock(&mylock);
```

问题 14.9：为什么上面的代码会产生回退?

如果您已经相信这三个规则是重要的，就让我们看看它们如何与典型的锁实现之间相互影响。

12.2.5. 锁实现回顾

锁和解锁操作的 Naive 伪代码如下所示。注意 atomic_xchg() 原语隐含在原子交换操作前后有一个内存屏障，这去除了在 spin_lock 中使用内存屏障的要求。请注意，并不象名称所示那样，atomic_read() 和 atomic_set() 实际上并不执行任何原子指令，相反，它不过是执行一个简单的加载存储操作。这个伪代码后面是一些 LINUX 的 unlock 操作实现，它在内存屏障后面跟着一个简单的非原子存储。这个最小化的锁实现必须处理 12.2.4 节中所有的锁属性。

```
1 void spin_lock(spinlock_t *lck)
2 {
3     while (atomic_xchg(&lck->a, 1) != 0)
4         while (atomic_read(&lck->a) != 0)
5             continue;
6 }
7
8 void spin_unlock(spinlock_t lck)
9 {
10     smp_mb();
11     atomic_set(&lck->a, 0);
12 }
```

spin_lock() 原语不能继续运行，直到之前的 spin_unlock() 原语完成。如果 CPU 1 释放一个锁，CPU 2 尝试获取它，那么操作序列可能如下:

CPU 1	CPU 2
(critical section)	atomic_xchg(
smp_mb();	lck->a->1
lck->a=0;	lck->a->1
	lck->a->0
	(implicit smp_mb() 1)
	atomic_xchg(
	(implicit smp_mb() 2)
	(critical section)

在这种情况下，成对的内存屏障足以适当的维护两个临界区。CPU 2 的 atomic_xchg(&lck->a, 1)看到 CPU 1 lck->a=0，因此 CPU 2 临界区的任何地方都必然看到 CPU 临界区前面的任何内容。相反的，CPU 1 的临界区不能看到 CPU 2

的临界区中的任何内容。

12.2.6. 一些简单的规则

也许理解内存屏障的最简单方法是理解下面这些简单规则：

- ✓ 每一个 CPU 按顺序看到它自己的访问。
- ✓ 如果一个单一共享变量被不同 CPU 加载保存，那么被一个特定 CPU 看到的值的序列将与其他 CPU 看到的序列是一致的。并且，至少存在一个这样的序列：它包含了向这个变量存储的所有值，每个 CPUs 序列将与这个序列一致。
- ✓ 如果一个 CPU 按顺序存储变量 A 和 B，并且如果第二个 CPU 按顺序装载 B 和 A，那么，如果第二个 CPU 对 B 的装载得到了第一个 CPU 对它存储的值，那么第二个 CPU 对 A 的装载也必然得到第一个 CPU 对它存储的值。
- ✓ 如果一个 CPU 在对 B 进行存储前，按顺序对 A 执行一个加载操作，并且，如果第二个 CPU 在对 A 进行存储前，按顺序执行一个对 B 的存储，并且，如果第二个 CPU 对 B 的加载操作得到了第一个 CPU 对它的存储结果，那么第一个 CPU 对 A 的加载操作必然不会得到第二个 CPU 对它的存储。
- ✓ 如果一个 CPU 在对 B 进行存储前，按序进行一个对 A 的加载操作，并且，如果第二个 CPU 在对 A 进行存储前，按序对 B 进行存储，并且，如果第一个 CPU 对 A 的加载得到第二个 CPU 对它的存储结果，那么第一个 CPU 对 B 的存储必然发生在第二个 CPU 对 B 的存储之后，因此第一个 CPU 保存的值被保留下来。

12.2.7. 抽象内存访问模型

考虑如图 12.6 所示的抽象系统模型：

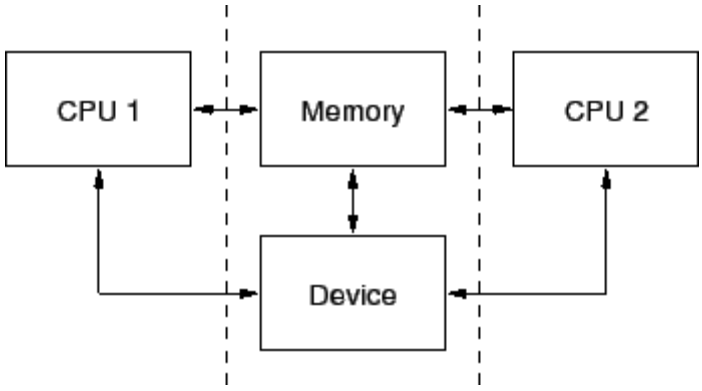


图 12.6: 抽象内存访问模型

每一个 CPU 执行一个产生内存访问操作的程序。在这个抽象 CPU 中，内存操作顺序是非常随意的，一个 CPU 实际上可以按它喜欢的顺序执行内存操作，只要程序因果关系能够得到保持。类似的，编译器也可以按它喜欢的顺序安排指令。

因此在上图中，一个 CPU 执行的内存操作的效果被系统中其他部分认为交叉穿过 CPU 和系统其他部分(虚线)。

例如, 考虑以下的事件序列，给定其初始值{A = 1, B = 2}：

CPU 1	CPU 2
A = 3;	x = A;
B = 4;	y = B;

被内存系统中途看到结果可以有 24 种组合，装载表示为“ld”，存储表示为“st”：

st A=3,
st B=4,
x=ld A3,
y=ld B4

st A=3,
st B=4,
y=ld B4,
x=ld A3

st A=3,
x=ld A3,
st B=4,
y=ld B4

st A=3,
x=ld A3,
y=ld B2,
st B=4

st A=3,
y=ld B2,
st B=4,
x=ld A3

st A=3,

y=ld B2,
x=ld A3,
st B=4

st B=4,
st A=3,
x=ld A3,
y=ld B4

st B=4,
...

...

因此导致 4 个不同的组合:

x == 1,
y == 2

x == 1,
y == 4

x == 3,
y == 2

x == 3,
y == 4

而且，由一个 CPU 向内存提交的存储在提交的同时，可能不能被另外一个 CPU 所察觉。

更进一步的例子，考虑如下给定初始值{A = 1, B = 2, C = 3, P = &A, Q = &C}的事件序列：

CPU 1	CPU 2
B = 4;	Q = P;
P = &B	D = *Q;

这里有一个明显的数据依赖，CPU2 装载进 D 的值依赖于从 P 得到的地址。最后，以下结果都是可能发生的:

(Q == &A)
and
(D == 1)

(Q == &B)
and

(D == 2)

(Q == &B)
and
(D == 4)

注意 CPU 2 从不会试图装载 C 到 D，因为 CPU 将在装载*Q 前装载 P 到 Q。

12.2.8. 设备操作

一些设备将它们的控制接口呈现为内存操作，但是控制寄存器访问的顺序非常重要。例如，假设一个有一些内部寄存器的以太网卡通过一个地址端口寄存器（A）和一个数据端口寄存器（D）访问。要读取内部寄存器 5，可能会使用以下代码：

```
*A = 5;
x = *D;
```

但是，这可能会出现如下两种顺序:

```
STORE *A = 5, x = LOAD *D
x = LOAD *D, STORE *A = 5
```

第二种情况几乎可以确定会出现故障，因为它在试图读取寄存器后设置地址。

12.2.9. 保证

可以期望一个 CPU 有一些最小的保证：

✓ 在一个给定的 CPU 上，依赖的内存访问将按序运行。

这意味着：

```
Q = P; D = *Q;
```

该 CPU 将按如下内存操作顺序运行:

```
Q = LOAD P, D = LOAD *Q
```

并且总是按这个顺序.

✓ 在个别 CPU 中，重叠的加载存储操作将是按顺序运行。

这意味着：

```
a = *X; *X = b;
```

该 CPU 仅仅按下面的内存操作顺序运行：

```
a = LOAD *X, STORE *X = b
```

并且：

```
*X = c; d = *X;
```

该 CPU 将仅仅按如下顺序运行:

```
STORE *X = c, d = LOAD *X
```

✓ 对单个变量的存储序列将向所有 CPUs 呈现出一个单一的序列，虽然这

个序列可能不能被代码所预见，实际上，在多次运行时，其顺序可能发生变化。

并且，有一些事情必须被假设，或者必须不被假设：

✓ 不能保证独立的加载和存储操作会按给定顺序运行。

这意味着：

X = *A; Y = *B; *D = Z;

可能得到如下序列:

X = LOAD *A, Y = LOAD *B, STORE *D = Z
X = LOAD *A, STORE *D = Z, Y = LOAD *B
Y = LOAD *B, X = LOAD *A, STORE *D = Z
Y = LOAD *B, STORE *D = Z, X = LOAD *A
STORE *D = Z, X = LOAD *A, Y = LOAD *B
STORE *D = Z, Y = LOAD *B, X = LOAD *A

✓ 必须假设：重叠的内存访问将被合并或者丢弃。

这意味着:

X = *A; Y = *(A + 4);

我们可能得到如下顺序:

X = LOAD *A; Y = LOAD *(A + 4);
Y = LOAD *(A + 4); X = LOAD *A;
{X, Y} = LOAD {*A, *(A + 4) };

并且：

*A = X; Y = *A;

可能得到如下顺序之一:

STORE *A = X; Y = LOAD *A;
STORE *A = Y = X;

12.2.10. 什么是内存屏障?

正如前面看到的，独立的内存操作以随机顺序执行，对于 CPU-CPU 交互及 IO 来说，这是一个问题。需要什么来指示编译器和 CPU 以限制其顺序？

内存屏障就是这样的手段。它们影响屏障两边的内存操作顺序。

这是重要的，因为系统中的 CPUs 和其他设备可以使用多种技术来提高性能--包括重排，延迟和组合内存操作；加载冒险；分支预测冒险及多种类型的缓存。内存屏障用来撤销或者制止这些技巧，允许代码安全的控制 CPU 之间或者 CPU 与设备之间的交互。

12.2.10.1.内存屏障详解

内存屏障有四个基本变种:

- ✓ 写 (或存储) 内存屏障，
- ✓ 数据依赖屏障，
- ✓ 读 (或者) 内存屏障，
- ✓ 通用内存屏障。

每一个变种都将在随后介绍。

12.2.10.1.1. 写内存屏障

一个写内存屏障提供这样的保证：在屏障之前的写操作看起来将在其后的写操作之前发生。

写屏障仅仅对写操作进行排序。它对加载没有任何效果。

CPU 可以被视为按时间顺序提交一系列存储操作。所有在写屏障之前的存储将发生在所有屏障之后的存储操作之前。

注意写屏障通常应当与读或者数据依赖屏障配对使用，参见“SMP 屏障对”一节。

12.2.10.1.2. 数据依赖屏障

数据依赖屏障是一种弱的读屏障形式。当两个装载操作中，第二个依赖于第一个的结果时 (如，第一个装载得到第二个装载的地址)，需要一个数据依赖屏障，以确保第二个装载的目标将在第一个地址装载之前被更新。

数据依赖屏障仅仅对相互依赖的加载进行排序。它对任何存储没有效果，对相互独立的加载或者重叠加载也没有效果。

正如读内存屏障中提到的一样，系统中的其他 CPUs 可以被视为向内存系统的提交序列。如果装载涉及到其他 CPUs 的存储操作中的一个，那么在屏障完成时，装载涉及到的存储之前的存储的效果将被数据依赖之后的任何装载所察觉。

参见“内存屏障序列示例”一节。

注意第一个装载实际上需要一个数据依赖而不是控制依赖。如果第二个装载的地址依赖于第一个装载，但是依赖的是一个条件而不是实际装载地址本身，那么它就是一个控制依赖，这需要一个完整的读屏障。参见“控制依赖”一节以了解更多信息。

注意数据依赖屏障通常应当与写屏障配对；参见“SMP 屏障对”一节。

12.2.10.1.3. 读内存屏障

读屏障是一个数据依赖屏障（xie.baoyou 注：但不仅仅是），并且，所有在屏障之前的加载操作将在屏障之后的加载操作之前发生，同时这个顺序要被系统中其他组件所认可。

读屏障仅仅对加载进行排序，它对存储没有任何效果。

读内存屏障隐含数据依赖屏障，因此可以替代它。

注意，读屏障通常应当与写屏障配对使用。参见“SMP 屏障对”一节。

12.2.10.1.4. 通用内存屏障

通用内存屏障保证屏障之前的加载、存储操作都将在屏障之后的加载、存储操作之前被系统中的其他组件看到。

通用内存屏障同时对加载和存储操作进行排序。

通用内存屏障隐含读和写内存屏障，因此也可以替换它们中的任何一个。

12.2.10.2.隐含的内存屏障

有一组隐含的内存屏障，这样称呼它们是因为它们被嵌入到锁原语中：

LOCK 操作和

UNLOCK 操作。

12.2.10.2.1. LOCK 操作

一个锁充当了一个屏障。它确保所有锁操作后面的内存操作看起来发生在锁操作之后。

锁操作之前的内存操作可能发生在它完成之后。

LOCK 操作几乎总是与 UNLOCK 操作配对。

12.2.10.2.2. UNLOCK 操作

Unlock 操作也充当一个屏障。它确保在 UNLOCK 操作之前的所有内存操作看起来发生在 UNLOCK 之前。

UNLOCK 操作之后的内存操作看起来可能发生在它完成之前。

LOCK 和 UNLOCK 操作确保相互之间严格按顺序执行。

LOCK 和 UNLOCK 操作的用法通常可以避免再使用其他类型的内存屏障(但是请注意在“MMIO 写屏障”一节中提到的例外).。

问题 14.10：以下对``a" 和 ``b"的存储序列有什么效果?

```
a = 1;
b = 1;
<write barrier>
```

12.2.10.3.关于内存屏障，不能做什么假设?

这是必然的事情：内存屏障不能超出给定体系结构限制：

- ✓ 不能保证在内存屏障之前的内存访问将在内存屏障指令完成时完成；屏障仅仅用来在 CPU 的访问队列中做一个标记，表示相应的访问不能被穿越。
- ✓ 不能保证在一个 CPU 中执行一个内存屏障将直接影响另外一个 CPU 或者影响系统中其他硬件。间接效果是第二个 CPU 看到第一个 CPU 的访问是按顺序的。
- ✓ 不能保证一个 CPU 将看到第二个 CPU 访问操作的正确顺序，即使第二个 CPU 使用一个内存屏障，除非第一个 CPU 也使用一个配对的内存屏障(参见“SMP 屏障对”一节)。
- ✓ 不能保证某些 CPU 片外硬件不会重排对内存的访问。CPU 缓存一致性机制将在 CPUs 之间传播内存屏障的间接影响，但是不会按顺序执行这项操作。

12.2.10.4.数据依赖屏障

使用内存依赖屏障的需求是有点微妙的，对它的需求并不总是非常明显。举例说明，考虑以下事件序列，初始值是{A = 1, B = 2, C = 3, P = &A, Q = &C}：

CPU 1	CPU 2
B = 4;	
<write barrier>	
P = &B;	
	Q = P;
	D = *Q;

很显然，这是一个数据依赖，从直觉上看，最终 Q 必须是&A 或者 &B, 并且：

(Q == &A)

意味着

(D == 1)

(Q == &B)

意味着

(D == 4)

与直觉相反，很有可能 CPU 2 在察觉到 B 之前，察觉到 P 被更新, 因此导致以下的情形:

(Q == &B)

并且

(D == 2) ????

这看起来象是一致性错误或者因果维护错误，其实不是，这种情况可能在某些实际的 CPUs 中找到 (如 DEC Alpha)。

要处理这种情况，必须在地址加载和数据加载之间插入一个数据依赖屏障 (初始值仍然是 {A = 1, B = 2, C = 3, P = &A, Q = &C}):

CPU 1	CPU 2
B = 4;	
<write barrier>	
P = &B;	
	Q = P;
	<data dependency barrier>
	D = *Q;

这强制两种情况之一发生，并防止出现第三种可能。

注意这种极端违反直觉的情形在分离缓存机器中非常容易出现。因此，例如，一个缓存带处理偶数编号的缓存行，另一个缓存带处理奇数编号的缓存行。P 指针可能存储在奇数编号的缓存行，变量 B 存储在偶数编号的缓存行。那么，如果偶数编号的缓存带很忙，而奇数编号的缓存带空闲，就可以出现指针 P 的新值被看到(&B)，而变量 B 的旧值被看到 (1)。

另一个需要数据依赖屏障的例子是，从内存中读取一个编号，用于计算数组访问的索引，假设初始值为{M[0] = 1, M[1] = 2, M[3] = 3, P = 0, Q = 3}：

CPU 1	CPU 2
M[1] = 4;	
<write barrier>	
P = 1;	
	Q = P;
	<data dependency barrier>
	D = M[Q];

数据依赖屏障对于 LINUX 内核的 RCU 系统来说非常重要，例如，参考 include/linux/rcupdate.h 中的 rcu_dereference()。它允许 RCU 指针被替换成一个新值。

参见 @@@"缓存一致性" 一节以了解更彻底的例子。

12.2.10.5.控制依赖

控制依赖需要一个完整的读内存屏障，而不是简单的数据依赖屏障。考虑下面的代码:

```
1 q = &a;
2 if (p)
3     q = &b;
4 <data dependency barrier>
5 x = *q;
```

这不会达到期望的效果，因为这实际上不是数据依赖，而是一个控制依赖。这种情况初始需要的是:

```
1 q = &a;
2 if (p)
3     q = &b;
4 <read barrier>
5 x = *q;
```

12.2.10.6.SMP 屏障对

当处理 CPU-CPU 交互时，几种类型的内存屏障总是应当配对。缺少适当的配对将总是会产生错误。

一个写屏障应当总是与数据依赖屏障或者读屏障配对，虽然通用屏障也是可以的。类似的，一个读屏障或者数据依赖屏障总是应当与至少一个写屏障配对使用，虽然，通用屏障也是可以的：

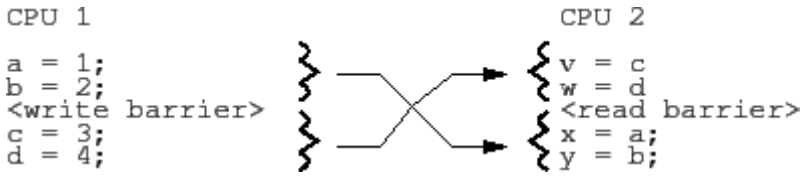
CPU 1	CPU 2
a = 1;	
<write barrier>	
b = 2;	
	x = b;
	<read barrier>
	y = a;

或者：

CPU 1	CPU 2
a = 1;	
<write barrier>	
b = &a;	
	x = b;
	<data dependency barrier>
	y = *x;

不管怎样，必须有读屏障，即使它可能是一个弱的读屏障。[14.8]

注意，在写屏障之前的存储通常预期来匹配读屏障或者数据依赖屏障之后的加载，反之亦然。



12.2.10.7.内存屏障配对示例

首先，写屏障仅仅对写操作的顺序有效。考虑如下事件序列：

STORE A = 1
STORE B = 2
STORE C = 3
<write barrier>
STORE D = 4
STORE E = 5

这个事件序列以这种顺序提交到内存一致性系统：系统中的其他部分可能会在乱序的操作集{D=4,E=5}之前察觉到乱序的操作集{A=1,B=2,C=3}，如图 12.7 所示.

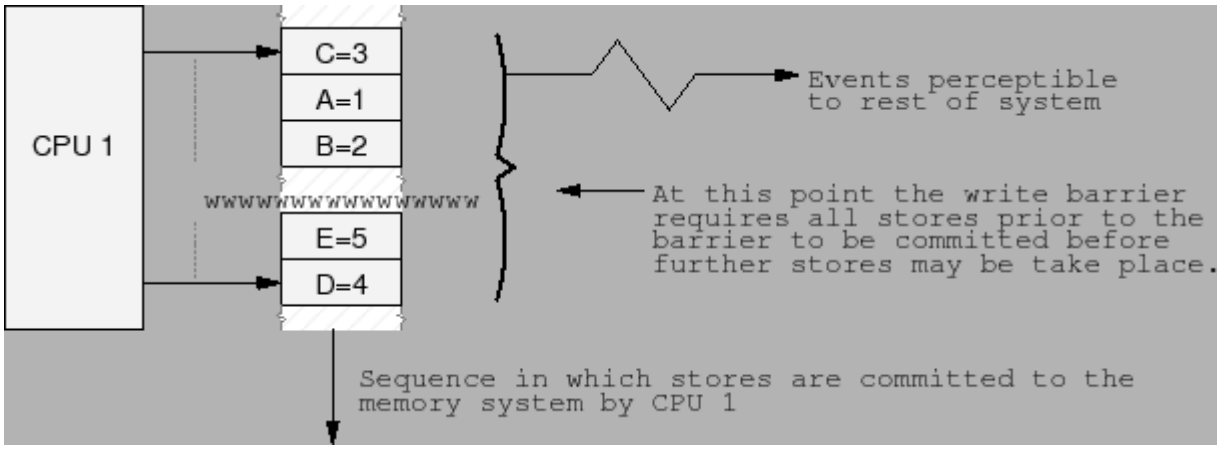


图 12.7: 写屏障语义

其次，数据依赖屏障仅仅到数据依赖装载操作有效。考虑如下事件序列，初始值为{B = 7, X = 9, Y = 8, C = &Y}：

CPU 1	CPU 2
a = 1;	
b = 2;	
<write barrier>	
c = &b;	LOAD X

d = 4;	LOAD C (gets &B)
	LOAD *C (reads B)

没有屏障进行干涉的话，CPU 2 可能会以随机的顺序察觉到 CPU1 上的事件，即使 CPU1 使用了写屏障：

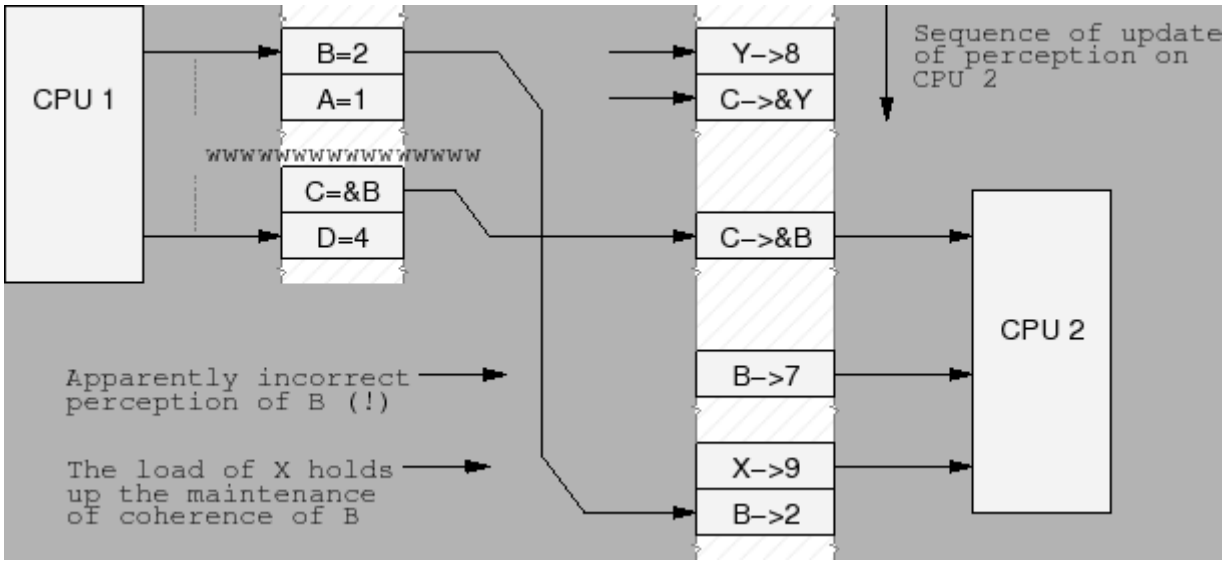


图 12.8: 数据依赖屏障

在上面的例子中，CPU 2 察觉到 B 为 7，尽管装载*C (为 B) 在加载 C 之后。

但是，如果在 CPU2 装载 C 和装载*C 之间放置一个数据依赖屏障，也使用初始值 {B = 7, X = 9, Y = 8, C = &Y}：

CPU 1	CPU 2
a = 1;	
b = 2;	
<write barrier>	
c = &b;	LOAD X
d = 4;	LOAD C (gets &B)
	<data dependency barrier>

	LOAD *C (reads B)

那么其顺序将象直觉所希望的那样，如图 12.9 所示。

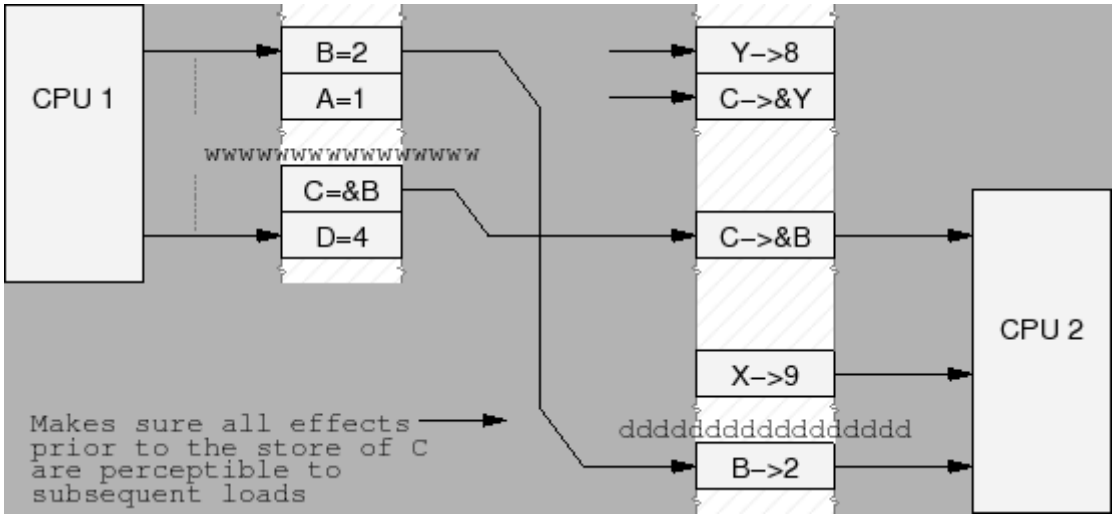


图 12.9：数据依赖屏障

第三，读屏障仅仅对装载有效，考虑如下的事件序列，其初始值为{A = 0, B = 9}：

CPU 1	CPU 2
a = 1;	
<write barrier>	
b = 2;	
	LOAD B
	LOAD A

如果没有屏障干预，CPU 2 可能以随机的顺序察觉到 CPU1 的事件，尽管 CPU1 使用了写屏障：

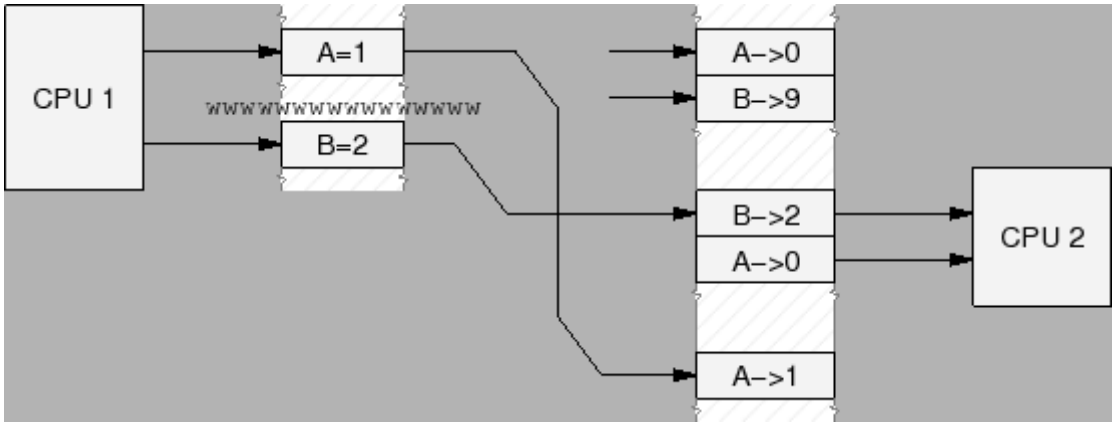


图 12.10: 需要读屏障

但是，如果在 CPU2 装载 B 和装载 A 之间放置一个读屏障，也使用初始值{A = 0, B = 9}:

CPU 1	CPU 2
a = 1;	
<write barrier>	
b = 2;	
	LOAD B
	<read barrier>
	LOAD A

那么被 CPU1 的写屏障影响的顺序将被 CPU2 正确的察觉到，如图 12.11 所示。

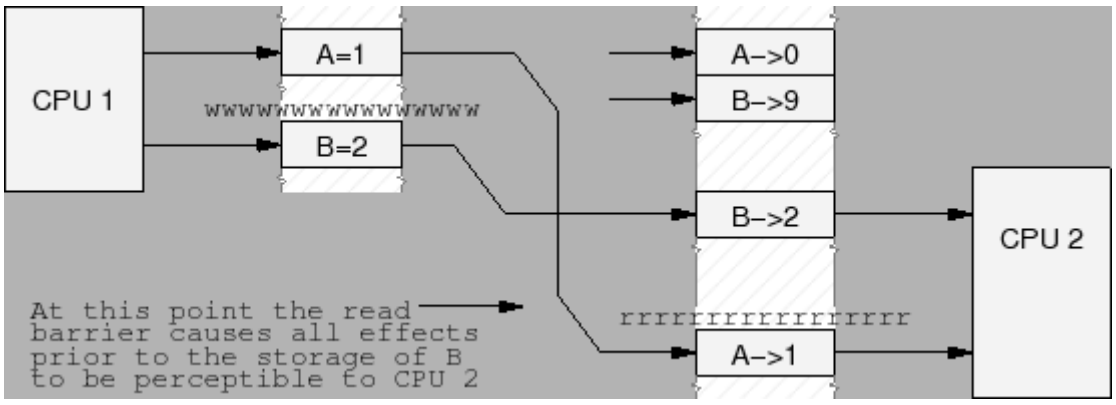


图 12.11：使用读屏障

为了更彻底的说明这一点，考虑如果在读屏障的两边包含装载 A 的代码会发生什么，仍然使用初始值{A = 0, B = 9}:

CPU 1	CPU 2
a = 1;	
<write barrier>	
b = 2;	
	LOAD B
	LOAD A (1^{st})
	<read barrier>
	LOAD A (2^{nd})

虽然两次装载 A 都发生在装载 B 之后，它们可能会得到不同的值，如图 12.12 所示。

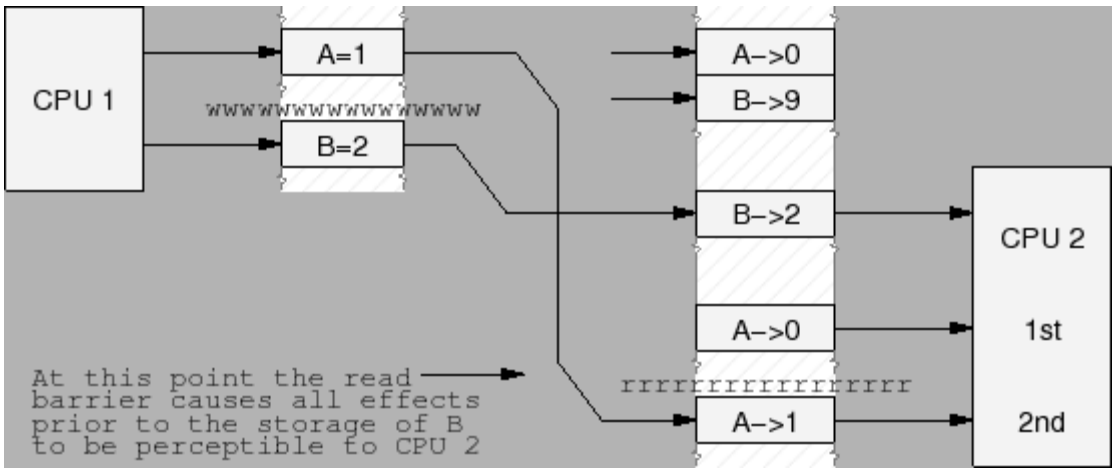


图 12.12：双重加载，使用读屏障

当然，CPU2 对 A 的更新在读屏障完成之前变得对 CPU2 可见，这也是可能的，如图 12.13 所示。

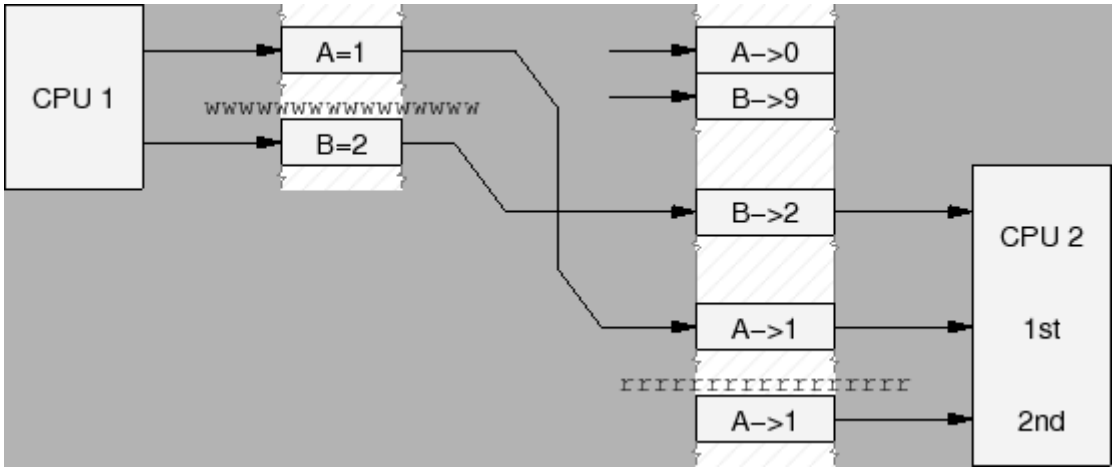


图 12.13: 提供读屏障，两次获取

如果对 B 的装载得到 B == 2，那么就能够保证第二次装载总是能够得到 A == 1。不能保证第一次对 A 的装载也是如此。它可能得到 A == 0 也可能得到 A == 1。

12.2.10.8.读内存屏障 vs. 加载冒险

许多 CPUs 对装载进行冒险：也就是说，它们发现自己将从内存装载一个项目，并且它们发现某个时刻没有将总线用于其他加载操作，那么就提前进行加载—即使它们还没有实际到达指令执行点。以后，这允许实际的加载指令立即完成，因为 CPU 已经得到它的值了。

结果可能是 CPU 实际上并不需要这个值 (也许是由于加载在分支内)，这种情况下它可能丢弃值或者仅仅缓存起来随后使用。例如，考虑如下情况：

CPU 1	CPU 2
	LOAD B
	DIVIDE
	DIVIDE
	LOAD A

在某些 CPUs 中，divide 指令需要一个较长时间才能完成，这意味着 CPU2 的总线在此期间可能空闲。因此，CPU2 可能在 divides 完成前冒险加载 A。当某个 dividees 发生异常时，这个冒险加载必须废弃，但是通常情况下，同时进行加载和 divides 将允许加载操作更快的完成，如图 12.14 所示。

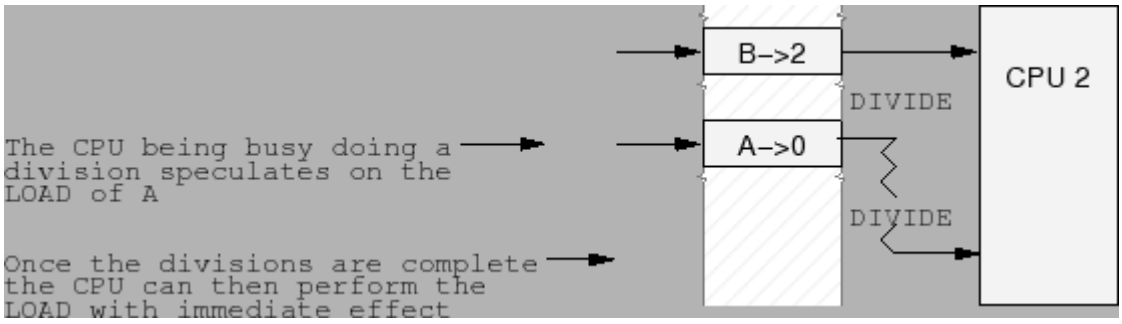


图 12.14: 冒险加载

在第二个加载前放置一个读屏障或者数据依赖屏障：

CPU 1	CPU 2

	LOAD B
	DIVIDE
	DIVIDE
	<read barrier>
	LOAD A

这将强制任何冒险获得的值被重新考虑。如果冒险内存点没有变化，那么冒险获得的值将被使用，如图 12.15 所示。另一方面，如果其他 CPU 对 A 进行了更新或者使它无效了，那么冒险过程将被中止，A 的值将被重新装载，如图 12.16 所示.

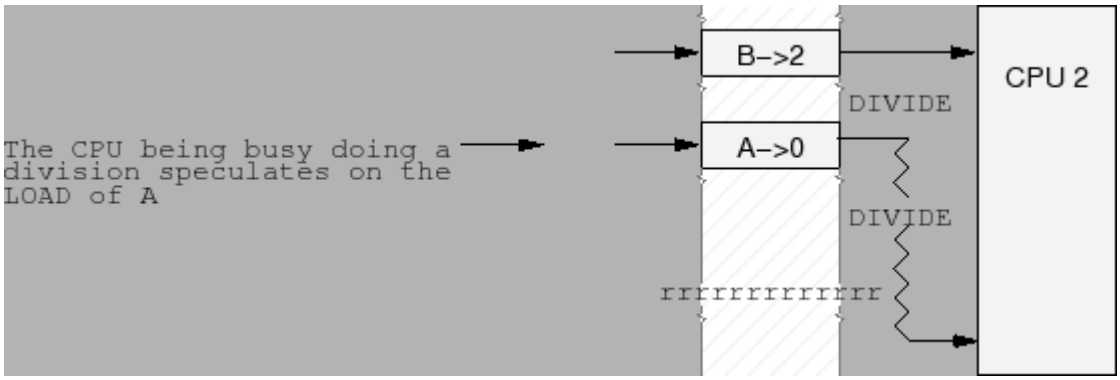


图 12.15: 冒险加载及屏障

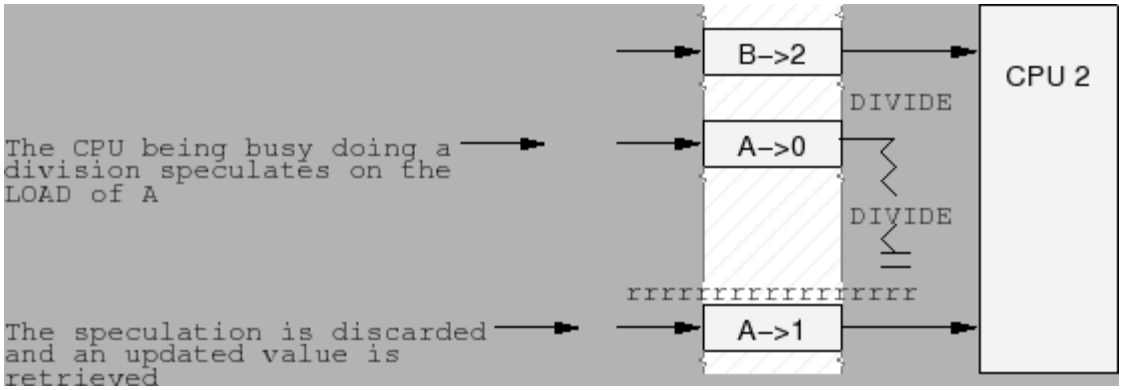


图 12.16：冒险加载被屏障中止

12.2.11. 锁约束

正如较早前提醒的一样，锁原语包含了隐含的内存屏障。这些隐含的屏障提供了如下保障：

LOCK 操作保证:

- ✓ LOCK 之后的内存操作将在 LOCK 操作操作完成之后完成。
- ✓ LOCK 操作之前的内存操作可能在 LOCK 操作完成之后完成。

UNLOCK 操作保证:

- ✓ UNLOCK 之前的内存操作将在 UNLOCK 操作完成前完成.
- ✓ UNLOCK 之后的操作可能在 UNLOCK 操作完成前完成。

LOCK vs LOCK 保证:

- ✓ 所有在另外的 LOCK 之前的 LOCK 操作，将在 LOCK 操作之前完成.

LOCK vs UNLOCK 保证:

- ✓ 所有在 UNLOCK 操作之前的 LOCK 操作将在 UNLOCK 操作之前完成。
- ✓ 所有在 LOCK 之前的 UNLOCK 操作将在 LOCK 操作之前完成.

失败的 LOCK 不能保证:

- ✓ 几种 LOCK 变体操作可能失败，可能是由于不能立即获得锁，也可能是由于接收到一个非阻塞信号或者在等待锁可用时发生异常。失败的锁并不隐含任何类型的屏障。

12.2.12. 内存屏障示例

12.2.12.1.锁示例

12.2.12.1.1. LOCK 后跟随 UNLOCK

LOCK 后面跟随 UNLOCK 不能设定为全内存屏障，因为 LOCK 之前的操作可能发生在 LOCK 之后，并且 UNLOCK 之后的访问可能发生在 UNLOCK 之前，因此这两个访问可能相互交叉。例如：

```
1 *A = a;
2 LOCK
3 UNLOCK
4 *B = b;
```

可能按如下顺序执行：

```
2 LOCK
4 *B = b;
1 *A = a;
3 UNLOCK
```

同样的,总是要牢记 LOCK 和 UNLOCK 都允许使之前的操作进入临界区。

问题 14.11：LOCK-UNLOCK 操作的序列如何才能是一个全内存屏障？

问题 14.12：什么样的 CPUs 存在这样的内存屏障指令，使得这些可穿透的锁原语能够产生？（xie.baoyou 注：应当是那些分别实现了读屏障和写屏障指令的体系）

12.2.12.1.2. 基于 LOCK 的临界区

虽然一对 LOCK-UNLOCK 不能起到全内存屏障的作用，但是这些操作还是会影响内存屏障。

考虑下面的代码:

```
1 *A = a;
2 *B = b;
3 LOCK
4 *C = c;
5 *D = d;
6 UNLOCK
7 *E = e;
8 *F = f;
```

这可以合法的按如下顺序执行，在同一行的成对操作表示 CPU 并发的执行这些操作：

```
3 LOCK
1 *A = a; *F = f;
7 *E = e;
4 *C = c; *D = d;
2 *B = b;
6 UNLOCK
```

表 12.2：基于锁的临界区

	Ordering: legitimate or not?
	*A; *B; LOCK; *C; *D; UNLOCK; *E; *F;
	*A; *B; LOCK; *C; *D; UNLOCK; *E; *F;
	*F; *A; *B; LOCK; *C; *D; UNLOCK; *E;
	*A; *B; LOCK; *C; *D; UNLOCK; *E; *F;
	*B; LOCK; *C; *D; *A; UNLOCK; *E; *F;
	*A; *B; *C; LOCK; *D; UNLOCK; *E; *F;
	*A; *B; LOCK; *C; UNLOCK; *D; *E; *F;

	*B; *A; LOCK; *D; *C; UNLOCK; *F; *E;
	*B; LOCK; *C; *D; UNLOCK; *F; *A; *E;

问题 14.13：对于乱序并发执行的成组操作，表 12.2 中的哪一行，对 A 到 F 变量的赋值和 LOCK/UNLOCK 操作乱序是合法的? 为什么要乱序，为什么不乱序?

12.2.12.1.3. 多个锁的顺序:

包含多个锁的代码仍然看到包含这些锁的顺序约束，但是必须小心。例如，考虑表 12.2 中的代码，名为“M”和“Q”的锁对是如何用的。

表 12.3: 多个锁的顺序

CPU 1	CPU 2
A = a;	E = e;
LOCK M;	LOCK Q;
B = b;	F = f;
C = c;	G = g;
UNLOCK M;	UNLOCK Q;
D = d;	H = h;

在这个例子中，不能保证对“A”到“H”的赋值顺序将如何发生。

问题 14.14：表 12.3 有什么约束?

12.2.12.1.4. 多 CPUs 使用同一个锁的顺序:

如果将表 12.3 中的不同的锁进行替换，所有 CPUs 都申请同一个锁。如表 12.4 所示：

表 12.4: 多 CPUs 使用同一个锁的顺序

CPU 1	CPU 2

A = a;	E = e;
LOCK M;	LOCK M;
B = b;	F = f;
C = c;	G = g;
UNLOCK M;	UNLOCK M;
D = d;	H = h;

在这种情况下，只要 CPU 1 在 CPU2 前申请到 M，或者相反。在第一种情况下，对 A、B、C 的赋值，必然在对 F、G、H 的赋值之前。另一方面，如果 CPU2 先申请到锁，那么对 E、F、G 的赋值必然在对 B、C、D 的赋值之前。

12.2.13. CPU

对内存屏障的顺序的察觉会受到 CPU 和内存之间的缓存的影响。只要缓存一致性协议维护内存一致性的顺序（xie.baoyou 注：这应当是针对某些 CPU 可以既使用缓存，又确保强序的情况来说的。这种情况下，不存在内存屏障的问题）。从软件的角度来说，这些缓存的目的都针对内存的。内存屏障可以被认为对图 12.17 中垂直线有作用。确保 CPU 按适当的顺序向内存展示值，就象确保其他 CPU 按适当顺序看到值的变化一样。

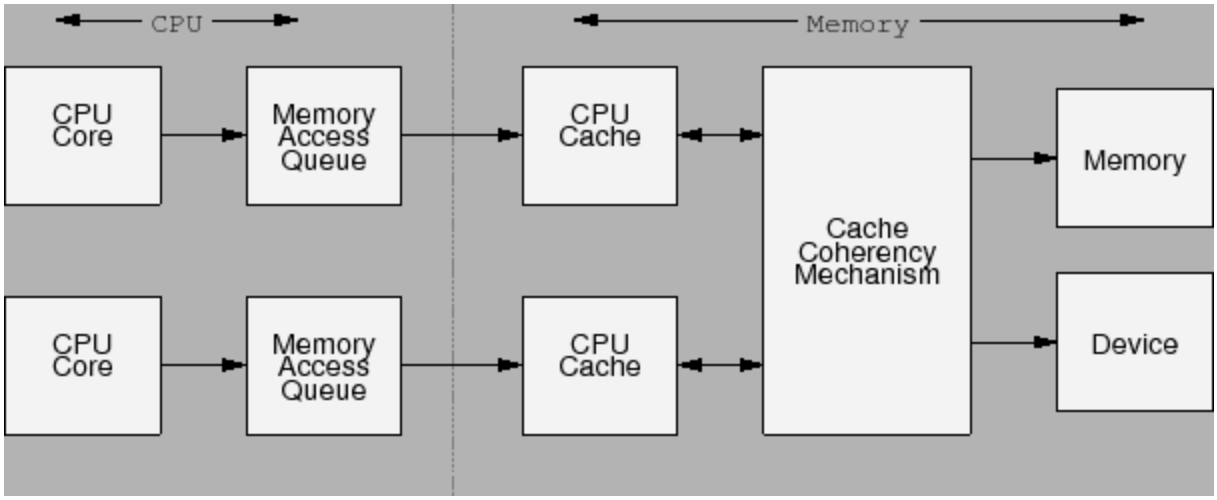


图 12.17: 内存体系

虽然缓存可以“隐藏”特定 CPU 对系统中其他部分的内存访问，缓存一致性协议确保所有其他 CPU 能够看到这些被隐藏的访问的影响，迁移并将缓存行无效是需要的。而且，CPU 核可能以任何顺序执行指令，仅有的限制是程序因果关系及被维护的内存顺序。这些指令可能是通常的内存访问，它们必须在 CPU

的内存访问队列中排序，但是可能继续执行，直到 CPU 已经用完它的内部资源，或者直到必须等待排队的内存访问完成。

12.2.13.1.缓存一致性

虽然缓存一致性协议保证特定 CPU 按顺序看到自己对内存的访问，并且所有 CPU 对包含在单个缓存行的单个变量的修改顺序是一致的，但是不保证对不同变量的修改能够按照顺序被其他所有 CPUs 看到--虽然某些计算机系统做出了这样的保证，但是可移植软件不能依赖它们。

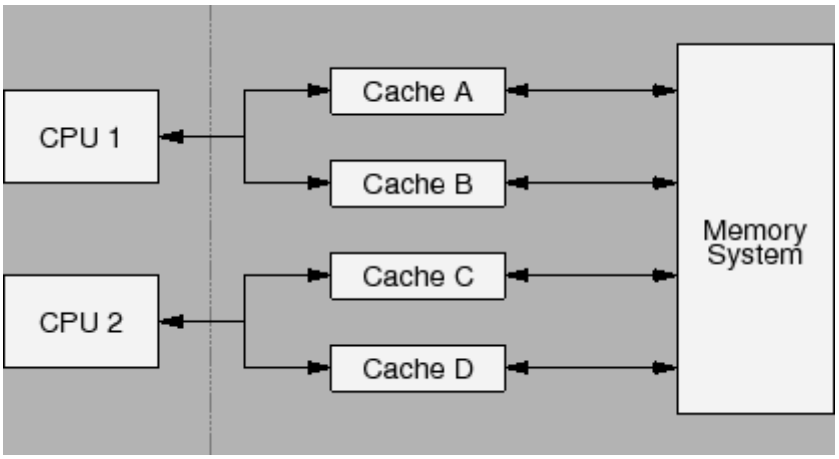


图 12.18: Split Caches

要明白为什么乱序可能发生，考虑如图 12.18 所示的 2-CPU 系统，每一个 CPU 拥有一个分离的缓存，这个系统有以下属性：

- ✓ 奇数编号的缓存行可能在缓存 A、C 中。
- ✓ 偶数编号的缓存行可能在缓存 B、D 中，或者在内存中。
- ✓ 当 CPU 核正在向它的缓存获取数据，它的其他缓存不必处于静止状态。其他缓存可以响应“使无效”请求，回写脏缓存行，处理 CPU 内存访问队列中的元素，或者其他。
- ✓ 每一个缓存都有各自的操作队列，被缓存用来维护请求一致性和顺序属性。

这些队列不必在相应的缓存行元素装载存储时进行刷新。

简单的说，如果缓存 A 忙，但是缓存行 B 空闲，那么与 CPU2 向偶数行存储相比，CPU1 向奇数编号的缓存行存储会被延迟。在不那么极端的情况下，CPU2 就可以看到 CPU1 的操作乱序。

关于硬件、软件方面内存序的更多详情，请参见附录 C。

12.2.14. 哪里需要内存屏障?

仅仅在两个 CPUs 之间或者 CPU 与设备之间需要交互时，才需要内存屏障。如果任何代码能够保证没有这样的交互，那么这样的代码是不必使用内存屏障的。

注意，这是最小的保证。不能体系结构给出了更多保证，具体描述在附录 C。但是，不能将代码设计来只运行在相应的体系中。

但是，正如锁原语和原子数据结构维护原语一样，原子操作的实现通常在它们的定义中包含了必要的内存屏障。但是，有一些例外，例如在 LINUX 内核中的 atomic_inc()。

一个忠告：使用原始的内存屏障应当是最后的选择。使用已经处理了内存屏障的已有原语是更好的选择。

12.3. 非阻塞同步

12.3.1. 简单 NBS

12.3.2. 冒险指针

12.3.3. 原子数据结构

队列和栈—避免竞争非阻塞属性通常大大简化代码。

12.3.4. ``Macho'' NBS

Cite Herlihy 和他的 crowd.

Describe constraints (X-freedom, linearizability, ...) and show examples breaking them.

13. 易于使用

``Creating a perfect API is like committing the perfect crime. There are at least fifty things that can go wrong, and if you are a genius, you might be able to anticipate twenty-five of them."

13.1. Rusty Scale for API Design

It is impossible to get wrong. dwim()

The compiler or linker won't let you get it wrong.

The compiler or linker will warn you if you get it wrong.

The simplest use is the correct one.

The name tells you how to use it.

Do it right or it will always break at runtime.

Follow common convention and you will get it right. malloc()

Read the documentation and you will get it right.

Read the implementation and you will get it right.

Read the right mailing-list archive and you will get it right.

Read the right mailing-list archive and you will get it wrong.

Read the implementation and you will get it wrong. The non-CONFIG_PREEMPT implementation of rcu_read_lock().

Read the documentation and you will get it wrong. DEC Alpha wmb instruction.

Follow common convention and you will get it wrong. printf() (failing to check for error return).

Do it right and it will break at runtime.

The name tells you how not to use it.

The obvious use is wrong. smp_mb().

The compiler or linker will warn you if you get it right.

The compiler or linker won't let you get it right.

It is impossible to get right. gets().

13.2. Shaving the Mandelbrot Set

The set of useful programs resembles the Mandelbrot set (shown in Figure) in that it does not have a clear-cut smooth boundary -- if it did, the halting problem would be solvable. But we need APIs that real people can use, not ones that require a Ph.D. dissertation be completed for each and every potential use. So, we ``shave the Mandelbrot set",15.1restricting the use of the API to an easily described subset of the full set of potential uses.

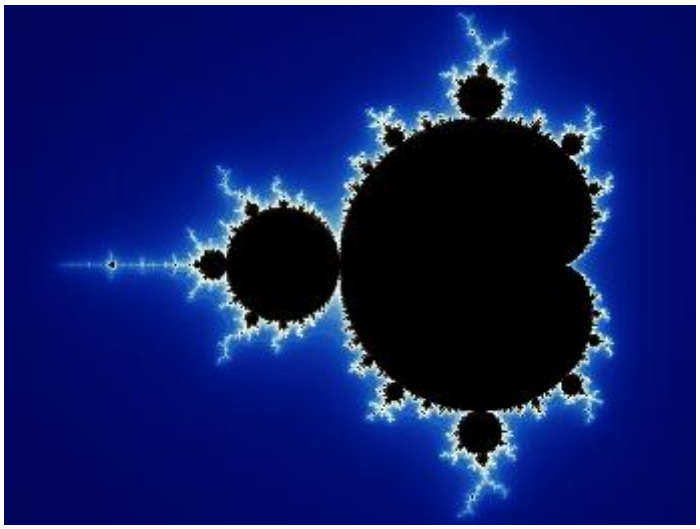


图 13.1: Mandelbrot Set (Courtesy of Wikipedia)

Such shaving may seem counterproductive. After all, if an algorithm works, why shouldn't it be used?

To see why at least some shaving is absolutely necessary, consider a locking design that avoids deadlock, but in perhaps the worst possible way. This design uses a circular doubly linked list, which contains one element for each thread in the system along with a header element. When a new thread is spawned, the parent thread must insert a new element into this list, which requires some sort of synchronization.

One way to protect the list is to use a global lock. However, this might be a bottleneck if threads were being created and deleted frequently.15.2Another approach would be to use a hash table and to lock the individual hash buckets, but this can perform poorly when scanning the list in order.

A third approach is to lock the individual list elements, and to require the locks for both the predecessor and successor to be held during the insertion. Since both locks must be acquired, we need to decide which order to acquire them in. Two conventional approaches would be to acquire the locks in address order, or to acquire

them in the order that they appear in the list, so that the header is always acquired first when it is one of the two elements being locked. However, both of these methods require special checks and branches.

The to-be-shaven solution is to unconditionally acquire the locks in list order. But what about deadlock?

Deadlock cannot occur.

To see this, number the elements in the list starting with zero for the header up to for the last element in the list (the one preceding the header, given that the list is circular). Similarly, number the threads from zero to . If each thread attempts to lock some consecutive pair of elements, at least one of the threads is guaranteed to be able to acquire both locks.

Why?

Because there are not enough threads to reach all the way around the list. Suppose thread 0 acquires element 0's lock. To be blocked, some other thread must have already acquired element 1's lock, so let us assume that thread 1 has done so. Similarly, for thread 1 to be blocked, some other thread must have acquired element 2's lock, and so on, up through thread , who acquires element 's lock. For thread to be blocked, some other thread must have acquired element 's lock. But there are no more threads, and so thread cannot be blocked. Therefore, deadlock cannot occur.

So why should we prohibit use of this delightful little algorithm?

The fact is that if you really want to use it, we cannot stop you. We can, however, recommend against such code being included in any project that we care about.

But, before you use this algorithm, please think through the following Quick Quiz.

Quick Quiz 15.1: Can a similar algorithm be used when deleting elements? End Quick Quiz

The fact is that this algorithm is extremely specialized (it only works on certain sized lists), and also quite fragile. Any bug that accidentally failed to add a node to the list could result in deadlock. In fact, simply adding the node a bit too late could result in deadlock.

In addition, the other algorithms described above are ``good and sufficient''. For example, simply acquiring the locks in address order is fairly simple and quick, while allowing the use of lists of any size. Just be careful of the special cases presented by empty lists and lists containing only one element!

Quick Quiz 15.2: Yetch! What ever possessed someone to come up with an

algorithm that deserves to be shaved as much as this one does??? End Quick Quiz

In summary, we do not use algorithms simply because they happen to work. We instead restrict ourselves to algorithms that are useful enough to make it worthwhile learning about them. The more difficult and complex the algorithm, the more generally useful it must be in order for the pain of learning it and fixing its bugs to be worthwhile.

Quick Quiz 15.3: Give an exception to this rule. End Quick Quiz

Exceptions aside, we must continue to shave the software ``Mandelbrot set'' so that our programs remain maintainable, as shown in Figure .

图 13.2: Shaving the Mandelbrot Set

14. 时间管理

Scheduling ticks

Tickless operation

Timers

Current time, monotonic operation

The many ways in which time can appear to go backwards

Causality, the only real time in SMP (or distributed) systems

15. 未来的冲突

本章描述并行编程在将来的一些可能冲突的地方。事态将如何发展还不太清楚。实际上，任何一项都不太清楚。由于每一种方法都有忠实的信徒，因此它们都显得非常重要。并且，如果很多人强烈的相信某个事情，您将有必要处理这些已经存在的东西。另外，这些方法中的一种或者多种完全有可能真实的发生。

因此，以下章节将给出可交易内存、共享内存并行编程方法，以及基于进程的并行编程方法。

15.1. 可交易内存

这种使用外部数据库进行事务的方法在数十年前就存在了[Lom77]，在使用数据库和不使用数据库之间的关键不同点在于：非数据库交易在定义交易的“ACID”属性中去除了“D”。在硬件方面，支持基于内存的交易方法，或者“可交易内存” (TM)是更新的方法 [HM93]，但不幸的是，支持这样的交易的商业硬件不会立即出现，尽管其他一些类似的计划已经提出[SSHT93]。不久前，Shavit 和 Touitou 提议一个纯软件的交易内存实现 (STM) ，它将能够运行在一般的硬件上。这个提议被忽视了多年，也许是因为研究团体的注意力更专注于非阻塞式同步。

不过风水是轮流转的，TM 开始受到更多关注[MT01,RG01]，在前几年（当前十年的中期）（xie.baoyou 注：应当是指 2005 年左右），只能用 “白热化”来形容人们的兴趣级别 [Her05,Gro07]，尽管还有一些警告的声音存在[BLM05,MMW07]。

TM 的基本思想是：原子的执行一个代码段，这样其他线程将不会看到任何中间状态。同样的，TM 的语义能够通过以重复的获得释放全局锁来替换每一个事务来简单的实现，尽管有不易度量的性能和可扩展性。这些复杂性中的大部分来自于 TM 的实现，不管是软件实现还是硬件实现。当并发事务能够安全的并行运行时，能够有效的检测到这些复杂性。由于这类检测是自动完成的，冲突的事务能够被中止或者回退。在某些实现中，这种冲突对程序员是可见的。

由于事务回退随着事务增大而渐增，TM 对小的基于内存的操作是很有吸引力的。例如象用于堆栈管理的链表维护，队列，哈希表及搜索树。无论如何，对于大事务来说，当前还是难于处理。特别的，这也包含象 IO 及进程创建这样的

非内存操作。以下章节讨论 TM 当前的遇到的挑战 [McK09b].

15.1.1. I/O 操作

有一种基于锁（至少是 RCU 读锁）的临界区能够执行 IO 操作。当您在一个事务中尝试执行一个 IO 操作时，将会发生什么？

潜在的问题是事务能够被回退。例如，由于发生了冲突。粗略的说，这需要在指定的事务中的所有操作是可重复的，这样，两次执行操作与执行一次效果是相同的。不幸的是， I/O 操作通常不是这样，在事务中包含通常的 I/O 操作是困难的。

以下是在事务中处理 I/O 的可选方法：

- ✓ 在事务中，将 I/O 缓存到内存缓冲区。这些缓冲区可以包含在事务中的任何内存位置。这看起来是一个可选的机制，并且可以在绝大多数通常情况下正常运行，例如象流 I/O 或者块 I/O。但是，当来自于多个进程的多个输出流合并到一个单个的文件时，需要特殊处理。如使用“a+”选项调用 fopen 或者 O_APPEND 标志调用 open()。另外，正如在下一节将看到的一样，网络操作通常不能被缓冲。
- ✓ 在事务中禁止 I/O，这样，任何尝试执行 I/O 操作将会中止未完成事务（也许是多个嵌套事务）。这种方法类似于处理非缓存 I/O 的传统 TM 方法。
- ✓ 在事务中禁止 I/O，但是需要编译器的帮助以强制禁止这种行为。
- ✓ 在特定时刻，仅仅允许处理特定的“inevitable”事务 [SMS08]，这样允许 inevitable 包含 IO 操作。这是通常的方法，但是严格的限制 IO 操作的性能和扩展性。而扩展性和性能是并行编程的首要目的。更糟糕的是，使用这种事务以允许 I/O 操作看起来禁止了手动终止事务操作。
- ✓ 创建新的硬件和协议，这样 I/O 操作能够被放入事务底层。在输入操作时，硬件需要正确的预见操作结果，如果预见失败就中止事务。

I/O 操作是 TM 最明显的缺点，目前还不清楚，在事务中支持 IO 是否有一个合理的综合解决方案。至少，合理的要求是包含一个可用的性能和可扩展性。

15.1.2. RPC 操作

能够在一个基于锁的临界区执行 PRCs。尝试在一个事务中执行一个 PRC 会发生什么？

如果 RPC 请求和它的响应被包含在同一个事务中，并且事务的某些部分依赖于响应的返回结果，那么就不可能使用在缓冲 IO 时用到的内存缓存区这种技

巧。任何尝试采用这种缓冲方法的努力将会导致事务产生死锁。直到事务完成前，RPC 请求才能发送，但是直到接收到请求前，事务不能成功，有如下例子：

```
1 begin_trans();
2 rpc_request();
3 i = rpc_response();
4 a[i]++;
5 end_trans();
```

直到接收到 RPC 请求的响应后，事务的内存地址才能被确定下来；直到请求的内存地址被确定后，才能确保是否提交事务。

以下是 TM 可用的选项：

- ✓ 在事务中禁止 RPC，任何试图执行一个 PRC 操作都将中止未完成的事务（也许是多个嵌套事务）。可选的，得到编译的帮助以强制禁止 RPC。这种方法能够工作，但是需要 TM 与其他原始同步机制交互。
- ✓ 在任何特定时刻，仅仅允许一个特定的“inevitable”事务 [SMS08] 开始执行，这样允许 inevitable 事务包含一个 RPC 操作。通常情况下是可行的，但是严重的限制了 RPC 操作的扩展性和性能。而扩展性和性能是并行编程的首要目标。而且，使用 inevitable 事务以容许 RPC 操作，这就不允许在 RPC 操作开始后手动中止事务。
- ✓ 在接收到 RPC 事务响应前，标示事务的特殊情形，并在发送 RPC 请求前，自动的将事务转换成 inevitable 事务。当然，如果几个并发事务尝试调用 RPC，则可能需要回退所有事务，仅仅保留一个事务。这必然降低性能和可扩展性。然而，这种方法对长的、以 RPC 结束的事务是有价值的。这种方法对手动中止事务来说也是存在问题的。
- ✓ 当 RPC 可以被移出事务时，进行特殊标识。然后使用类似于缓冲 I/O 的技术进行处理。
- ✓ 扩展事务底层，使得 RPC 服务器端也象客户端那样很好的运行。从理论上讲，这是可行的。并且已经被分布式数据库证明。但是，尚不能确定性能和扩展性是否能够满足分布式数据库技术。基于内存的 TM 不能解决慢速磁盘设备的延迟问题。当然，随着 solid-state 磁盘的出现，对于大型数据库来说，这也不能确定。

正如前一节的警告一样，I/O 是 TM 的一个有名的缺点，RPC 是其中一个特别突出的问题。

15.1.3. 内存映射操作

在一个基于锁的临界区内执行内存映射操作（包括 mmap()，shmat()和

munmap()[Gro01]）是完全合法的。当您在一个事务中尝试执行这样的操作会发生什么？更进一步说，在当前线程的事务中，重新映射已经包含了某些变量的区域会发生什么？如果内存区域包含其他线程事务的变量时会发生什么？

不用特别注意考虑 TM 系统区域被重新映射的情况。

以下是一些内存映射可用选项：

- ✓ 在一个事务中进行内存重新映射是不合法的。并且会中止当前未完成的任务。这显然稍微简单了一点。
- ✓ 在一个事务中进行内存重新映射是不合法的，并且需要编译协助确保这个限制。
- ✓ 在一个事务中进行内存重新映射是合法的，但是在影响区域范围内的其他事务将被中止。
- ✓ 在一个事务中进行内存重新映射是合法的，但是如果映射区域与当前任务的区域有重叠，则映射将失败。
- ✓ 所有内存映射操作，不管是否处于一个事务内，都将检查相应的区域是否与系统内其他事务的区域重叠。如果重叠，则内存映射将失败。

TM 冲突管理机制将检查内存映射对其他事务的影响，它动态的确定是否让内存映射失败，还是让冲突的事务中止。

要注意 munmap() 删除了内存映射，将有更多的问题。

15.1.4. 多线程事务

当获取一个锁的时候，创建进程或者线程是完全合法的。这不仅合法，而且非常简单，如下面代码所示：

```
1 pthread_mutex_lock(...);
2 for (i = 0; i < ncpus; i++)
3     tid[i] = pthread_create(...);
4 for (i = 0; i < ncpus; i++)
5     pthread_join(tid[i], ...)
6 pthread_mutex_unlock(...);
```

这段伪代码使用 pthread_create()为每一个 CPU 创建一个线程，然后使用 pthread_join()等待每一个线程结果运行。整个过程处于 pthread_mutex_lock()保护之下。类似的，也可以用 fork() 和 wait()。当然，临界段需要需要十分大，但是在产品中确实存在大临界区的例子。

在事务中进行线程创建线程，关于这点 TM 能做些什么?

- ✓ 在事务中执行 pthread_create() 是非法的，将导致事务被中止或者发生不可预计的后果。可选的，借助于编译器确保在事务中不进行 pthread_create()。

- ✓ 允许在事务中执行 pthread_create() ，但是仅仅允许父线程这样做。这种方法看起来与已有的 TM 实现相当一致。但是稍不留意就会掉进坑里。这种方法带来了更多的问题，比如如何处理子线程访问冲突。
- ✓ 将 pthread_create()函数调用。这个方法也有问题。 它不能处理一些罕见的情况，如子线程与另外的线程通信。这个方法也带来一些冲突访问方面的问题，它不允许事务事务并行执行。
- ✓ 扩展事务以包含父线程和它的子线程。这个方法带来访问冲突方面的问题。也许父线程和子线程之间允许冲突发生，但是在子线程之间却不一定这样。它还会带来一些其他问题：在提交事务前，如果父线程不等待子线程将会发生什么？还有更多问题：如果父进程根据事务中的变量值作为判断条件，决定是否调用 pthread_join()将发生什么？ 这些问题的答案是相当直截了当的。TM 的答案就留给读者了。

数据库世界的并行执行是相当常见的。奇怪的是，当前的 TM 计划没有为此而提供。另一方面，上面的例子是非常怪异的用法，这在一般的简单教科书例子中是找不到的，也许是希望被忽略。也就是说，TM 研究者有兴趣在事务中使用 fork/join 是骗人的。也许这个主题将很快被解决。

15.1.5. 外部的事务访问

在基于锁的临界段中，并发访问变量，甚至在锁保护的临界区外面修改变量都是十分合法的。一个常见的例子是统计计数。相同的事情可能在 RCU 读锁保护的临界区中发生，并且这是更常见的情况。

在实际的数据库系统产品中，采用一种名为``dirty reads"的机制。外部的事务访问已经受到 TM 的关注，这不令人意外。

以下在 TM 中可选的外部事务选项：

- ✓ 一旦外部事务访问冲突，总是导致事务被中止。这是强原子性。
- ✓ 一旦外部事务访问冲突就被忽略。因此仅仅在事务内的冲突才中止事务。这是弱原子性。
- ✓ 在特殊情况下，事务允许被完成。例如，分配内存时，或者在基于锁的临界段中。
- ✓ 扩展硬件以允许完成一些操作以完成在多个事务中对单个变量的并发访问。

看起来反对这样的事务：没有交互需要时使用其他同步机制。这样的话，当同时有非事务访问和事务时，有更多的复杂性和混乱就不足为奇了。但是除非限制事务对独立的数据结构进行小的修改，或者限制新程序不与大的已经存在的并

行代码进行交互。那么在一些拥有大的并行冲突的项目中，事务必须这样组合。

15.1.6. 延时

一个重要的特殊情况是：包含外部事务访问的事务包括延时。当然，在事务中进行延时的思路公然违抗了 TM 的原子属性。但是弱原子性如何解释呢？ 而且，与内存映射 I/O 交互有时需要小心的控制时间。而且应用程序常常由于不同的目的需要延时。

关于在事务中进行延时的问题，TM 能做一些什么呢？

- ✓ 在事务中忽略延时。这看起来很不错，但是象其他“好”方法一样，包含正确逻辑的代码不能正确工作。这样的代码，可能在关键代码段中需要重要的延时，在事务中则会失败。
- ✓ 遇到延时操作时，中止事务。这是比较吸引人的方法，但是不幸的是，并不是总能自动检测到延时操作。一些重要的计算是循环操作，有时又将循环用于延时以等待一段时间的消逝。
- ✓ 让延时操作正常运行。不幸的是，一些 TM 仅仅在发布时才修改延时方面的实现。

不再清楚是否有一个唯一的正确答案。弱原子 TM 实现最终可能可选的。即使在这种情况下，事务中的代码可能需要重新实现，以允许中止事务。

15.1.7. 锁

在获得一个锁时，通常也需要其他锁。这工作得很好，至少软件工程师在利用某些技术避免了死锁时是这样的。在 RCU 读临界区中申请锁是不常见的。当您尝试在一个事务中申请锁会发生什么？

从原理上讲，答案是微不足道的：简单的维护数据结构，将锁作为事务的一部分就行了，这也会工作得很好。实际上，有一些不明显的复杂因素在里面，这依赖于 TM 系统的详细实现。这些复杂因素可以被解决，但是其代价是在事务以外增加 45%的锁请求，在事务内增加 300%的锁请求。虽然在包含少量锁的事务中，这是可以接受的，但是在一些基于锁的产品中，这常常是完全不能接受的。

- ✓ 仅使用锁友好的 TM 实现。不幸的是，非锁友好的 TM 实现有一些吸引人的特性，包括成功交易的低负载，以及提供非常大的交易能力。
- ✓ 在基于锁的程序中，仅仅使用少数的 TM。以此来适应锁友好型的 TM 实现的限制。
- ✓ 完全撇开基于锁的遗留系统，用事务来重新实现所有东西。这种方法不缺少支持者。但是这需要解决这里列出的所有问题。由于解决这些问题

需要时间，因此同步机制还有改进的机会。

在基于锁的系统中，仅仅使用 TM 作为优化手段，在 TxLinux [RHP+07]中已经这样做了。这种方法看起来是合理的，但是存在锁设置约束（如需要避免死锁）。

✓ 尽量减少负载。

TM 和锁之间的接口可能有一些问题，这对很多人来说是令人惊奇的。需要强调一点：有必要在新机制和原有机制之间进行实验。幸运的是，开源的出现使得有大量这样的软件可用了。

15.1.8. 读者-写者锁

在获得其他锁的时候，请求读锁是很平常的。在工程师使用正确的技术避免死锁的时候，它是能够正确的工作的。在 RCU 读锁保护的临界区内，申请读锁也是可以的。不要担心死锁，因为 RCU 读锁不会导致引起死锁 。但是当您在一个事务中申请读锁会发生什么事情呢？

不幸的是，直接在事务中使用传统的基于计数的读写锁达不到读写锁的目的。要明白这一点，考虑一对事务并发的尝试申请同一个读锁。由于申请读锁会修改读写锁数据结构，这将产生冲突，并回退两个事务中的其中一个。这与读写锁允许多个读者的目的完全不相符。

以下是 TM 中的一些可用的选项：

✓ 使用每 CPU 或者每线程的读写锁 [HW92]，这允许特定的 CPU（或者线程）在申请读锁时仅仅维护局部数据这将避免两个并发事务之间在申请锁时产生冲突。不幸的是： (1) 每 CPU/线程写锁的的代价非常高，(2) 每 CPU/线程的内存代价可能不能得到满足，(3) 仅仅当您能够访问源代码时，这种变化才是可行的。最近的可扩展读写锁[LLO09]可以避免这些问题。

✓ 在基于锁的程序中，少量的使用 TM，这样可以避免在事务中使用读锁。

✓ 完全撇开基于锁的遗留系统。使用事务重新实现所有东西。这个方法有不少支持者。

✓ 在基于锁的系统中，仅仅将 TM 作为一个优化措施，正如 TxLinux [RHP+07] 一样。这种方法看起来不错，但是系统中仍然存在锁设计约束。此外，当多个事务尝试申请同一个读锁时，会产生不必要的回退。

当然，结合使用 TM 和读锁可能有一些其他不明显的问题，它事实上将读锁变成排它锁。

15.1.9. 持续性

存在不同类型的锁。一个有趣的差别是持续性。换句话说，在使用锁的进程中，锁是否能够独立的存在。

非持续性锁包括 pthread_mutex_lock()、pthread_rwlock_rdlock()，以及大多数内核级别的锁。典型的，使用 pthread_mutex_lock()，这意味着当进程退出时，它的所有锁将消失。为了在程序退出时自动的清理锁，可以利用这个特性。但是不同进程共享锁使得这件事情变得更加困难。这需要在应用间共享内存。

持续性锁帮助避免在无关应用之间共享内存。持续性锁 APIs 包括 flock 系列，lockf()，System V 信号量，以及 O_CREAT 标志的 open()调用。这些持续性 APIs 能够用于保护多应用之间的操作。并且，在 O_CREAT 这种情况下，甚至能在操作系统重启后有效。通过分布式锁管理，锁能够跨越多计算机系统。

持续性锁能够用于任何应用，包括使用不同语言及开发环境编写的应用。实际上，一个持续性锁可以被一个 C 语言编写的应用正确的获得，并被一个以 Python 编写的应用释放。

TM 如何提供类似的持续性功能？

- ✓ 限制持续性交易在特定目标环境，如 SQL。数据库系统对此的支持已经有超过十年的历史了。但是持续性锁不提供相同的灵活性性。
- ✓ 使用由一些存储设备和文件系统提供的快照功能。不幸的是，这不能处理网络通信，在不提供快照能力的设备（如内存记忆棒）上，也不处理 IO。
- ✓ 构建某种时间机器。

当然，事务内存这个称呼应当暂停，这个名称与可持续性事务的概念是冲突的。

15.1.10. 动态链接装载

基于锁的临界段以及 RCU 读保护的临界段都可以合法的包含调用动态链接装载的代码，包括 C/C++共享库和 Java 类库。当然，包含在这些库中的代码在编译时是未知的。因此，如果在事务中调用动态加载的函数将会发生什么？

这个问题包含两个部分： (a) 在事务中如何动态链接并装载函数(b) 面对在事务中包含的未知代码，您将做什么?为清楚起见，第二项中包含一些挑战锁和 RCU 的东西。例如，动态链接函数可能导致死锁，或者在 RCU 读保护的情况下，经过一次“静止”状态（注：经过静止状态后，RCU 保护的东西将可能失效）。不同之处在于：在锁保护及 RCU 保护的临界段中，允许操作的类型是不同的。

实际上，不同的 TM 实现有不同的限制。

在动态链接装载方面，TM 能做些什么呢？对一个问题来说，包含以下可选项：

- ✓ 将动态链接和装载视同缺页异常。这样，链接装载函数可能中止事务。重试时，将会发现函数已经存在，因此事务能够正常处理。
- ✓ 在事务内不允许动态链接加载 。

对第二个问题来说，不能检测尚未装载函数中的 TM 不友好操作，可能的选项包括以下几方面：

- ✓ 执行相应的代码：如果有 TM 不友好的操作，则简单的中止事务。不幸的是，这个方法使得编译器无法检测一组指定的事务是否可以安全的组合在一起。一个方式是总是允许将 inevitable 事务组合在一起。但是，当前的实现是仅仅允许在一个指定时间处理一个 inevitable 事务。这将限制性能和可扩展性。Inevitable 事务也排除了手动中止事务操作。
- ✓ 通过函数修饰符标示函数是 TM 友好的。这可以由编译器来强制标识。当然，对很多语言来说，这需要扩展语言。并且需要一定的时间。也就是说，标准化已经在进行中了 [ATS09]。

如上所述，在事务中不允许动态链接与装载。

I/O 当然是 TM 的一个有名的弱项，动态链接装载也可以看作一种特殊的 I/O。然而，TM 建议者必须要么解决这个问题，要么认命，将 TM 作为并行编程者工具箱中的其中一个(已经有很多建议者是这样的了) 。

15.1.11. 调试

通常的调试操作（如断点）可以在基于锁或者 RCU 读锁这样的临界区中是可以正常工作的。但是，在事务内存硬件执行一个异常将中止事务，这意味着断点将终止未完成事务。

将如何调试事务呢？

- ✓ 在包含断点的事务中使用软件模拟技术。
- ✓ 仅仅使用能够处理断点异常的硬件 TM 实现。不幸的是，目前（2008.09）所有这样的实现仅仅是研究原型。
- ✓ 仅仅使用软件 TM 实现，这比硬件 TM 实现更能容忍异常。当然，软件 TM 比硬件 TM 的消耗要高，因此这个方法可能不会被接受。
- ✓ 更小心的编程，这样以避免在事务中产生 BUG。一旦您已经理解这点，请让所有人都知道这个秘密！

有一些理由让人相信事务内存将在其他同步机制的基础上提高生产力，但是

如果传统的调试技术不能应用到事务上的话，可能轻易的失去这种对生产力的提高能力。特别是新手在编写大的事务时，更是这样。相对的，“top-gun”宏程序员也许能够不需要这些调试帮助，特别是小事务。

因此，如果事务内存是为了提高新手程序员的生产效率，那么调试问题就需要解决。

15.1.12. exec() 系统调用

在持有锁时能够执行 exec()系统调用，也能在持有 RCU 读锁时执行。

在非持续性情况下 (包括 pthread_mutex_lock(), pthread_rwlock_rdlock()及 RCU)，如果 exec()成功了，整个地址空间就消失了，并且是在持有锁的情况下地址空间消失了。当然，如果 exec() 执行失败，地址空间仍然存在，因此相关的锁也存在。

另一方面，持续性情况下 (包括 flock 系列, lockf(), System V 信号量, 以及带 O_CREAT 标志的 open()) 将是幸运的，而不管 exec()是否成功。

问题 17.1：在非持续性情况下，mmap 范围内的内存数据结构将如何表现? 在非持续性情况下，在临界区中存在 exec 将发生什么?

当您在事务中执行一个 exec()系统调用将会发生什么？

- ✓ 允许在事务中包含 exec() ，因此在事务中执行 exec()将中止事务。
- ✓ 由编译器确保在事务中不会包含 exec() 。有一个针对 TM 的 C++草案处理这点。这允许标示函数是否是事务 TM 安全。与在运行时中止事务相比，这个方法有一些优势。
- ✓ 将事务视为非持续性锁，这样在 exec()失败的时候，事务还会侥幸运行。如果 exec()提交成功，则事务不会提交。被 mmap()影响到的变量的情况，留给读者作为练习。
- ✓ 当 exec()系统调用将要成功时，终止事务和 exec()系统调用。但是当 exec()系统调用将要失败时，则继续运行事务。这看起来是正确的方法，但是需要做的工作比较多。

exec()系统调用对于一般的 TM 应用来说，也许是一个奇怪例子。采用哪种解决方法目前还不是完全清楚。在事务中禁止 exec 也许是最合乎逻辑的。

15.1.13. RCU

由于 RCU 主要用于 LINUX 内核，因此即使人们认为结合 RCU 和 TM 没有理论工作要做，这是可以理解的。但是，来自 Texas 大学 TxLinux 组没有选择，

不得不接手这些理论工作 [RHP+07]。事实上，他们在 Linux2.6 内核中应用 TM。由于内核使用了 RCU，这使得他们需要集成 TM 和 RCU。不幸的是，虽然文献上说明 RCU 实现的锁 (如 rcu_ctrlblk.lock)已经转换为事务，但是没有说在基于 RCU 的更新中，使用锁会发生什么问题。 (如 dcache_lock).

重要提醒：RCU 允许读者和写者并行运行， 更准确的说，允许数据在更新的时候，RCU 读者对它进行访问。当然，这个 RCU 属性可能对它的性能，可扩展性，实时响应都是非常重要的。

基于 TM 的更新如何与并发的 RCU 读者交互？有如下可能性：

- ✓ RCU 读者终止并发的更新冲突的 TM 事务。这是 TxLinux 项目事实上采用的方法。这个方法保护 RCU，也保护 RCU 读者的性能，可扩展性及实时响应属性。但是不幸的是，它不必要的中止了与之冲突的更新操作。在最坏的情况下， 一个长时间运行的读者会导致所有更新者产生饥饿现象，理论上，这会导致系统挂起。另外，不是所有的 TM 实现都提供强原子性以实现这个方法。
- ✓ 在与 RCU 更新冲突时，RCU 读者获得旧的值。这保持了 RCU 的语义，及其性能，也保护了 RCU 更新免受饥饿的影响。但是，并不是所有的 TM 实现都能够提供及时的访问旧值的功能。准确的说，基于日志的 TM 实现在日志中维护旧值，在这种情况下，它工作得不是很好。也许 rcu_dereference() 能够允许 RCU 在大部分 TM 实现中访问旧值，虽然性能可能会是一个问题。
- ✓ 如果一个 RCU 读者与一个事务产生了冲突访问，那么 RCU 访问被延迟，直到冲突的事务被中止或者被提交。这个方法保护了 RCU 的语义，但是不保护 RCU 的性能和实时响应能力，特别是存在长时间运行的事务时。另外，并不是所有的 TM 都有延时冲突访问的能力。也就是说，这个方法看起来对支持小事务的硬件 TM 实现是非常合理的。
- ✓ 将 RCU 转换为事务。这个方法可以保证与任何 TM 实现都可以非常好的兼容，但是使得 RCU 读锁保护的临界段也受到 TM 回退的影响。使得 RCU 失去了实时响应的保证。也降低了 RCU 读的性能。而且，在 RCU 读临界区包含 TM 实现所不能处理的情况时，这个方法也是不可行的。
- ✓ 很多地方使用 RCU 写更新修改一个单指针，以提供一个新的数据结构。在这种情况下，能够安全的允许 RCU 看到这个指针，即使随后事务被回退。只要事务遵守内存序，并且回退过程使用 call_rcu()释放相应的数据结构。不幸的是，并不是所有的 TM 实现都在事务中遵守内存序。显然，这个想法是假设事务是原子的，不能假设事务的访问序适合这一点。

- ✓ 禁止在 TM 中使用 RCU 更新。这确保能够正常工作，但是看起来有一些限制。

看起来将有其他的方法出现，特别是出现用户态 RCU 时。

15.1.14. 讨论

对于常见的 TM 问题，有以下结论：

一个有趣的 TM 属性是，事务受到回退以及重试的影响。在 TM 事务存在不可逆操作时，它成为 TM 的困难的基础。包括不可缓冲的 I/O，RPCs，内存映射操作，延时，以及 exec()系统调用。这也带来了不幸的后果：带来了固有的复杂性。

TM 另外一个有趣的特性，由 Shpeisman 提出[SATG+09]，是 TM 与它所保护的数据之间的同步。这是 TM IO、内存映射操作、额外事务访问、调试断点等问题的基础。与之相对的是，传统的同步原语，包括锁和 RCU，在它们要保护的数据与同步原语之间是完全分开的。

TM 领域中，很多工作者的目标之一是：使大的顺序代码段变得易于并行编程。同样的，通常希望独立的事务连续运行，在多线程事务方面，需要做很多事情。

TM 研究者和开发者在这方面将会做些什么呢？

一个方法是关注小型化的 TM，关注于通过硬件提供优化于其他同步原语的实质性的优势。这实际上是 SUN 在它的 Rock CPU 上采用的方法 [DLMN09]。一些研究者看起来接受这个方法，但是其他研究者对 TM 寄予了更高的期望。

当然，TM 十分有可能用于大型的程序，本节列出了实现此目的需要解决的一些问题。

15.2. 共享内存并行编程

15.3. 基于任务的并行编程

A. 重要问题

随后的章节讨论一些与 SMP 编程相关的重要问题。每一节也展示了如何避免需要小心处理的一致性问题。如果您的目标是使您的 SMP 代码既快又好的运行，那么这一点是非常重要的。

虽然这些问题的答案与单线程之上的直观感觉有点不一样，但是稍微努力一点，它们都不难理解。如果您正在设法解决递归问题，那么这里不会提供任何有用的帮助，那是另一个挑战性问题了。

A.1 “after“的含义是什么?

“After”是一种直观上的感受，但是非常令人惊讶的是，它实际上是一个非常难于理解的概念（xie.baoyou 注：记得某位物理学说曾经说过，他花了几十年时间才理解作用力和反作用力的概念，也许与此很类似。“after”的概念对理解并行编程很重要）。一个重要的非直观上的问题是：代码可能在任何地方，被延迟任意长的时间。我们考虑一个生产者和消费者线程使用一个全局结构进行通信，这个全局结构包含一个时间片“t”和整型字段“a”、“b”、“c”。生产者循环的记录当前时间（自 1970 年以来的秒数），然后更新“a”、“b”、“c”的值。如下图 A.2。最后，消费者输出一个不规则的列表：时间看起来在倒退。

```
1 /* WARNING: BUGGY CODE. */
2 void *producer(void *ignored)
3 {
4 int i = 0;
5
6 producer_ready = 1;
7 while (!goflag)
8 sched_yield();
9 while (goflag) {
10 ss.t = dgettimeofday();
11 ss.a = ss.c + 1;
12 ss.b = ss.a + 1;
13 ss.c = ss.b + 1;
14 i++;
15 }
16 printf("producer exiting: %d samples\n", i);
17 producer_done = 1;
```

```
18 return (NULL);
19 }
```

图 A.1: ``After'' 生产者函数

```
1 /* WARNING: BUGGY CODE. */
2 void *consumer(void *ignored)
3 {
4 struct snapshot_consumer curssc;
5 int i = 0;
6 int j = 0;
7
8 consumer_ready = 1;
9 while (ss.t == 0.0) {
10 sched_yield();
11 }
12 while (goflag) {
13 curssc.tc = dgettimeofday();
14 curssc.t = ss.t;
15 curssc.a = ss.a;
16 curssc.b = ss.b;
17 curssc.c = ss.c;
18 curssc.sequence = curseq;
19 curssc.iserror = 0;
20 if ((curssc.t > curssc.tc) ||
21 modgreater(ssc[i].a, curssc.a) ||
22 modgreater(ssc[i].b, curssc.b) ||
23 modgreater(ssc[i].c, curssc.c) ||
24 modgreater(curssc.a, ssc[i].a + maxdelta) ||
25 modgreater(curssc.b, ssc[i].b + maxdelta) ||
26 modgreater(curssc.c, ssc[i].c + maxdelta)) {
27 i++;
28 curssc.iserror = 1;
29 } else if (ssc[i].iserror)
30 i++;
31 ssc[i] = curssc;
32 curseq++;
33 if (i + 1 >= NSNAPS)
34 break;
35 }
36 printf("consumer exited, collected %d items of %d\n",
37 i, curseq);
38 if (ssc[0].iserror)
39 printf("0/%d: %.6f %.6f (%.3f) %d %d %d\n",
40 ssc[0].sequence, ssc[j].t, ssc[j].tc,
```

```
41 (ssc[j].tc - ssc[j].t) * 1000000,
42 ssc[j].a, ssc[j].b, ssc[j].c);
43 for (j = 0; j <= i; j++)
44 if (ssc[j].iserror)
45 printf("%d: %.6f (%.3f) %d %d %d\n",
46 ssc[j].sequence,
47 ssc[j].t, (ssc[j].tc - ssc[j].t) * 1000000,
48 ssc[j].a - ssc[j - 1].a,
49 ssc[j].b - ssc[j - 1].b,
50 ssc[j].c - ssc[j - 1].c);
51 consumer_done = 1;
52 }
```

图 A.2：“After”消费者函数

问题 A.1：在这个例子中，您能够看出哪些 SMP 代码错误？完整的代码请参见 time.c。

直观上，我们可能觉得生产者和消费者之间的时间戳差异应该非常小。因为生产者记录时间戳和整型值用不了多少时间。在一个双核 1GHZ x86 机器上的输出结果如表 A.1.。在这里，“seq”列是循环次数，“time”列是第二个 CPU 中看到的不规则时间，“delta”列是消费者的时间戳与生产者的时间戳之间的差异（注：从直观上来说，应该为正数）。 ``a", ``b", ``c" 列显示生产者读取到的值与前一次读到的值之间的变化。

表 A.1: ``After" 示例输出结果

seq	time (seconds)	delta			
17563:	1152396.251585	(-16.928)	7	7	7
18004:	1152396.252581	(-12.875)	4	4	4
18163:	1152396.252955	(-19.073)	8	8	8
18765:	1152396.254449	(-1 8.773)	16	16	16
19863:	1152396. 56960	(-6.914)			

			8	8	8
21644:	1152396.260959	(-5.960)	8	8	8
23408:	1152396.264957	(-20.027)	5	5	5

为什么时间会倒退呢?括号中的值是以纳秒为单位的差异值。大的超过了 10 纳秒。有一个竟然超过了 100 纳秒！请注意，在这段时间内，CPU 可能执行超过 100，000 条指令（注：100，000 条指令不太正确，100 纳秒应当是 100 条指令，可能是作者单位换算错误）！

可能的原因如下：

- ✓ 生产者获得时间戳 (图 A.2，第 13 行).
- ✓ 消费者被抢占
- ✓ 过去了一段时间.
- ✓ 生产者获得时间戳 (图 A.2，第 10 行).
- ✓ 消费者重新执行，并读取到生产者的时间戳 (图 A.1，第 14 行).

在这种情况下，生产者的时间戳可能位于消费者的时间戳后面任意一段时间。

您如何在 SMP 代码中免受这样的问题困扰呢？

简单的使用 SMP 原语就可以了。

在本例中，最简单的修复办法是使用锁，例如，在生产者代码中，在第 10 行前获得一个锁(图 A.1)，并且，消费者在第 13 行前获得一个锁(图 A.2)。在第 13 行和第 17 行后，也必须释放这个锁。这些锁导致第 10-13 行以及第 13-17 行的代码互斥。换句话说，它们相互之间原子的执行。如图 A.3 所示：锁使得代码看起来象一个盒子，因此消费者的时间戳必须在前一个生产者的时间戳之后获得。图中的代码段被称为“临界段”。在同一时刻，只能有一个这样的临界段能够执行。

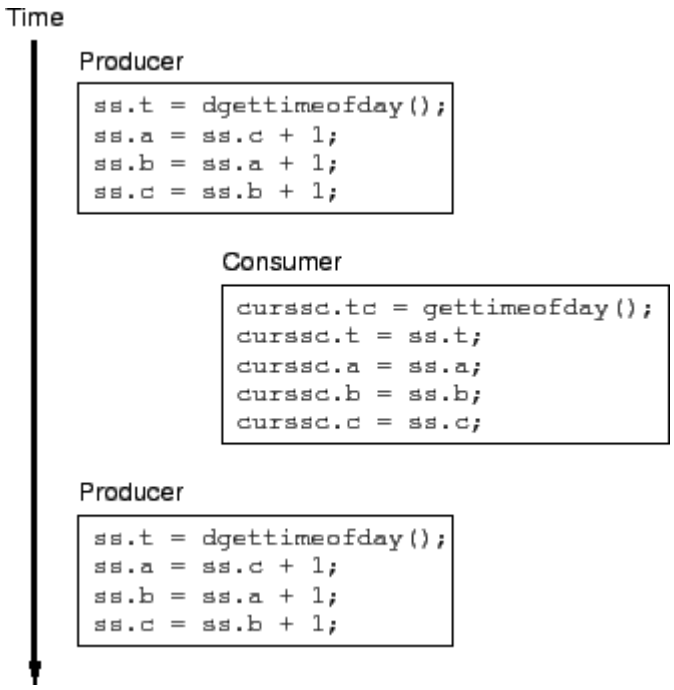


图 A.3：使用锁的效果

使用了锁的输出结果如下表 A.2。这里再没有时间倒退的情况发生。仅仅有这样的情况发生：消费者两次读到的计数差值超过了 1，000。

表 A.2: Locked ``After'' Program Sample Output

seq	time (seconds)	delta	a	b	c
58597:	1156521.556296	(3.815)	1485	1485	1485
403927:	1156523.446636	(2.146)	2583	2583	2583

问题 A.2：为什么消费者读到的数据有这么大的差值？完整的代码请参考文件 timelocked.c。

简短的说，如果你申请一个互斥锁，您在获得一个锁时，所做的任何事情都只能在前一个获取到锁的代码执行完成之后才能发生。不用考虑哪一个 CPU 是否执行内存屏障。也不用考虑 CPU 或者编译器乱序操作。当然，这个锁操作会限制两个并行运行的代码段在多核上获得性能上增加，可能的结果是“安全但是慢”。第 5 章描述了增加性能和可扩展性的方法。

但是，在很多情况下，如果您担心在一个特定的代码段前后将会发生什么的话，您应当使用标准的同步原语，让这些原语解除您的担心吧 。

B.同步原语

除了最简单的并行程序外，其他所有并行程序都需要同步原语。这个附录给出一个 LINUX 内核中的同步原语概要。

为什么是 LINUX？因为它是一个著名的、大的、容易获得的并行代码。我们坚信：读代码是比写代码更重要的学习方法，因此，使用类似于 LINUX 内核的代码，能够让您在本书的基础上更进一步的学习并行编程。

为什么不严格的使用 LINUX API？首先，LINUX API 会随着时间而变化，因此尝试跟踪它将会遇到很多问题。其次，很多 LINUX 内核 API 的成员专门用于产品化的操作系统内核。这引入了一些复杂性。虽然，这种复杂性对于 LINUX 内核自身来说是必要的，但是对业余编程来说，多少会产生一些困扰。例如，适当的错误检查（如内存耗尽）在 LINUX 内核中是必须的，但是业余程序完全可以中止程序，或者返回。

最后，在这些 API 和产品级 API 之间，应当定义一个映射层。一个 pthread 实现是可用的 (CodeSamples/api-pthreads/api-pthreads.h)，但是难于建立一个 LINUX 内核模块。

问题 B.1：给出一个不使用同步原语的并行编程例子

以下章节描述常用的同步原语类别。 @@@ 更多深奥的原语将在以后的版本中介绍。

B.1 节包含初始化原语；B.2 节描述线程创建、销毁以及控制原语；B.3 节描述锁原语；B.4 节描述每线程及每 CPU 变量原语；B.5 节给出了几个原语与性能的关系。

B.1 初始化

@@@ 当前包含 ../api.h, 并且仅仅只有 pthread 版本. Expand and complete once the CodeSamples structure settles down.

B.1.1 smp_init()

在调用其他原语前，您必须先调用 smp_init()。

B.2 线程创建、销毁及控制

这些 API 关注“threads”。每一个这样的线程拥有一个类型为 thread_id_t 的标识符，在同一时刻，不会存在两个同时运行的线程拥有相同的标识符。线程共享所有内容，但是不共享每线程状态，包含 PC 指针及堆栈。

线程 API 如图 B.1 所示，其成员在随后的章节中描述。

```
int smp_thread_id(void)
thread_id_t create_thread(void *(*func)(void *), void *arg)
for_each_thread(t)
for_each_running_thread(t)
void *wait_thread(thread_id_t tid)
void wait_all_threads(void)
```

图 B.1: Thread API

B.2.1 create_thread()

create_thread 原语创建一个新线程，第一个参数 func 是线程开始执行的地方。第二参数是传递给它的参数。当 func 返回的时候，线程将退出。create_thread() 原语返回被创建的子线程的标识符。

当超过 NR_THREADS 个线程被创建时，这个原语将中止程序。NR_THREADS 是一个编译时常量，并且可以被修改。某些系统存在可运行的线程数限制。

B.2.2 smp_thread_id()

由于 create_thread() 返回的 thread_id_t returned 是系统相关的，smp_thread_id() 原始返回与线程相关的索引号。这个索引号保证小于自系统启动以来，创建的所有线程总数。因此可用于位图、数组索引以及类似的地方。

B.2.3 for_each_thread()

for_each_thread() 遍历所有已经存在的线程，包含所有已经创建即将存在的线程。这个对处理每线程变量来说是有用的。

B.2.4 for_each_running_thread()

for_each_running_thread()宏仅仅遍历当前存在的线程。与线程创建、删除之间的同步是调用者的责任。

B.2.5 wait_thread()

wait_thread()原语等待指定的线程完成。没有方法干涉指定的线程执行。相反的，它仅仅是等待它。注意 wait_thread()的返回值是相应的线程的返回值。

B.2.6 wait_all_threads()

wait_all_thread()原语等等当前所有运行线程完成。与线程创建、删除之间的同步是调用者的责任。但是，这个原语通常用于清除工作，因此通常不必需要同步。

B.2.7 用法示例

图 B.2 显示了类似于 hello-world 的子线程例子。每个线程有独自的堆栈，因此每个线程有它私有的参数及 myarg 值。每个子线程在退出前简单的打印它的参数和 smp_thread_id() 。注意在第 7 行的返回语句终止了线程，它将 NULL 返回给调用 wait_thread()的调用者。

```
1 void *thread_test(void *arg)
2 {
3 int myarg = (int)arg;
4
5 printf("child thread %d: smp_thread_id() = %d\n",
6 myarg, smp_thread_id());
7 return NULL;
8 }
```

图 B.2: 子线程示例

父程序在图 B.3 中显示。它在第 6 行调用 smp_init()初始化线程系统，在第 7-14 行解析参数。它在第 16-17 行创建一定数量的子线程，并在第 18 行等待它们完成。注意 wait_all_threads() 丢弃了线程的返回值，这种情况下全是 NULL 返回值，没有什么意义。

```
1 int main(int argc, char *argv[])
2 {
```

```
3 int i;
4 int nkids = 1;
5
6 smp_init();
7 if (argc > 1) {
8 nkids = strtoul(argv[1], NULL, 0);
9 if (nkids > NR_THREADS) {
10 fprintf(stderr, "nkids=%d too big, max=%d\n",
11 nkids, NR_THREADS);
12 usage(argv[0]);
13 }
14 }
15 printf("Parent spawning %d threads.\n", nkids);
16 for (i = 0; i < nkids; i++)
17 create_thread(thread_test, (void *)i);
18 wait_all_threads();
19 printf("All threads completed.\n", nkids);
20 exit(0);
21 }
```

图 B.3: 父线程示例

B.3 锁

锁 API 如图 B.4 所示，每一个 API 在随后的章节中描述。

```
void spin_lock_init(spinlock_t *sp);
void spin_lock(spinlock_t *sp);
int spin_trylock(spinlock_t *sp);
void spin_unlock(spinlock_t *sp);
```

图 B.4: Locking API

B.3.1 spin_lock_init()

spin_lock_init() 初始化指定的 spinlock_t 变量，并且必须在将它传递给任何 spinlock 原语前被调用。

B.3.2 spin_lock()

spin_lock() 获取指定的 spinlock，如果有必要，就等待 spinlock 变为可用。在某些环境中，如 pthreads 中，这个等待将调用“spinning”，但是在其他一些环

境中，如 LINUX 内核，它将调用 blocking（xie.baoyou 注：作者应该是指实时补丁中的可睡眠的 spinlock）.

关键点是：在任意指定时刻，仅仅一个线程能够获得 spinlock。

B.3.3 spin_trylock()

spin_trylock()获取一个指定的 spinlock，但仅仅在它立即可用时才获取它。如果获取到锁，则返回真，否则返回假。

B.3.4 spin_unlock()

spin_unlock() 释放指定的锁，以允许其他线程获取它。

B.3.5 用法示例

一个名字 mutex 的 spinlock 用于保护计数变量：

```
spin_lock(&mutex);
counter++;
spin_unlock(&mutex);
```

问题 B.2：如果没有 mutex 的保护，变量 counter 将会出现什么问题?

但是，spin_lock()和 spin_unlock()会带来性能上的问题，我们将在 B.5 节进行分析。

B.4 每线程变量

图 B.5 显示了每线程变量 API。这个 API 类似于全局变量的每线程变量。虽然严格的说，这些 API 不是必需的，但是它能大大简化代码的编写。

```
DEFINE_PER_THREAD(type, name)
DECLARE_PER_THREAD(type, name)
per_thread(name, thread)
__get_thread_var(name)
init_per_thread(name, v)
```

图 B.5: 每线程变量 API

问题 B.3：在没有提供每线程变量 API 的系统中，你将如何工作?

B.4.1 DEFINE_PER_THREAD()

DEFINE_PER_THREAD()定义一个每线程变量。不幸的是，不可能在 LINUX 内核中提供一个这样的宏，但是有一个 init_per_thread()，允许在运行时进行初始化。

B.4.2 DECLARE_PER_THREAD()

DECLARE_PER_THREAD() 是一个 C 语言的声明。 用于访问其他文件中定义的每线程变量。

B.4.3 per_thread()

per_thread() 访问特定的线程变量.

B.4.4 __get_thread_var()

__get_thread_var() 访问当前线程的每线程变量.

B.4.5 init_per_thread()

init_per_thread() 将特定值设置到每线程变量中。

B.4.6 用法示例

假设我们有一个非常快速递增的计数器，但是很少访问它。 在 B.5 节将清楚的看到，使用每 CPU 变量将有助于实现这样的计数器。可以如下定义计数器：

```
DEFINE_PER_THREAD(int, counter);
```

必须如下初始化计数器:

```
init_per_thread(counter, 0);
```

线程可以如下递增它的计数器实例：

```
__get_thread_var(counter)++;
```

然后，计算所有计数器实例的总和。

```
for_each_thread(i)
  sum += per_thread(counter, i);
```

另外，使用其他机制也可以得到类似的效果，但是每线程变量用起来方便不

说，性能还不错。

B.5 性能

在使用锁和使用每线程变量进行计数器递增之间进行比较是有用的：

@@@ 需要补充 cache 方面的说明.

@@@ 需要更多的多核性能结果

@@@ 在临界段中也能工作吗?

性能差异非常大。本书的上的是帮助你编写 SMP 程序，也涉及实时响应方面的问题，以避免大的性能损失。下一节开始讨论引起性能损失的几个原因。

C.为什么使用内存屏障

CPU 设计者是怎么引起这个问题的？

简单的说，由于内存乱序以允许更好的性能，在某些情况下，需要内存屏障以强制保证内存顺序。例如，当正确操作依赖于内存顺序的时候。

要更好的回答这个问题，需要理解 CPU cache 是如何工作的，特别是需要使 cache 工作得很正确。如下：

- ✓ Cache 的结构
- ✓ 描述 cache 一致性协议如何确保 CPU 接受内存中的值。
- ✓ store buffers 及无效队列如何协助 cache 和 cache 一致性协议实现高性能。

我们将看到，内存屏障是必须的，但是对于高性能和可扩展性来说是不好的。这是由于：CPU 按照被访问的内存块进行了排序。

C.1 Cache 结构

现代 CPU 的速度比现代内存的速度快得多。2006 年的 CPU 可以在每纳秒内执行十条指令。但是需要很多个十纳秒才能从内存中取出一个数据。它们的速度不太一致（超过 2 个级别），这导致在现代 CPU 中产生了数 M 级别的缓存。这些缓存的速度与 CPU 是一致的，如图 C.1，典型的，可以在几个时钟周期内被访问 C.1。

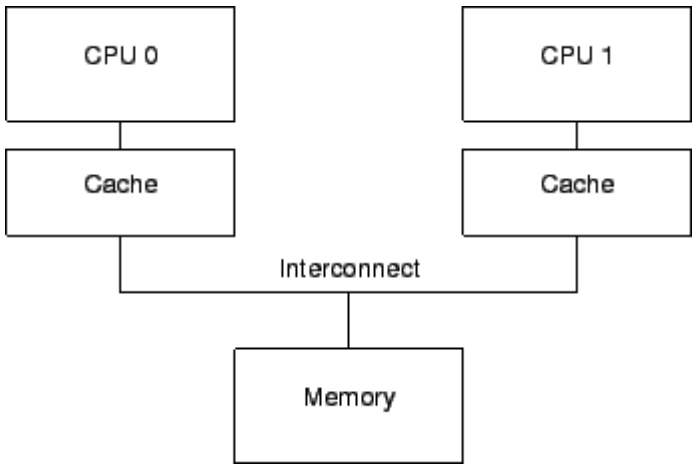


图 C.1: Modern Computer System Cache Structure

CPU 缓存和内存之间的数据流是固定长度的块，称为“cache lines”，通常是 2 的 N 次方长。范围从 16 到 256 字节不等。当一个特定的数据第一次被 CPU

访问时，在 cache 中还不存在，这称为“cache miss”(或者被称为“startup”或者“warmup” cache miss)。cache miss 意味着 CPU 必须等待(“stalled”) 数百个 CPU 周期。但是，数据将被装载入 CPU 缓存，后续的访问将通过缓存，因此可以全速运行。

经过一段时间后，CPU 的缓存将会全满，后续的 misses 需要换出 cache 中现有的数据。这样的 cache miss 被称为“capacity miss”，因为它是由于缓存能力限制而造成的。但是，大多数的缓存可能由于一个新数据而被换出，即使此时缓存还没有满。这是由于大量的缓存是通过固定长度的哈希桶来实现的 (或者叫“sets”，CPU 设计者是这样称呼的) ，如图 C.2.

这个缓存有 16 个“sets”和 2 个“ways”，共 32 个“lines”，每个节点包含一个 256 字节的“cache line”，它是一个 256 字节对齐的内存块。这个缓存稍微显得大了一点，但是这使得十六进制的运行更简单。从硬件的角度来说，这是一个 two-way set-associative 缓存，类似于软件的带 16 个桶的哈希表，每个桶的哈希链最多有两个元素。长度 (本例中是 32 个缓存行) 和相连性 (本例中是 2) 都被称为缓存的“geometry”。由于缓存是硬件实现的，哈希函数非常简单：从内存地址中取出 4 位作为哈希键值。

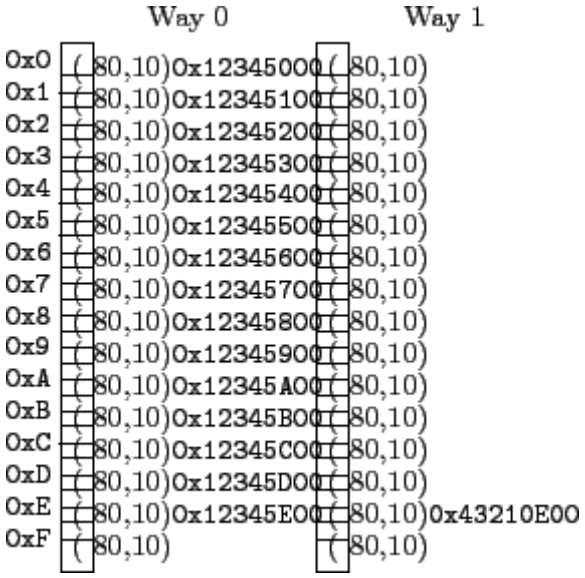


图 C.2: CPU Cache Structure

如图 C.2，每个块代表一个缓存条目，可以包含一个 256 字节的缓存行。不过，一个缓存条目可能为空，在图中标示为空块。其他的块用它所包含的内存地址表示。由于缓存行必须是 256 字节对齐，因此地址的低 8 位为 0。

在程序代码装载地址 0x43210E00- 0x43210EFF 时，图中的情况就可能发生。假设程序正在访问地址 0x12345F00，这个地址会哈希到 0xF 行，该行的两路都是空的，因此可以提供对应的 256 字节缓存行。如果程序访问地址 0x1233000，将会哈希到第 0 行，相应的 256 字节缓存行可以放到第 1 路。但是，如果程序访

问地址 0x1233E00，将会哈希到第 0xE 行，这是一个已经存在的行，必须有一个被替换出去，以腾出空间给新的行。如果随后访问被替换的行，会产生一次 cache miss，这样的 miss 被称为“associativity miss”。

更进一步说，我们仅仅考虑了读数据的情况。当写的时候会发生什么呢？由于在一个指定的 CPU 写数据前，让所有 CPU 都意识到数据被修改这一点是非常重要的。因此，它必须首先从缓存中移除，或者叫“invalidated”（使无效）。一旦使无效操作完成，CPU 可以安全的修改数据。如果数据存在于 CPU 缓存中，但是是只读的，这个过程称为“write miss”。一旦指定的 CPU 完成使无效操作，CPU 可以重新写（或者读）数据。

最后，如果另外某个 CPU 尝试访问数据，将会形成一次 cache miss，此时，由于第一个 CPU 为了写而使得缓存项无效，这被称为“communication miss”。因为这通常是由于几个 CPU 使用缓存通信造成的。

很明显，所有 CPU 必须小心的维护数据的一致性。所有取数据、使无效、写操作，很容易想得到，数据可能已经丢失（或者说损坏），或者说在不同的 CPU 缓存之间拥有冲突的值。这些问题由“缓存一致性协议”来解决，这将在下一节中描述。

C.2 缓存一致性协议

缓存一致性协议管理缓存行的状态，以防止数据不一致或者丢失。这些协议可能十分复杂，可能有数十种状态。但是我们仅仅需要关心 MESI 协议的四种状态。

C.2.1 MESI 状态

MESI 存在“modified”，“exclusive”，“shared”和“invalid”四种状态，协议可以在一个指定的缓存中应用这四种状态。因此，协议在每一个缓存行中维护一个两位的状态“tag”， 这个“tag”附着在缓存行的物理地址或者数据后。

处于“modified”状态的缓存行是由于相应的 CPU 最近进行了内存存储。并且相应的内存确保没有在其他 CPU 的缓存中出现。因此，“modified”状态的缓存行可以被认为被 CPU 所“owned”。由于缓存保存了最新的数据，因此缓存最终有责任将数据写回到内存，并且也应当为其他缓存提供数据，必须在当前缓存缓存其他数据之前完成这些事情。

“exclusive”状态非常类似于“modified”状态，唯一的例外是缓存行还没有被相应的 CPU 修改，这表示缓存行中的数据及内存中的数据都是最新的。但是，由于 CPU 能够在任何时刻将数据保存到该行，而不考虑其他 CPU，处于

“exclusive”状态也可以认为被相应的 CPU 所“owned”。也就是说，由于内存中的值是最新的，该行可以直接丢弃而不用回写到内存，也可以为其他缓存提供数据。

处于“shared”状态的缓存行可能被复制到至少一个其他 CPU 缓存中，这样在没有得到其他 CPU 的许可时，不能向缓存行存储数据。由于“exclusive”状态下，内存中的值是最新的，因此可以不用向内存回写值而直接丢弃缓存中的值，或者向其他 CPU 提供值。

处于“invalid”状态的行是空的，换句话说，它没有保存任何有效数据。当新数据进入缓存时，它替换一个处于“invalid”状态的缓存行。这个方法是比较好的，因为替换其他状态的缓存行将引起大量的 cache miss。

由于所有 CPUs 必须维护缓存行中的数据一致性视图，因此缓存一致性协议提供消息以调整系统缓存行的运行。

C.2.2 MESI 协议消息

前面章节中描述的内容需要在 CPU 之间通信。如果 CPUs 在单个共享总线上，只需要如下消息就足够了：

- ✓ Read：“read”消息包含缓存行需要读的物理地址。
- ✓ Read Response：“read response”消息包含较早前的“read”消息的数据。这个“read response”消息可能由内存或者其他缓存提供。例如，如果一个缓存请求一个处于“modified”状态的数据，则缓存必须提供“read response”消息。
- ✓ Invalidate“invalidate”消息包含要使无效的缓存行的物理地址。其他的缓存必须从它们的缓存中移除相应的数据并且响应此消息。
- ✓ Invalidate Acknowledge：一个接收到“invalidate”消息的 CPU 必须在移除指定数据后响应一个“invalidate acknowledge”消息。
- ✓ Read Invalidate：“read invalidate”消息包含要缓存行读取的物理地址。同时指示其他缓存移除数据。因此，它包含一个“read”和一个“invalidate”。“read invalidate”也需要“read response”以及“invalidate acknowledge”消息集。
- ✓ Writeback：“writeback”消息包含要回写到内存的地址和数据。(并且也许会“snooped”其他 CPUs 的缓存)。这个消息允许缓存在必要时换出“modified”状态的数据以腾出空间。

很有趣的是，共享内存的多核系统实际上是一个消息传递计算机。这意味着：使用分布式共享内存的 SMP 机器集群与使用消息传递实现共享内存的系统是两

个不同级别的体系结构。

问题 C.1：如果两个 CPUs 尝试同时使无效相同的缓存行会发生什么?

问题 C.2：当一个“invalidate”消息出现在一个大的多核系统中，每一个 CPU 必须发送一个“invalidate acknowledge”响应。不会产生 在系统总线上产生一个“invalidate acknowledge”风暴吗?

问题 C.3：如果 SMP 机器总是使用消息传递，为什么 SMP 没有出现麻烦?

C.2.3 MESI 状态图

一个指定缓存行的状态变化及协议消息发送及接收如下图 C.3：

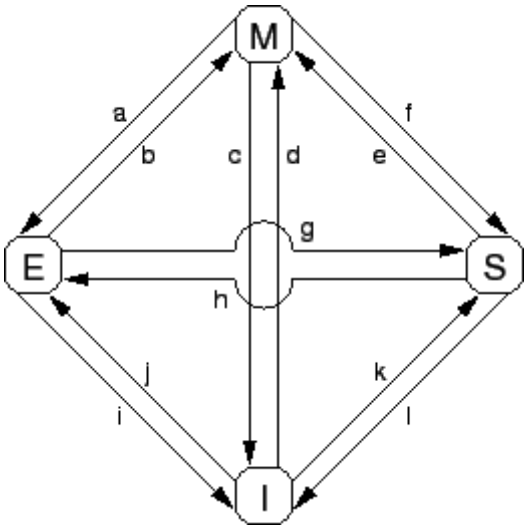


图 C.3: MESI Cache-Coherency State Diagram

图中的转换为下面所示：

- ✓ Transition (a)：缓存行被回写到内存，但是 CPU 仍然将它保留在缓存中，并在以后修改它。这个事务需要一个“writeback”消息。
- ✓ Transition (b)：CPU 将数据写到缓存行，该缓存行目前处于排它访问。不需要发送或者接收任何消息。
- ✓ Transition (c)：CPU 收到一个“read invalidate”消息，相应的缓存行已经被修改。CPU 必须使无效本地副本，然后响应“read response”和“invalidate acknowledge”消息，同时发送数据给请求的 CPU，指示它的本地副本不再有效。
- ✓ Transition (d)：CPU 进行一个原子读写操作，相应的数据没有在它的缓存中。它发送一个“read invalidate”消息，通过“read response”接收

数据。一旦它接收到一个完整的“invalidate acknowledge”响应集合，CPU 就完成此事务。

- ✓ Transition (e)：CPU 进行一个原子读写操作，相应的数据在缓存中是只读的。它必须发送一个“invalidate”消息，并等待“invalidate acknowledge”响应集合以完成此事务。
- ✓ Transition (f)：其他一些 CPU 读取缓存，其数据是从本 CPU 提供的，本 CPU 包含一个只读副本，可能是由于数据已经回写到内存中。这个事务开始于接收到一个“read”消息，并且本 CPU 响应了一个“read response”消息。
- ✓ Transition (g)：其他 CPU 读取数据，并且数据是从本 CPU 的缓存或者内存中提供的。本 CPU 包含了一个只读副本。这个事务开始于接收到一个“read”消息，并且本 CPU 响应了一个“read response”消息。
- ✓ Transition (h)：当前 CPU 将要写入一些数据到缓存行，于是发送一个“invalidate”消息。直到它接收到所有“invalidate acknowledge”消息后，CPU 才完成事务。可选的，其他 CPUs 通过“writeback”消息将缓存行的数据换出。这样，当前 CPU 就是最后一个缓存该数据的 CPU。
- ✓ Transition (i)：其他的 CPU 进行了一个原子读写操作，相应的缓存行被本 CPU 持有。本 CPU 将缓存行变成无效状态。这个事务开始于接收到“`read invalidate”消息，并且本 CPU 响应一个“read response”消息以及一个“invalidate acknowledge”消息。
- ✓ Transition (j)：本 CPU 保存一个数据到缓存行，但是数据还没有在它的缓存行中。因此发送一个“read invalidate”消息。直到它接收到“read response”消息以及所有“invalidate acknowledge”消息后，它才完成事务。
- ✓ Transition (k)：本 CPU 装载一个数据，但是数据还没有在缓存行中。CPU 发送一个“read”消息，当它接收到相应相应的“read response”消息后完成事务。
- ✓ Transition (l)：其他 CPU 存储一个数据到缓存行，但是该缓存行处于只读状态。这个事务开始于接收到一个“invalidate”消息，当前 CPU 响应一个“invalidate acknowledge”消息。

问题 C.4：硬件如何处理上面描述的延迟的事务?

C.2.4 MESI 协议示例

让我们从缓存行视图来理解这一点。最初，地址 0 处于内存中。在一个 4CPU

的系统中，它在几个直接映射的单行缓存中移动。表 C.1 展示了数据流，第一列是操作顺序，第二行表示执行操作的 CPU，第三行表示执行的操作，接下来是四个 CPU 的缓存行状态（内存地址在 MESI 状态后面）。最后两列表示相应的内存内容是否是最新的。(``V'') 为最新， (``I'')不是最新。

最初，CPU 缓存行处于“invalid”状态，数据在内存中。当 CPU 0 从地址 0 装载数据，它进入“shared”状态，并且内存中的数据是有效的。CPU 3 也从地址 0 装载数据，这样它也处于“shared”状态，并且内存中的数据仍然有效。接下来 CPU 0 装载其他缓存行 (地址 8)，通过使无效操作强制将地址 0 的数据换出缓存，缓存中的数据被换成地址 8 的数据。 CPU 2 装载地址 0 的数据，但是该 CPU 发现它将很快就会存储数据，因此它使用一个“read invalidate”消息以获得一个独享副本。CPU 3 缓存中的数据变成无效（但是内存中的数据仍然是有效的）。按下来 CPU 2 开始预期的存储操作，并将状态改变为“modified”。内存中的数据将不再是最新的。CPU 1 开始一个原子加操作，使用一个“read invalidate”操作从 CPU2 的缓存中窥探数据并使之无效，这样 CPU1 的缓存变成“modified”状态， (内存中的数据仍然是过期的)。最后，CPU1 从地址 0 读取数据，使用一个“writeback”消息将数据回写到内存。

表 C.1: Cache Coherence Example

			CPU Cache				Memory	
Sequence #	CPU #	Operation	0					
0		Initial State	-/I	/I	-/I	-/I		
1	0	Load	0/S	/I	-/I	-/I		
2	3	Load	0/S	-/I	-/I	0/S		
3	0	Invalidation	8/S	/I	-/I	0/S		
4	2	RMW	8/S	-/I	/E	-/I		
5	2	Store	8/S	/I	/M	-/I	I	
6	1	Atomic Inc	8/S	/M	/I	-/I		
7	1	Writeback	8/S	/S	/I	-/I		

问题 C.5：什么操作顺序会将 CPU 的缓存全部变为“invalid”状态?

C.3 不必要的存储延迟

虽然图 C.1 显示的缓存结构提供了好的性能，它可以在同一个 CPU 上反复的读取、存储数据，但是对于给定缓存行的第一次写来说，其性能是不好的。要理解这点，参考图 C.4，它显示了 CPU0 写数据到一个缓存行，而这个缓存行被 CPU1 所缓存。由于 CPU0 必须在开始写之前，等待缓存行数据的到来。CPU0 必须延迟额外的时间 C.3。

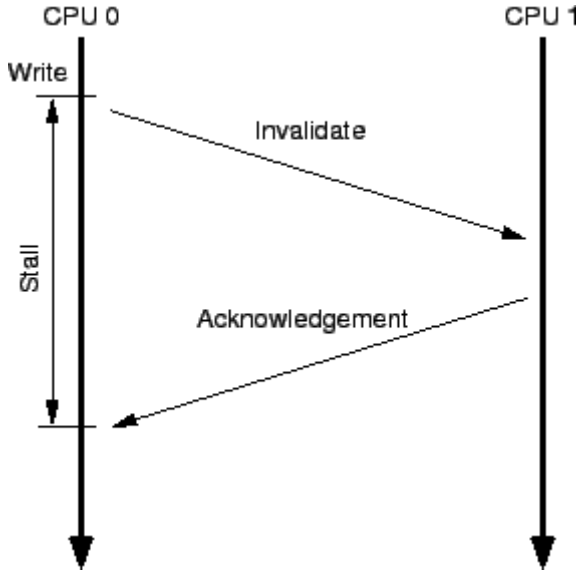


图 C.4: Writes See Unnecessary Stalls

其实没有理由强制让 CPU0 延迟这么久—最终，不管 CPU1 发送给它的缓存数据是什么，CPU0 都会无条件的覆盖它。

C.3.1 Store Buffers

避免这种不必要的写延迟的方法之一，就是在每个 CPU 和它的缓存之间，增加“store buffers”。如图 C.5。通过增加这些存储缓冲区，CPU0 可能简单的将要保存的数据放到存储缓冲区中，并且继续执行。当缓存行最后从 cpu1 转到 CPU0 时，数据将从存储缓冲区转到缓存行中。

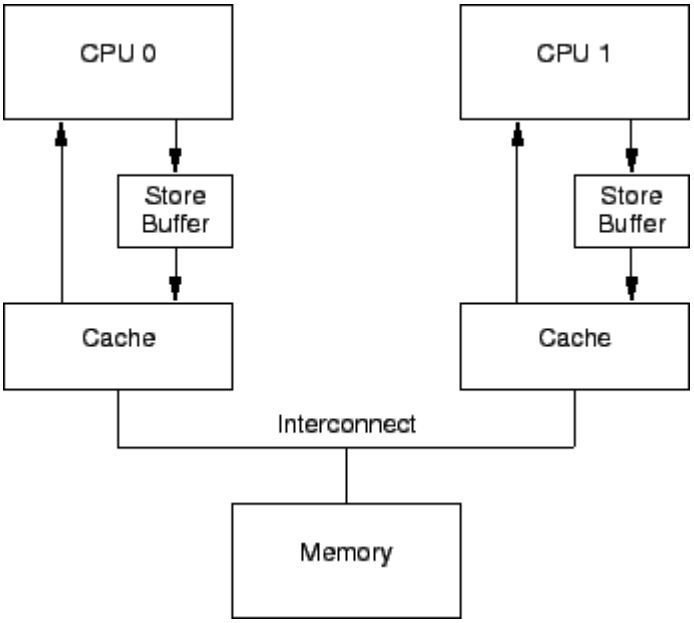


图 C.5: Caches With Store Buffers

但是，有一些复杂的事情需要处理，将在下面两节中描述。

C.3.2 Store Forwarding

第一个复杂的地方：违反了一致性。考虑如下代码：变量“a”和“b”都初始化为 0，CPU0 包含变量“a”的缓存行，而 CPU1 包含变量“b”的缓存行：

```
1  a = 1;
2  b = a + 1;
3  assert(b == 2);
```

有时，并不会按预期发生断言。可是，如果有谁足够愚蠢到使用如图 C.5 所示的简单体系结构，结果将令人惊奇。这样的系统可能看起来会按以下的事情顺序发生：

- ✓ CPU 0 开始执行 a = 1。
- ✓ CPU 0 发现“a”在缓存中。
- ✓ CPU 0 因此发送一个“read invalidate”消息，以获得包含“a”的独享缓存行。
- ✓ CPU 0 将“a”记录到存储缓冲区。
- ✓ CPU 1 接收到“read invalidate”消息，并且发送缓存行数据，然后从它的缓存行中移除。
- ✓ CPU 0 开始执行 b = a + 1。
- ✓ CPU 0 从 CPU1 接收到缓存，它仍然得到一个为“0”的“a”值。
- ✓ CPU 0 从它的缓存中读取到“a”的值，发现其值为 0。

- ✓ CPU 0 将存储队列中的条目应用到缓存行，设置缓存行中的值为 1。
- ✓ CPU 0 将值 0 加 1，并存储该值到包含“b”的缓存行中 (假设已经包含在 CPU0 的缓存行中)。
- ✓ CPU 0 执行 assert(b == 2), 并引起错误。

问题在于我们拥有两“a”的副本，一个在缓存中，另一个在存储缓冲区中。

这个例子破坏了一个重要的保证：即每一个 CPU 将总是按照程序顺序看到它的操作。没有这个保证则与软件直觉相反的，这样硬件会按照“store forwarding”来实现。这样每个 CPU 引用它的存储缓存区与它的缓存一样。如图 C.6。换句话说，一个特定的 CPU 存储操作直接发给后续的读操作，而不用通过缓存。

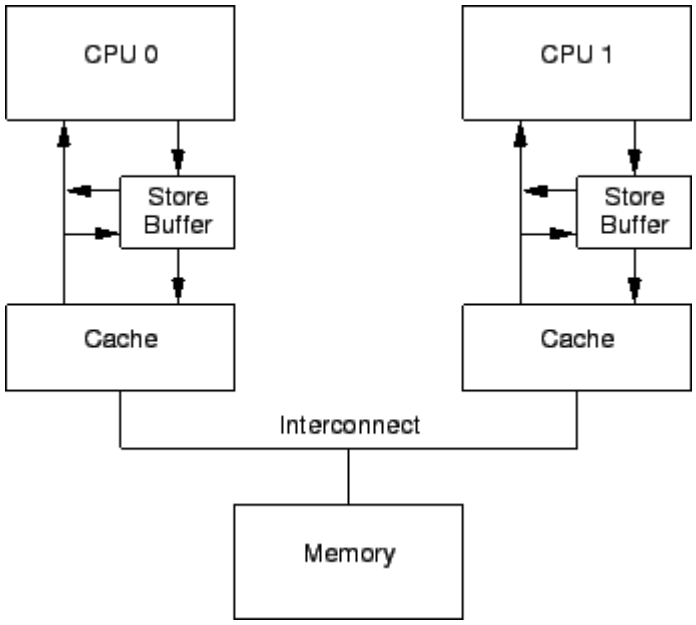


图 C.6：Caches With Store Forwarding

当实现 store forwarding 时，上面的顺序中， 将发现在存储缓冲区中的“a”的值 1 ，因此最终的“b”值将是 2，这正是我们期望的。

C.3.3 存储缓冲区及内存屏障

第二个复杂性在于：违反了全局内存序，考虑如下的代码顺序，其中变量“a”、“b”的初始值是 0。

```
1 void foo(void)
2 {
3     a = 1;
4     b = 1;
5 }
6
7 void bar(void)
```

```
 8 {
 9     while (b == 0) continue;
10     assert(a == 1);
11 }
```

假设 CPU 0 执行 foo()，CPU 1 执行 bar()，再假设包含“a”的缓存行仅仅位于 CPU1 的缓存中，包含“b”的缓存行仅仅包含在 CPU1 中。接下来的操作顺序如下：

CPU 0 执行 a = 1。缓存行不在 CPU0 的缓存中，因此 CPU0 将“a”的新值放到存储缓冲区，并发送一个“read invalidate”消息。

CPU 1 执行 while (b == 0) continue，但是包含“b”的缓存行不在缓存中，它发送一个“read”消息。

CPU 0 执行 b = 1，它已经在缓存行中有“b”的值了 (换句话说，缓存行已经处于“modified”或者“exclusive”状态)，因此它存储新的“b”值在它的缓存行中。

CPU 0 接收到“read”消息，并且发送缓存行中的新的“b”的值 1，同时将缓存行设置为“shared”状态。

CPU 1 接收到包含“b”值的缓存行，并将其值写到它的缓存行中。

CPU 1 现在完成执行 while (b == 0) continue, 由于它发现“b”的值是 1，它开始处理下一条语句。

CPU 1 执行 assert(a == 1)，并且, 由于 CPU 1 工作在旧的“a”的值，因此验证失败。

CPU 1 接收到“read invalidate”消息, 并且发送包含“a”的缓存行到 CPU0，同时使它的缓存行变成无效。但是已经太迟了。

CPU 0 接收到包含“a”的缓存行，将且将存储缓冲区的数据保存到缓存行中，这使得 CPU1 验证失败。

问题 C.6: 在上面的第 1 步，为什么 CPU0 需要执行一个“read invalidate”而不是简单的“invalidate”操作?

在此，硬件设计者不能直接的帮助我们，因为 CPUs 没有办法处理相关联的变量。因此，硬件设计者提供内存屏障指令，以允许软件告诉 CPU 这些关联变量的存在。程序必须修改，以包含内存屏障：

```
1 void foo(void)
2 {
3     a = 1;
4     smp_mb();
5     b = 1;
6 }
7
```

```
 8 void bar(void)
 9 {
10     while (b == 0) continue;
11     assert(a == 1);
12 }
```

内存屏障 smp_mb()将导致 CPU 在保存后续的存储操作到缓存行前，刷新它的存储缓冲区。CPU 可能简单的停下来，直到存储缓冲区变成空，也可能是简单的使用存储缓冲区保存后续的存储操作，直到前面所有的存储缓冲区已经被保存到缓存行中。

后一种情况下，操作可能是如下所示：

CPU 0 执行 a = 1。缓存行不在缓存中，因此 CPU 0 将“a”的新值放到存储缓冲区，并发送一个“read invalidate”消息.

CPU 1 执行 while (b == 0) continue，但是包含“b”的缓存行不在缓存中，因此它发送一个“read”消息.

CPU 0 执行 smp_mb(),，并标记当前所有存储缓冲区的条目。(也就是说 a = 1).

CPU 0 执行 b = 1。它的缓存行已经存在了。 (也就是说, 缓存行已经处于“modified”或者“exclusive”状态)，但是在存储缓冲区中存在一个标记条目。因此，它不将新值存放到缓存行，而是存放到存储缓冲区中。 (但是“b”不是一个标记条目).

CPU 0 接收到“read”消息，同时发送包含“b”值的缓存行给 CPU1。它也标记本地缓存行为“shared”。

CPU 1 读取到包含“b”的缓存行，并将它复制到本地缓存中。

CPU 1 现在可以装载“b”的值了，但是发现它的值仍然为“0”，因此它重复该语句。“b”的新值被安全的隐藏在 CPU0 的存储缓冲区中。

CPU 1 接收到“read invalidate”消息，并且发送包含“a”的缓存行给 CPU0，并且使它的缓存行无效。

CPU 0 接收到包含“a”的缓存行，并且使用存储缓冲区的值替换缓存行为“modified”状态。

由于被存储的“a”是存储缓冲区中唯一被 smp_mb()标记的条目，因此 CPU0 也能够存储“b”的新值到缓存行中-除非包含“b”的缓存行处于“shared”状态。

CPU 0 发送一个“invalidate”消息给 CPU 1。

CPU 1 接收到“invalidate”消息，刷新包含“b”的缓存行，并且发送一个“acknowledgement”消息给 CPU 0.

CPU 1 执行 while (b == 0) continue，但是包含“b”的缓存行不在缓存中，因此它发送一个“read”消息给 CPU 0。

CPU 0 接收到“acknowledgement”消息，将包含“b”的缓存行设置成“exclusive”状态。 CPU 0 现在存储新的“b”值到缓存行。

CPU 0 接收到“read”消息，同时发送包含新的“b”值的缓存行给 CPU 1。它也标记该缓存为“shared”。

CPU 1 接收到包含“b”的缓存行，并将它复制到本地缓存中。

CPU 1 现在能够装载“b”的值了，由于它发现“b”的值为 1，它退出循环并执行下一条语句。

CPU 1 执行 assert(a == 1)，但是包含“b”的缓存行不在它的缓存中。一旦它从 CPU0 获得这个缓存行，它将使用最新的“a”的值，验证语句将通过。

C.4 不必要的存储延迟

不幸的是，每一个存储缓冲区相对而言都比较小，意味着 CPU 执行一段较小的代码就可能填满它的存储缓冲区。(例如，当所有的结果都发生了 cache misses 时)。这样，CPU 在继续执行前，必须等待 invalidations 完成。相同的情况也在内存屏障后面发生，当所有后续存储操作指令必须等待 invalidations 完成，而不管这些存储是否存在 cache misses.

这可以通过 invalidate acknowledge 消息更快到得到解决。实现这一点的方法之一是使用每 CPU 的 invalidate 消息队列或者“invalidate queues”。

C.4.1 无效队列

使无效应答消息需要如此长的时间，其原因之一是它们必须确保相应的缓存行实际变成无效了。如果缓存比较忙的话，这个使无效操作可能被延迟。例如，如果 CPU 集中的装载或者存储数据，并且数据都在缓存中。另外，如果在一个较短的时间内大量的使无效消息到达，一个特定的 CPU 会忙于处理它们。这会使得其他 CPU 陷于停顿。

但是，在发送应答前，CPU 不必实际的使无效缓存行。它可以将使无效消息排队。并且它明白在发送更多的关于该缓存行的消息前，需要处理这个消息。

C.4.2 使无效队列及使无效应答

图 C.7 显示一个带无效队列的系统。一个带无效队列的 CPU 可以迅速应答一个使无效消息，而不是必须等待相应的行实际变成无效状态。当然，CPU 必须在准备发送无效消息前，参考它的无效队列。-- 如果一个相应的缓存行条目在无效队列中，则 CPU 不能立即发送无效消息，它必须等待无效队列被处理。

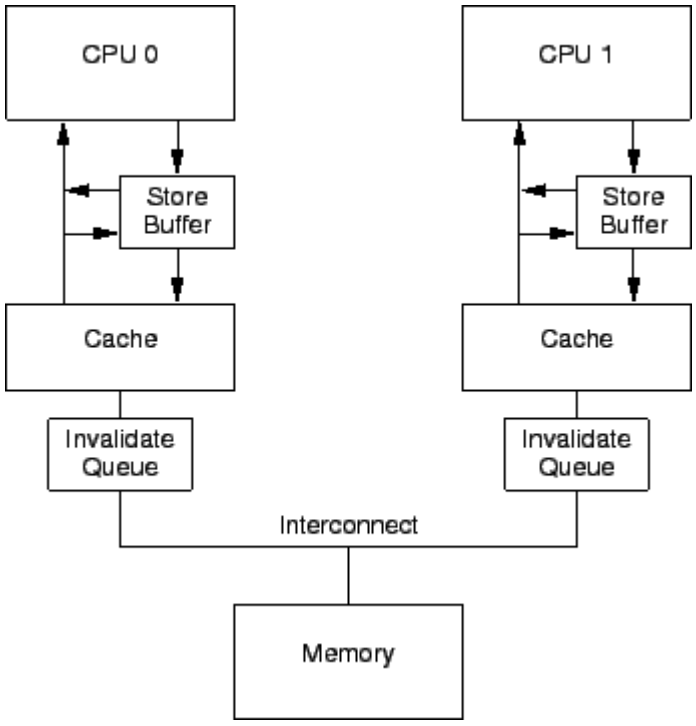


图 C.7: Caches With Invalidate Queues

将一个条目放进无效队列，实际上是由 CPU 承诺在发送任何 MESI 协议消息前处理该条目而不考虑缓存行。只要相应的数据结构不存在大的竞争，CPU 会很出色的完成此事。

但是，实际上消息能被缓冲在无效队列中，这带来了额外的内存乱序的机会，这将在下一节讨论。

C.4.3 无效队列及内存屏障

我们假设 CPU 将使无效请求排队，并立即响应它们。这个方法将缓存使无效的延迟降到最小，但是将使用内存屏障失效，看看如下示例：

假设“a”和“b”被初始化为 0， “a”是只读的(MESI ``shared" 状态，“b”被 CPU 0 拥有 (MESI ``exclusive" 或者 ``modified" 状态)。然后假设 CPU 0 执行 foo() 而 CPU 1 执行 bar()，代码片段如下：

```
1 void foo(void)
2 {
3     a = 1;
4     smp_mb();
5     b = 1;
6 }
7
8 void bar(void)
```

```
 9 {
10     while (b == 0) continue;
11     assert(a == 1);
12 }
```

操作顺序可能如下:

- ✓ CPU 0 执行 a = 1。CPU0 中相应的缓存行是只读的，因此 CPU0 将“a”的新值放入存储缓冲区，并发送一个“invalidate”消息，以从 CPU1 的缓存中刷新相应的缓存行。
- ✓ CPU 1 执行 while (b == 0) continue, 但是包含“b”的缓存行不在缓存中，因此它发送一个“read”消息.
- ✓ CPU 1 接收到 CPU 0 的“invalidate”消息，将它无效，并立即响应它。
- ✓ CPU 0 接收到来自于 CPU 1 的响应，因此通过第 4 行的 smp_mb()，从存储缓冲区移动“a”的值到缓存行。
- ✓ CPU 0 执行 b = 1。它拥有这个缓存行(也就是说, 缓存行已经处于 ``modified'' 或者 ``exclusive'' 状态)，因此它将“b”的新值存储到缓存行中。
- ✓ CPU 0 接收到“read”消息，并且发送包含“b”的新值的缓存行到 cpu1，也标记缓存行为“shared”状态。
- ✓ CPU 1 接收到包含“b”的缓存行并且将其应用到本地缓存。
- ✓ CPU 1 现在执行完 while (b == 0) continue，因为它发现``b''的值为 1, 接着处理下一条语句.
- ✓ CPU 1 执行 assert(a == 1)，由于旧的“a”值还在它 CPU1 的缓存中，因此陷入错误。
- ✓ 虽然陷入错误，CPU 1 处理已经排队的“invalidate”消息，并且刷新包含“a”值的缓冲行。

问题 C.7：在第 1 种情况下的第一步，如 C.4.3 节, 为什么发送一个“invalidate”而不是“read invalidate”消息? CPU 0 不需要共享这个缓存行的其他变量?

没有那么多地方来加快 invalidation 响应，如果这样做，将导致内存屏障被忽略，这是比较明确的。但是，内存屏障指令能够与无效队列交互，这样，当一个特定的 CPU 执行一个内存屏障时，它标记无效队列中的所有条目，并强制所有后续的装载操作进行等待，直到所有标记的条目都保存到 CPU 的缓存中。因此，我们可以在 bar 函数中添加一个内存屏障，如下：

```
1 void foo(void)
2 {
3     a = 1;
```

```
 4     smp_mb();
 5     b = 1;
 6 }
 7
 8 void bar(void)
 9 {
10     while (b == 0) continue;
11     smp_mb();
12     assert(a == 1);
13 }
```

问题 C.8：什么??? 为什么需要在这里加一个内存屏障，直到 while 循环完成前，CPU 不可能执行 assert()?

有了这个变化后，操作顺序可能是如下的：

- ✓ CPU 0 执行 a = 1。相应的缓存行在 CPU0 的缓存中是只读的，因此 CPU0 将新值放入它的存储缓冲区，并且发送一个“invalidate”消息以刷新 CPU1 的缓存。
- ✓ CPU 1 执行 while (b == 0) continue，但是包含“b”的缓存行不在它的缓存中，因此它发送一个“read”消息。
- ✓ CPU 1 接收到 CPU 0 的“invalidate”消息，将它清除，并立即响应它。
- ✓ CPU 0 接收到 CPU1 的响应，因此它处理 smp_mb()，将“a”从它的存储缓冲区移到缓存行。
- ✓ CPU 0 执行 b = 1.它已经拥有缓存行 (换句话说, 缓存行处于 ``modified" 或者 ``exclusive" 状态)，因此它存储“b”的新值到缓存行。
- ✓ CPU 0 接收 ``read" 消息，并且发送包含新的“b”值的缓存行给 CPU1，同时标记缓存行为 ``shared" 状态.
- ✓ CPU 1 接收到包含“b”的缓存行并更新到它的缓存中。
- ✓ CPU 1 现在结束执行 while (b == 0) continue，因为它发现 ``b" 的值为 1，它处理下一条语句，这是一条内存屏障指令。
- ✓ CPU 1 必须延迟，直到它处理无效队列中的所有消息。
- ✓ CPU 1 处理已经入队的 ``invalidate" 消息，从它的缓存中刷新包含“a”的缓存行。
- ✓ CPU 1 执行 assert(a == 1)，由于包含“a”的缓存行已经不在它的缓存中，它发送一个 ``read" 消息。
- ✓ CPU 0 响应 ``read" 消息，发送它的包含新的“a”值的缓存行。
- ✓ CPU 1 接收到缓存行，它包含新的“a”的值，因此不会陷入失败。

即使有很多 MESI 消息，CPUs 最终都会正确的应答。这一节阐述了 CPU 设计者为什么必须小心的处理它们的缓存一致性优化操作。

C.5 读和写内存屏障

在前一节，内存屏障用来标记存储缓冲区和无效队列。但是在我们的代码片段中，foo()没有必要做无效队列相关的任何操作，类似的，bar()也没有必要做与存储缓冲区相关的任何操作。

因此，很多 CPU 提供弱的内存屏障指令。简单的说，一个“读内存屏障”仅仅标记它的无效队列，一个“读内存屏障”仅仅标记它的存储缓冲区，完整的内存屏障同时标记无效队列及存储缓存缓冲区。

这样的效果是：读内存屏障仅仅保证装载顺序，因此所有在读内存屏障之前的装载将在所有之后的装载前完成。类似的，写内存屏障仅仅保证写之间的顺序。完整的内存屏障同时保证写和读之间的顺序。

如果我们修改 foo 和 bar ，以使用读和写内存屏障，将会是如下所示：

```
1 void foo(void)
2 {
3     a = 1;
4     smp_wmb();
5     b = 1;
6 }
7
8 void bar(void)
9 {
10    while (b == 0) continue;
11    smp_rmb();
12    assert(a == 1);
13 }
```

某些计算机甚至有更多的内存屏障，但是理解这三个屏障通常能够很好的理解内存屏障了。

C.6 内存屏障示例

本节提供了一些有用的内存屏障用法。虽然它们大多数时候能够正常工作，但是一些只能在特定 CPUs 上运行。如果要编写在所有 CPUs 都能正常运行的代码，这些代码就需要作废。为了更好的理解本节的内容，我们首先需要关注乱序体系结构。

C.6.1 乱序体系结构

Paul 已经见过很多种乱序的计算机系统，但是乱序确实很让人难以捉摸，理

解它需要丰富的硬件方面的知识。与其针对一个特定的硬件说事，但不如带领读者看看详细的技术规范。让我们虚构一个最大限度的乱序体系结构吧。

这个硬件必须遵守以下的约束 [McK05a,McK05b]:

- ✓ 每一个 CPU 总是按照编程顺序来感知内存访问。
- ✓ 仅仅在操作不同地址时，CPUs 才对特定的存储操作进行重新排序。
- ✓ 一个特定 CPU 在内存屏障之前的所有装载操作 (smp_rmb()) 将被所有随后的读内存屏障后面的操作之前被所有 CPU 所感知。 （xie.baoyou 注：这有点象是读绕口令）
- ✓ 所有在写内存屏障之前的写操作 (smp_wmb()) 都将比随后的写操作先感知。
- ✓ 所有在内存屏障之前的内存访问 (装载和存储) (smp_mb())都将比随后的内存访问先感知。

问题 C.9：可以保证每一个 CPU 看到它自己的内存访问顺序，同时能保证每一个用户级的线程也看到自己的内存访问顺序吗？为什么？

假设一个大的非统一缓存的体系(NUCA) 系统，为了给 CPUs 提供一个公平的内部访问带宽，在每一个节点接口提供了一个每 CPU 队列，如图 C.8。虽然一个特定 CPU 的访问是由内存屏障排序的，但是，一对 CPUs 的访问将被重排：

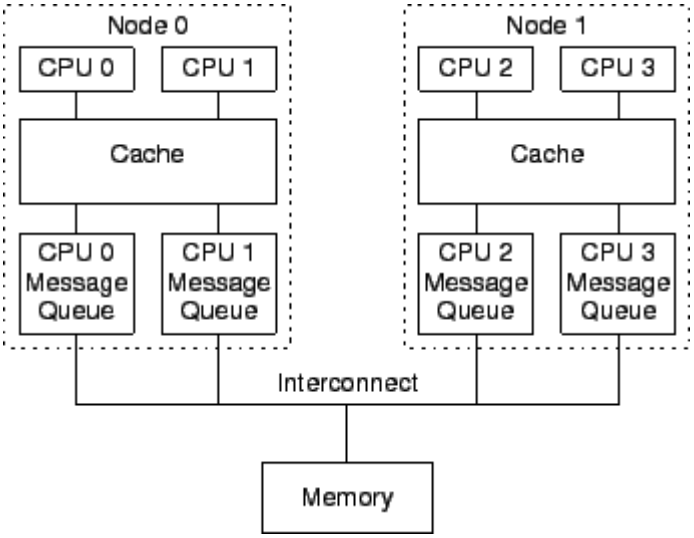


图 C.8: Example Ordering-Hostile Architecture

C.6.2 示例 1

表 C.2 展示了三个代码段，同时被 CPUs 0, 1, 和 2 执行。``a", ``b", 和 ``c" 被初始化为 0.

表 C.2: Memory Barrier Example 1

CPU 0	CPU 1	CPU 2
a = 1;		
smp_wmb();	while (b == 0);	
b = 1;	c = 1;	z = c;
		smp_rmb();
		x = a;
		assert(z == 0 || x == 1);

假设 CPU 0 刚经过很多 cache misses，因此它的消息队列是满的，但是 CPU 1 独占的使用它的缓存，因此它的消息队列是空的。 CPU 0 在向``a" 和 ``b"赋值时，看起来节点 0 的缓存是立即生效的 (因此对 CPU 1 来说也是可见的)，但是将阻塞于 CPU0 之前的流量。与之相对的是，CPU 1 工作于 ``c"时，将基于 CPU 1 的空队列。因此， CPU 2 将在看到 CPU0 对“a”的赋值前，先看到 CPU 1 对 ``c" 的赋值，这导致验证失败，即使有内存屏障也是这样。

从原理上说，可移植的代码不能依赖这个代码顺序，可是，实际上它可以运行在所有主流的系统中。

问题 C.10：可以在 CPU1 的``while" 语句与赋值语句间插入一个屏障来解决这个问题吗？为什么？

C.6.3 示例 2

表 C.3 示了代码片段，在 CPUs 0，1 和 2 上并行执行。 ``a" 和 ``b" 都初始化为 0.

表 C.3: Memory Barrier Example 2

CPU 0	CPU 1	CPU 2
a = 1;	while (a == 0);	
	smp_mb();	y = b;
	b = 1;	smp_rmb();
		x = a;

		assert(y == 0 || x == 1);

我们再一次假设 CPU 0 刚遇到很多 cache misses，因此它的消息队列满了，但是 CPU1 独享它的缓存，因此它的消息是空的。那么，CPU0 给“a”赋值将立即反映在 Node0 上， (因此对于 CPU 1 来说也是可见的)，但是会阻塞于 CPU0 以前的流量上面。相对的，CPU1 对“b”的赋值将基于 CPU1 的空队列进行工作。因此，CPU2 在看到 CPU0 对“a”的赋值前，可以看到 CPU1 对“b”的赋值。这导致 assert 失败，尽管存在内存屏障。

从原理上来说，编写可移植代码不能用上面的例子，但是，实际上这段代码可以在大多数主流的计算机正常运行。

C.6.4 示例 3

表 C.4 展示的代码片段，在 CPUs 0, 1, 和 2 上并行执行.所有变量都初始化为 0

表 C.4: Memory Barrier Example 3

CPU 0	CPU 1	CPU 2	
1	a = 1;		
2	smb_wmb();		
3	b = 1;	while (b == 0);	while (b == 0);
4		smp_mb();	smp_mb();
5		c = 1;	d = 1;
6	while (c == 0);		
7	while (d == 0);		
8	smp_mb();		
9	e = 1;		assert(e == 0 || a == 1);

请注意：不管是 CPU 1 还是 CPU 2 都要看到 CPU0 在第三行对“b”的赋值前，才能处理第 5 行。一旦 CPU 1 和 2 已经执行了第 4 行的内存屏障，它们就能够看到 CPU0 在第 2 行的内存屏障前的所有赋值。类似的，CPU0 在第 8 行的内存屏障与 CPU1 和 CPU2 在第 4 行的内存屏障相对，因此 CPU0 将不会执行第 9 行的内存赋值，直到它看对“a”的赋值被其他 CPU 可见。因此，CPU2 在

第 9 行的 assert 将不会触发。

问题 C.11：假设 CPU1 和 CPU2 在第 3-5 行是中断处理代码，而 CPU2 的第 9 行运行在进程级。将会发生什么？要正常工作，需要做些什么？换句话说，如何避免 assert 的发生？

问题 C.12：如果 CPU 2 执行 assert(e==0||c==1)，这个 assert 会触发吗?

LINUX 内核的 synchronize_rcu() 原语使用了类似于本例中的算法。

C.7 特定 CPUs 的内存屏障指令

每个 CPU 都有它自己特定的内存屏障指令。实际上，很多软件环境，包括 pthreads 和 Java，禁止直接使用内存屏障，限制程序员使用互斥锁。表 C.5 中，前四列指示 CPU 是否允许四种装载和存储组合进行重排。接下来两列表示 CPU 是否允许带原子指令的加载、存储被重排。

第七列，数据依赖读操作，需要进行解释。简短的讲，Alpha 需要为关联数据之间的读使用内存屏障。是的，这表示 Alpha 可能在它取得指针值之前，取得指针指向的值。请参考：http://www.openvms.compaq.com/wizard/wiz_2637.html. 这种极端弱序的模型的好处是缓存硬件更加简单。允许更高的时钟频率。

最后一列表示一个 CPU 是否拥有一个不一致的指令缓存和流水线。一些 CPUs 需要特殊的指令以执行自修改代码。

带括号的 CPU 表示允许这样的模式，但是实际上很少使用。

表 C.5: Summary of Memory Ordering

	Loads Reordered After Loads?	Loads Reordered After Stores?	Stores Reordered After Stores?	Stores Reordered After Loads?	Atomic Instructions Reordered With Loads?	Atomic Instructions Reordered With Stores?	Dependent Loads Reordered?	Incoherent Instruction Cache/Pipeline?
Alpha	Y	Y	Y	Y	Y	Y	Y	Y
AMD64				Y				
ARMv7-A/R	Y	Y	Y	Y	Y	Y		Y
IA64	Y	Y	Y	Y	Y	Y		Y

(PA-RISC)	Y	Y	Y	Y				
PA-RISC CPUs								
POWER™	Y	Y	Y	Y	Y	Y		Y
(SPARC RMO)	Y	Y	Y	Y	Y	Y		Y
(SPARC PSO)			Y	Y		Y		Y
SPARC TSO				Y				Y
x86				Y				Y
(x86 OOStore)	Y	Y	Y	Y				Y
zSeries®				Y				Y

不直接使用内存屏障有它的理由。但是，某些环境，如 LINUX 内核，需要直接使用内存屏障。因此，Linux 提供了精心选择的内存屏障原语，如下：

smp_mb(): 加载、存储内存屏障。这表示在内存屏障之前的加载、存储将在随后的加载、存储之前提交。

smp_rmb(): 读内存屏障仅仅禁止加载重排。

smp_wmb(): 写内存屏障仅仅禁止存储重排。

smp_read_barrier_depends() 强制禁止重排后续操作中，对前面的操作有依赖的操作。除了 ALPHA 之外，这个原语在其他体系上都是空操作。

mmiowb()禁止 MMIO 写乱序。在 spinlock 已经强制禁止 MMIO 乱序的平台中，这个原语是空操作。非空 mmiowb()平台包括(但是不是全部) IA64、FRV、MIPS 和 SH。这个原语比较新，因此少有驱动使用它。

smp_mb()、smp_rmb()和 smp_wmb()原语也强制编译禁止会引起内存重排的优化。smp_read_barrier_depends() 有类似的效果，但是仅仅是在 Alpha CPUs 上才有用。参见 12.2 节以获得更多的使用这个原语的信息。

这些原语仅仅在 SMP 上才产生代码，但是，它们也存在一个 UP 版本 (mb(), rmb(), wmb(), 和 read_barrier_depends()) ，这些原语在 UP 内核中也产生代码。smp_ 版本应用在大多数情况下。但是，后面的这些 UP 版本的原语在编写驱动时也是有用的。缺少内存屏障指令时，CPUs 和编译器都将重排这些访问。最好将设备访问作为强序访问，否则可能会让内核崩溃，某些情况下，会损坏您的硬件。

大多数内核开发者只要用好这些接口就行了，没有必要担心所有 CPU 内存

屏障的特性。当然，如果你想深入了解一个特定 CPU 的代码，将不是这样。

此外，所有的 LINUX 内核锁原语(spinlocks、读写锁、信号量、RCU, ...)包含必要的屏障原语。因此，如果您使用这些代码，就不必使用内存屏障原语。

也就是说，每种 CPU 的内存一致性模型知识对调试是有用的，更不用说编写体系特定的代码或者同步原语了。

此外，一知半解是非常有害的。对那些想要深入理解个别 CPU 的内存一致性模型的人来说，下一节描述了很多常见的 CPU 一致性模型。虽然没有什么能够代替阅读 CPU 文档，但是这些节是一个良好的概述。

C.7.1 Alpha

对一个已经宣布结束其生命周期的 CPU 讨论这么多，这看起来有点奇怪。但 Alpha 是有趣的，因为它是一个非常弱序的结构，它尽可能的重排内存操作。因此 LINUX 内核已经为它定义了内存序原语。这些原语必须工作在所有 CPU 上。因此，理解 Alpha 上的原语对于内核开发者来说是非常重要的。

Alpha 和其他 CPUs 在代码上的不同如下图 C.9 所示。第 9 行的 smp_wmb() 保证第 6-8 行的初始化操作在第 10 行的加操作前被执行。因此，无锁搜索将能够正常运行。这在所有 CPU 上都能正常运行，唯独在 Alpha 上不行。

```
1 struct el *insert(long key, long data)
2 {
3    struct el *p;
4    p = kmalloc(sizeof(*p), GFP_ATOMIC);
5    spin_lock(&mutex);
6    p->next = head.next;
7      p->key = key;
8      p->data = data;
9    smp_wmb();
10   head.next = p;
11   spin_unlock(&mutex);
12 }
13
14 struct el *search(long key)
15 {
16   struct el *p;
17   p = head.next;
18   while (p != &head) {
19         /* BUG ON ALPHA!!! */
20         if (p->key == key) {
21               return (p);
22         }
```

```
23          p = p->next;
24   };
25   return (NULL);
26 }
```

图 C.9: Insert and Lock-Free Search

Alpha 是非常弱序的，因此图 C.9 第 20 行能看见第 6-8 行初始化之前的旧的值。

图 C.10 显示了这是如何发生的。假设链表头 head 被 cache bank 0 处理，而新节点被 cache bank 1 处理。在 Alpha 上，smp_wmb() 将确保第 6-8 行的缓存无效操作在在第 10 行之前先到达到 interconnect，但是不确保新值到达读 CPU 的顺序。例如，读 CPU 的 cache bank 1 可能非常忙，但是 cache bank 0 是空闲的。这可能导致新节点的刷新操作被延迟，因此读 CPU 获得了指针的新值，但是节点值仍然是旧的。参阅先前提到的 WEB 站点以获得更多信息 C.6

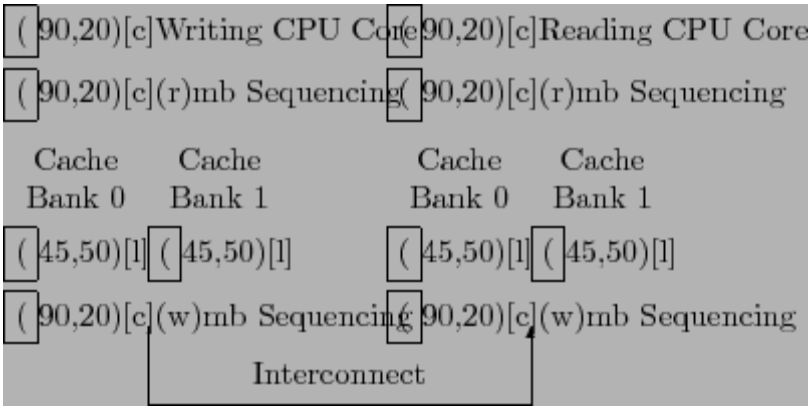


图 C.10: Why smp_read_barrier_depends() is Required

可以在读取指针及解除指针引用之间加一个 smp_rmb() 原语。但是，这增加了系统的开销，而这种开销在其他系统上是不必要的。 (如 i386, IA64, PPC, 和 SPARC) ，这些系统在读端会考虑数据引用的问题。
smp_read_barrier_depends() 原语在 LINUX2.6 中被加入，以消除这些系统上的开销。该原语可能如图 C.11 第 19 行这样使用。

也可以实现一个软件屏障，用来替换 smp_wmb()。它确保所有读 CPUs 按照写 CPU 的写顺序看到数据。但是，这个方法被 LINUX 开源社区认为在极端的弱序的系统（如 Alpha）会产生严重的开销。 这个软件屏障可以通过向其他所有 CPUs 发送 IPI 来实现。当收到这样一个 IPI 时，CPU 执行一个内存屏障指令。需要额外的逻辑以避免死锁。当然，有数据依赖处理的 CPUs 定义一个类似于 smp_wmb()的屏障就行了。 也许应当在以后再看看本节。

```
1 struct el *insert(long key, long data)
2 {
3 struct el *p;
```

```
4 p = kmalloc(sizeof(*p), GFP_ATOMIC);
5 spin_lock(&mutex);
6 p->next = head.next;
7 p->key = key;
8 p->data = data;
9 smp_wmb();
10 head.next = p;
11 spin_unlock(&mutex);
12 }
13
14 struct el *search(long key)
15 {
16 struct el *p;
17 p = head.next;
18 while (p != &head) {
19 smp_read_barrier_depends();
20 if (p->key == key) {
21 return (p);
22 }
23 p = p->next;
24 };
25 return (NULL);
26 }
```

图 C.11: Safe Insert and Lock-Free Search

Linux 内存屏障原语是根据 Alpha 指令来命名的，因此 smp_mb() 名为 mb, smp_rmb() 名为 rmb, smp_wmb() 名为 wmb。 Alpha 是唯一实现 smp_read_barrier_depends() 的 CPU 而其他 CPU 均是空操作。

问题 C.13: 为什么 Alpha 的 smp_read_barrier_depends() 是一个 smp_mb() 而不是 smp_rmb()?

关于 Alpha 的详细情况，请参见参考手册 [SW95]。

C.7.2 AMD64

AMD64 与 x86 是兼容的，最近修改了它的内存模型 [Adv07] ，某些时候可以提供强序的实现。AMD64 的 smp_mb() 原语是 mfence，smp_rmb()是 lfence, smp_wmb() 是 sfence.

C.7.3 ARMv7-A/R

ARM CPU 体系在嵌入式应用中很流行，特别是在电源受限的应用中，如电话。虽然如此，ARM 多核已经存在五年以上的时间了。它的内存模型类似于 Power (参见 C.7.6 节, 但是 ARM 使用了不同的内存屏障指令集 [ARM10]:

DMB (数据内存屏障)导致在屏障前的相同类型的操作，看起来先于屏障后的操作先执行。操作类型可以是所有操作，也可能仅限于写操作(类似于 Alpha wmb 以及 POWER 的 eieio 指令)。另外，ARM 允许三种范围的缓存一致性：单处理器，处理器子集 (``inner'') 以及全局范围内的一致(``outer'')。

DSB (数据同步屏障) 导致相同类型的操作在随后的操作执行前先完成。操作类型与 DMB 相同。在早期的版本中，DSB 指令被 DWB 调用(可以选择是清除写缓冲区还是数据写屏障) 。

ISB (指令同步屏障) 刷新 CPU 流水线，这样所有随后的指令仅仅在 ISB 指令完成后才被读取。例如，如果您编写一个自修改的程序（如 JIT），应当在生成代码及执行代码之间执行一个 ISB 指令。

没有哪一个指令与 LINUX 的 rmb()语义完全相符。因此必须将 rmb()实现为一个全 DMB。DMB 和 DSB 指令有一个递归的定义，与 POWER 的效果类似。

ARM 也实现了控制依赖，因此，如果一个条件分支依赖于一个加载操作，那么在条件分支后面的存储操作都在加载操作后执行。但是，并不保证在条件分支后面的加载操作也是有序的。如下例：

```
1 r1 = x;
2 if (r1 == 0)
3     nop();
4 y = 1;
5 r2 = z;
6 ISB();
7 r3 = z;
```

在这个例子中，存储、加载控制依赖导致在第 1 行的加载 X 操作的顺序在第 4 行的对 Y 的存储操作之前。但是，ARM 并不考虑加载、加载控制依赖，因此，第 1 行的加载也许会在第五行的加载操作后面发生。另一方面，第二行的条件分支与第六行的 ISB 指令确保第七行在第一行后面发生。注意，在第三行和第四行之间插入一个 ISB 指令将确保第一行和第五行之间的顺序。

C.7.4 IA64

IA64 是一个弱的一致性模型。因此，在没有内存屏障指令时，IA64 将武断的重排内存引用 [Int02b]。IA64 有一个 名为 mf 的 memory-fence 指令，但是也

有一个 ``half-memory fence" 用于装载、存储以及用于一些原子指令 [Int02a]。acq 防止 acq 后面的内存引用指令被重排，但是允许前面的内存引用指令被重排，这种奇怪的东西如图 C.12 所示。类似的，rel 防止前面的内存指令被重排，但是允许后面的内存指令被重排到 rel 之前。

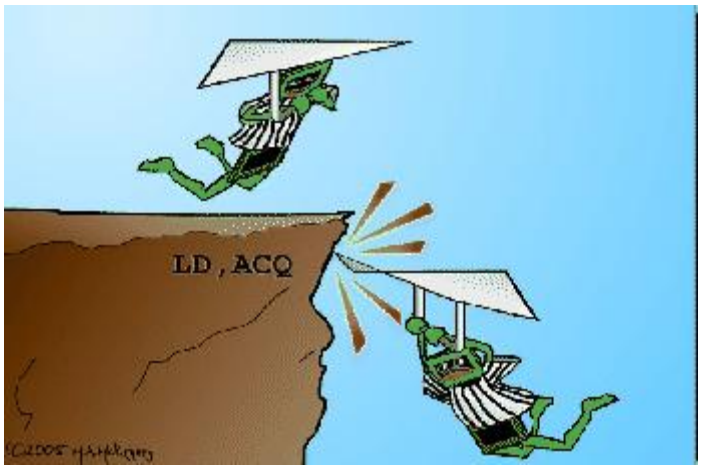


图 C.12: Half Memory Barrier

这些 half-memory fences 对临界段是有用的，因为它可以安全的将操作放入一个临界段。但是，仅仅只有这个 CPUs 有这个属性。

在 LINUX 内核中，IA64 mf 指令被用于 smp_rmb()，smp_mb()和 smp_wmb()。

最后，IA64 提供一个全局的 ``mf"。这提供了“可传递性”的概念，如果一个特定代码段看见一个已经发生的访问操作，随后的代码段将看到更早的内存访问操作。假设所有的代码段都正确的使用了内存屏障.

C.7.5 PA-RISC

虽然 PA-RISC 体系允许重排所有装载和存储操作，实际上 CPUs 是严格按照顺序运行的 [Kan96]。这意味着 Linux 的内存屏障原语是空操作，但是，使用了 GCC 的 memory 属性来禁止编译器优化。

C.7.6 POWER / Power PC

POWER 和 Power PC CPU 有多种内存屏障指令 [IBM94,LSH02]：

sync 导致所有在它之后的操作开始之前，之前的所有操作都已经完成。因此，这个指令的消耗是十分大的。

lwsync (轻量级 sync) 在装载操作和随后的装载和存储操作之间进行排序。但是，它不对随后的装载操作进行排序。非常有趣的是， lwsync 指令强制与

zSeries, SPARC TSO 的顺序相同.

eieio (enforce in-order execution of I/O) 导致前面缓存的存储操作在随后的所有存储操作之前，看起来已经完成。但是，缓存存储与非缓存存储是分别排序的。

isync 强制在随后的指令开始执行前，前面的指令已经完成。

不幸的是，没有哪一个指令与 LINUX 的 wmb() 语义相符，这个原语请求将存储进行排序，但是不需要 sync 的其他高开销的操作。但是没有选择的余地：ppc64 版本的 wmb() 和 mb() 都定义为高开销的 sync 指令。但是，Linux 的 smp_wmb() 从不用于 MMIO (在 UP 上，由于驱动也必须小心的对 MMIO 进行排序)。因此，它被定义为轻量级的 eieio 指令。smp_mb() 也被定义为 sync 指令，但是 smp_rmb() 和 rmb() 被定义为 lwsync 指令.

Power 有 ``cumulativity"属性，用于获得传递性。当正确的使用它后，任何代码，只要它能够看到更早代码的结果，也就能看到这段更早前的代码能够看到的结果。更多详情请参见 McKenney 和 Silvera [MS09]。

Power 也实现了控制依赖。这与 ARM 是非常相似的，有一个例外是：Power isync 指令用于代替 ARM 的 ISB 指令。

很多 POWER 体系的成员有非一致性的指令缓存，因此存储到内存并不必然反映到指令缓存中。可喜的是，很少有人还会写自修改代码。但是 JITs 和编译器会干这些事情。此外，在 CPUs 的角度来看，重新编译一个最近运行的程序也是属于自修改代码。icbi 指令 (instruction cache block invalidate) 从指令缓存中刷新特定的缓存行，可用于这种情况。

C.7.7 SPARC RMO, PSO, and TSO

SPARC 上的 Solaris 使用 TSO (total-store order)，Linux 在`sparc" 32-bit 体系结构也是如此。但是， 64-bit Linux 内核 (``sparc64" 体系) 运行在 RMO (relaxed-memory order) 模式 [SPA94]。SPARC 体系也提供一个中间的 PSO (partial store order)。任何运行在 RMO 的程序都能够运行在 PSO 和 TSO，类似的，运行在 PSO 的程序也能运行在 TSO。移植一个共享内存的并行程序需要小心的加上内存屏障。虽然如前所述，使用标准的同步原语的程序不必担心内存屏障。

SPARC 有非常复杂的内存屏障指令 [SPA94] ，允许五种级别的顺序控制：

- ✓ StoreStore：在存储之间进行排序. (这被用于 LINUX smp_wmb() 原语.)
- ✓ LoadStore：在装载及随后的存储之间排序。

- ✓ StoreLoad：在装载及随后的加载之间排序。
- ✓ LoadLoad：在加载操作之间排序. (被用于 Linux smp_rmb() 原语.)
- ✓ Sync：在开始随后的操作前，全部完成之前的所有操作。
- ✓ MemIssue： 在随后的内存操作完成前，完成所有之前的内存操作。对一些内存映射 IO 来说很重要。
- ✓ Lookaside：与 MemIssue 相同, 但是仅仅应用于先写后读的情况。
- ✓ Linux smp_mb() 原语同时使用前四个指令， #LoadLoad | #LoadStore | #StoreStore | #StoreLoad，因此将内存操作完全排序。

既然这样, 为什么需要#MemIssue? 因为#StoreLoad 允许随后的读操作从写缓冲区中获取值，如果向 MMIO 寄存器写值就惨了。相对的，#MemIssue 在允许读执行前，需要等待写缓冲区被刷新。确保装载操作将从 MMIO 寄存器实际的读到它的值。驱动可以使用 #Sync 代替它，但是轻量级的 #MemIssue 更好。

#Lookaside 是#MemIssue 的轻量级版本，当向特定 MMIO 写入后，紧跟着需要读取它的值时，这个指令是有用的。但是，当写入的 MMIO 寄存器要影响到其他将要读取的寄存器时，重量级的#MemIssue 是必须的。

不清楚为什么 SPARC 不将 wmb()定义为#MemIssue ，将 smb_wmb() 定义为 #StoreStore，当前的定义对某些驱动来说会有 BUG。这是非常有可能的：运行在所有 SPARC CPUs 的 LINUX 定义了比 CPU 允许的、更稳健的内存序模型。

SPARC 在保存指令与执行指令之间需要一个 flush 指令 [SPA94]。这用来从 SPARC 的指令缓存中刷新以前的值。注意： flush 需要一个地址，并且仅仅从指令缓存中刷新指定地址的缓存。在 SMP 系统上，所有 CPU 的缓存都将被刷新，但是没有合适的方法确定其他 CPU 是否完成了刷新。

C.7.8 x86

由于 x86 CPUs 提供 ``process ordering" ，因此所有 CPU 都与 CPU 写内存的顺序一致。smp_wmb() 实现为一个空操作 [Int04b]. 但是，它需要一个编译器指令，以避免编译进行重排。

另一方面， x86 CPUs 传统上不保证装载顺序，smp_mb() 和 smp_rmb() 被解释为 lock;addl. 这个原子指令实际上是一个装载和存储屏障。

最近，Intel 为 x86 发布了一个内存模型 [Int07]。它说明：Intel CPUs 确保比以前的规范要求更严的内存序。更近一段时间，Intel 发布了一个更新内存模型 [Int09, Section 8.2]。要求对存储来说，实现全局序。虽然个别的 CPU 仍然允许当前 CPU 比其他 CPU 更早的看到它们的存储结果。这个特例允许硬件进行重要的优化，包括存储缓冲区的优化。另外，内存序遵从“传递性”，因此，如果 CPU0

看到 CPU1 存储的值，那么 CPU0 也能看到 CPU1 能够看到的，CPU1 以前存储的值。软件使用原子操作会使这些优化无效，这是原子操作比非原子操作开销更大的原因之一。全局存储序在老的处理器上并不能得到保证。

但是，请注意某些 SSE 指令是弱序的(clflush and non-temporal move instructions [Int04a])。有 SSE 的 CPUs 可以用 mfence 实现 smp_mb()，lfence 实现 smp_rmb()，sfence 实现 smp_wmb()。

某些版本的 x86 CPU 有一个模式位，允许在存储之间乱序，在这些 CPUs 上，smp_wmb() 必须定义为 lock;addl。

虽然很多旧的 x86 实现可以适应自修改代码而不需要特殊的指令，但是新版本的 x86 体系不要求 CPUs 适应这一点。这一点会为 JIT 实现带来一点点不方便的地方。

C.7.9 zSeries

zSeries 是 IBM TM 的主要成员，以前著名的有 360、370 和 390 [Int04c]。后来就是 zSeries。bcr 15,0 指令用于 Linux smp_mb()、smp_rmb()和 smp_wmb()原语。这也是非常强的内存序语义，如表 C.5。允许 smp_wmb() 实现为 nop (当您看到这一点的时候，LINUX 内核可能已经将 smp_wmb()修改为 nop 了).

对绝大部分 CPUs 来说，zSeries 不保证指令流与缓存之间的一致性。因此，自修改代码必须在执行前先执行一个 serializing 指令。也就是说，许多 zSeries 机器是适应自修改代码的，不需要 serializing 指令。zSeries 指令集提供很多 serializing 指令，包含 compare-and-swap，某些类型的分支指令 (如前述的 bcr 15,0 指令)，以及 test-and-set，以及其他指令。

C.8 内存屏障是永恒的?

最近，已经有一些系统值得关注，它们较少进行乱序执行，乱序内存引用。这个趋势继续下去的话，内存屏障会不会变成历史?

赞成这个方法的人会拿笨重的多线程硬件体系说事，这样每一个线程都必须等待内存就绪，这可能是数十个、数百个、数千个线程在某一时刻干这件事情。在这样的体系结构中，没有必要再使用内存屏障了。因为一个特定的线程在处理下一条指令前，将简单的等待其他操作完成。由于可能有上千个其他线程，CPU 将被完全利用，没有 CPU 会被浪费。

反对者会说：这限制了将应用程序扩展到上千个线程。在实时系统中，会增加响应时间，对某些应用来说，可能会增加数十微秒。这不能满足实时响应的要求。

谁是对的? 这可没法下定论，因此咱们还是作两手准备吧.

C.9 对硬件设计者的建议

硬件设计者可以做很多事情，这些事情给软件开发者带来了困难。以下是我们在过去遇到的一些事情，希望能够帮助防止在将来出现这些问题：

✓ I/O 设备忽略了缓存一致性。

这个不好的特性将导致从内存中进行 DMAs 会丢失刚从输出缓冲区中对它进行的修改。也导致 CPU 缓存的输入缓冲区在 DMA 完成后被覆盖。要使您的系统在这样的情况下正常工作，您必须在为 IO 设备准备 DMA 缓冲区时，小心的刷新 CPU 缓存。而且，您需要非常小心的避免指针方面的 BUG!

✓ 外部总线不能发送缓存一致性数据

这是上面问题的一个更难缠的变种，导致很多设备—甚至是内存自身—不能遵从缓存一致性。不得不告诉您：在从嵌入式系统转移到多核体系时，不用怀疑，这样的问题将会越来越多。希望这些问题能够在 2015 年得到处理。

✓ 设备中断忽略了缓存一致性。

这说得够天真的了—中断不进行内存引用，还是要进行? 但是假设一个 CPU 有一个分开的缓存，其中一个 bank 非常忙，因此一直战胜了输入缓冲的最后一个缓存行。如果相应的 I/O-complete 中断到达这个 CPU，该 CPU 中引用此这个缓存行的内存引用将返回旧的值，并导致数据错误，随后将引起内核挂起。

✓ 核间中断 (IPIs)忽略了缓存一致性.

在相应的消息缓冲区已经提交到内存之前，IPI 就到达目标 CPU，这可能会有问题。

✓ 上下文切换得到缓存一致性。

如果内存访问乱序太严重，那么上下文切换就很困扰。如果任务从一个 CPU 迁移到另一个 CPU，而源 CPU 上的内存访问在目标 CPU 上还不完全可见，那么任务就会很容易看到变量还是以前的值，这会带来致命的问题。

✓ 过度宽松的模拟器和仿真器

编写模拟器来进行内存乱序是很困难的。因此在这上面运行得很好的软件，在实际硬件上运行时，可能是非常糟糕的。

我们再次支持硬件设计者避免这些问题!

D. RCU 实现

本附录描述几个全功能的产品级品质的 RCU 实现。理解这些实现需要彻底理解 第 1 章和第 8 章，而且对 LINUX 内核需要有相当深刻的了解。后者可以在几个课本及网站上找到 [BC05,CRKH05,Cor08,Lov05]。

如果你对 RCU 还很陌生，则应当从简单的“toy”RCU 实现开始，可以在第 8.3.4 节找到它.。

D.1 节描述了``Sleepable RCU"，或者叫 SRCU，允许 SRCU 读者随意睡眠。这是一个简单的实现，也是一个产品级的实现。可以从它开始学习这些 RCU 实现。

D.2 节给出了分组 RCU 的高扩展性实现的概况。分组 RCU 设计用于支持上千个 CPU 这样的 SMP 系统。

D.3 节带领读者阅读这些实现的代码（迟于 2008）。

最后, D.4 节提供了可抢占 RCU 实现的详细视图，它用于实时系统。

D.1 可睡眠 RCU 实现

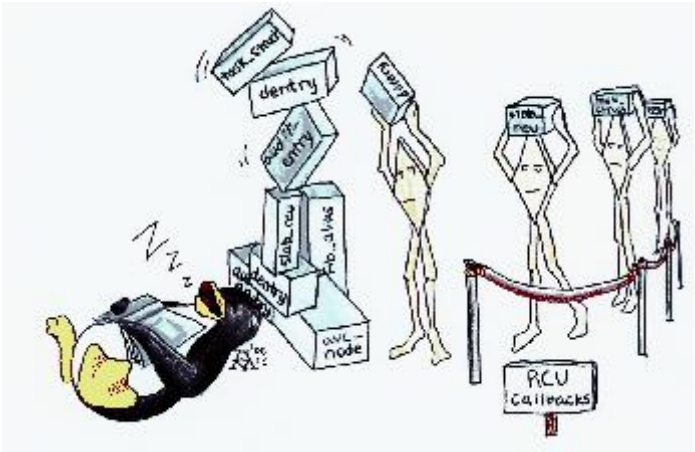


图 D.1: Sleeping While RCU Reading Considered Harmful

经典 RCU 要求读临界区遵从与自旋锁临界区相同的规则：任何类型的阻塞或者睡眠都是严格禁止的。这常常阻碍了 RCU 的使用，Paul 已经收到大量“可睡眠 RCU”的请求，以允许在 RCU 读临界区中可以任意睡眠。以前 Paul 以难以实现的理由拒绝了所有这些请求，这导致在 grace period 结束时，大量的内存等待释放。最终会导致不幸的后果，如图 D.1 所示，几乎所有的不幸都是由于内

存耗尽而将系统挂起。最终，任何一个一致性控制原语可能导致系统挂起—甚至在正确的使用它们时-这种情况不应当存在。

但是，实时内核要求自旋锁临界区可被抢占 [Mol05]，也要求 RCU 读临界区也可以被抢占 [MS05]。因而，可抢占临界区由于需要获取锁的原语而阻塞，以避免死锁。这意味着 RCU 和自旋锁临界区可以阻塞。但是，这两种形式的睡眠有优先级提升和继承这样的特殊属性，这样可以依优先级唤醒睡眠的进程。

然而，在内核中使用 RCU，还是强烈的要求“RCU 读临界区永不阻塞”。也就是说，象等待 TCP 入连接这样的无限睡眠是严格禁止的，即使是在实时内核中也是如此。

问题 D.1：为什么在经典的 RCU 读临界区中禁止睡眠?

问题 D.2：为什么不通过在静止状态中排除上下文切换来允许在经典 RCU 中睡眠，而仅仅在静止状态中保留用户态执行及 IDLE 循环两种状态?

D.1.1 SRCU 实现原理

设计 SRCU 的主要难题是：防止处于 RCU 读临界区中的任务睡眠时，阻塞大量的 RCU 回调函数。SRCU 使用两个策略实现这个目的：

- ✓ 不再提供 grace-period 接口，如经典 RCU 中的 call_rcu() API, 以及
- ✓ 在每一个使用 SRCU 的子系统中分开对 grace-period 检测。

在随后的章节中讨论这些策略的基本原理。

D.1.1.1 废除 Grace-Period APIs

call_rcu() API 的问题是：单个线程可以生成这样的代码：在任意数量的内存块上等待 grace period ，举例如下:

```
1 while (p = kmalloc(sizeof(*p), GFP_ATOMIC))
2     call_rcu(&p->rcu, f);
```

相对的, 使用类似的 synchronize_rcu()的话，每个线程在等待 grace period 时，只会阻塞在一个单一的内存块上：

```
1 while (p = kmalloc(sizeof(*p),
2                    GFP_ATOMIC)) {
3     synchronize_rcu();
4     kfree(&p->rcu, f);
5 }
```

因此，SRCU 提供一个类似于 synchronize_rcu()的 API，但是不再提供 call_rcu()。

D.1.1.2 隔离 Grace-Period 检测

在经典 RCU 中，单一的读临界区可以不确定的延迟所有 RCU 回调，如下所示：

```
1 /* BUGGY: Do not use!! */
2 rcu_read_lock();
3 schedule_timeout_interruptible(longdelay);
4 rcu_read_unlock();
```

如果 RCU 仅仅用在单个子系统中，并且小心的设计以经得起长时间的延时，那么这类行为是可以容许的。实际上，某个子系统中的读临界区 BUG 可以将所有 RCU 用户延迟，导致这些长时间的 RCU 读延迟被废除。

解决这个问题的一个方法是：基于某个子系统进行 grace-period 检测。这样，一个长时间睡眠的 RCU 读者仅仅延迟它所在的子系统的 grace periods。因为每一个子系统仅仅拥有有限数量的内存块阻塞在它的 grace period 上，并且子系统的数量也是有限的，那么在一个 grace period 周期内等待的内存数量也是有限的。特定子系统的设计者有责任实现第一点，以确保 SRCU 读端的睡眠是有限制的，并有责任实现第二点限制等待在 synchronize_srcu()上的的内存数量。

SRCU 采用的具体的方法，将在随后的章节中描述。

D.1.2 SRCU API 及用法

SRCU API 如图 D.2 所示。随后的章节描述如何使用它们。

```
int init_srcu_struct(struct srcu_struct *sp);
void cleanup_srcu_struct(struct srcu_struct *sp);
int srcu_read_lock(struct srcu_struct *sp);
void srcu_read_unlock(struct srcu_struct *sp, int idx);
void synchronize_srcu(struct srcu_struct *sp);
long srcu_batches_completed(struct srcu_struct *sp);
```

图 D.2: SRCU API

D.1.2.1 初始化及退出

每一个使用 SRCU 的子系统必须创建一个 struct srcu_struct，要么定义一个这种类型的变量，要么动态的分配内存。例如通过 kmalloc()分配内存。一旦这

个结构存在，就可通过 init_srcu_struct()进行初始化，如果成功将返回 0，错误则返回一个错误码（如内存耗尽）。

如果 struct srcu_struct 是动态分配的，那么必须在释放前调用 cleanup_srcu_struct()。类似的，如果 struct srcu_struct 是 LINUX 内核模块中定义的变量，那么也必须在模块被卸载前调用 cleanup_srcu_struct()。另一方面，调用者必须小心确保所有的 SRCU 读临界区都已经在调用 cleanup_srcu_struct()前完成(并且没有更多的读临界区开始运行)。实现这一点的方法在 D.1.2.4 节中描述。

D.1.2.2 读端原语

读端 srcu_read_lock() 和 srcu_read_unlock() 原语可以按如下方法使用：

```
1 idx = srcu_read_lock(&ss);
2 /* read-side critical section. */
3 srcu_read_unlock(&ss, idx);
```

ss 变量是在 D.1.2.1 节中初始化的 struct srcu_struct，idx 变量是一个整数值，用来告诉 srcu_read_unlock()：相应的 srcu_read_lock()开始于哪一个 grace period。

传递索引值这一点，是与 RCU API 不一致的，当需要时，可以在任务结构中保存这个信息。但是，由于一个特定的任务可能处于多个嵌套的 SRCU 读临界区中，因此，SRCU 不能在任务结构中适当的保存这个索引。

D.1.2.3 写端原语

synchronize_srcu() 原语可以按如下方式使用:

```
1 list_del_rcu(p);
2 synchronize_srcu(&ss);
3 kfree(p);
```

与经典 RCU 类似，这个原语会一直阻塞，直到所有在 synchronize_srcu()之前开始的 SRCU 读临界区都全部完成。如表 D.1 所示。在这个表中，CPU 1 仅仅需要 CPU0 的读临界区完成，而不必等待 CPU2 上的读临界区完成，因为 CPU2 在 CPU1 执行 executing synchronize_srcu()前，没有开始它的临界区。最后，CPU 1 的 synchronize_srcu()不必等待 CPU3 的读临界区，因为 CPU3 使用 s2 而不是 s1 作为它的 struct srcu_struct。这样，CPU 3 的 SRCU 读临界区与不同的 grace periods 相关联。

表 D.1: SRCU Update and Read-Side Critical

Sections

	CPU 0	CPU 1	CPU 2	CPU 3
	i0 = srcu_read_lock(	s1)		
		synchronize_srcu(	s1) enter	
			i2 = srcu_read_lock(	s1)
	srcu_read_unlock(	s1, i0)		
		synchronize_srcu(	s1) exit	
			srcu_read_unlock(	s1, i2)

srcu_batches_completed() 原语可以被用于监控一个特定的 grace periods 的处理过程。这个原语用于验证 SRCU 操作的``torture tests"。

D.1.2.4 安全退出

安全的退出 SRCU 可能是一个挑战，幸运的是，大多数用户不必做这件事情。例如，使用操作系统在启动时初始化的 SRCU，而不必清除它们。但是，在可加载模块中使用 SRCU 的话，就必须在模块被安全卸载前清除它们。

有些情况下（如 RCU 压力测试模块），仅仅少数已知的线程使用特定 struct srcu_struct 对应的 SRCU 读临界区。在这些情况下，模块退出函数仅仅需要杀死相应的线程，等待它们退出，然后退出。

其他情况下（例如设备驱动），系统中的任何线程都可以使用 SRCU 读端原语。虽然可以使用前面所述的方法，但是这相当于是做了一次复位操作，这种方法不可取。图 D.3 显示了一种方法进行清除工作而不用复位：

```
1 int readside(void)
2 {
3 int idx;
4
```

```
5 rcu_read_lock();
6 if (nomoresrcu) {
7 rcu_read_unlock();
8 return -EINVAL;
9 }
10 idx = srcu_read_lock(&ss);
11 rcu_read_unlock();
12 /* SRCU read-side critical
section. */
13 srcu_read_unlock(&ss, idx);
14 return 0;
15 }
16
17 void cleanup(void)
18 {
19 nomoresrcu = 1;
20 synchronize_rcu();
21 synchronize_srcu(&ss);
22 cleanup_srcu_struct(&ss);
23 }
```

图 D.3: SRCU Safe Cleanup

readside() 函数将 RCU 和 SRCU 读临界区交错运行，RCU 运行在第 5-11 行，后面的 SRCU 运行在第 10-13 行。RCU 读临界区使用 RCU [McK04] 以保护 nomoresrcu 值。如果设置了这个值，表示我们准备退出，因此必须不能进入 SRCU 读临界区，这样返回-EINVAL。另一方面，如果我们没有退出，我们就进入 SRCU 读临界区。

cleanup() 函数首先在第 19 行设置 nomoresrcu 的值，随后必须通过第 20 行的 synchronize_rcu()原语等待当前所有正在运行的 RCU 读临界区完成。一旦 cleanup() 函数到达第 21 行，所有调用 readside() 的函数看到 nomorersrcu 等于 0 时，都必然已经到达第 11 行，因此它们必须进入 SRCU 读临界区。后面调用 readside() 将在第 8 行退出，这将使它们不再进入读临界区。

因此，一旦 cleanup() 完成第 21 行的 synchronize_srcu()，所有 SRCU 读临界区已经完成，并且没有新的任务进入读临界区。因此在第 22 行安全的调用 cleanup_srcu_struct()。

D.1.3 实现

本节描述 SRCU 的数据结构，初始化和清除原语，读端原语，写端原语。

D.1.3.1 数据结构

SRCU 的数据结构如图 D.4 所示。Completed 字段是自 struct srcu 初始化以来，grace periods 周期的数量，如图 D.5 所示。它的低位用于 struct srcu_struct_array 的索引。per_cpu_ref 字段指向这个数组， mutex 字段用于某个时刻仅仅允许处理一个 synchronize_srcu()。

```
1 struct srcu_struct_array {
2     int c[2];
3 };
4 struct srcu_struct {
5     int completed;
6     struct srcu_struct_array *per_cpu_ref;
7     struct mutex mutex;
8 };
```

图 D.4: SRCU Data Structures

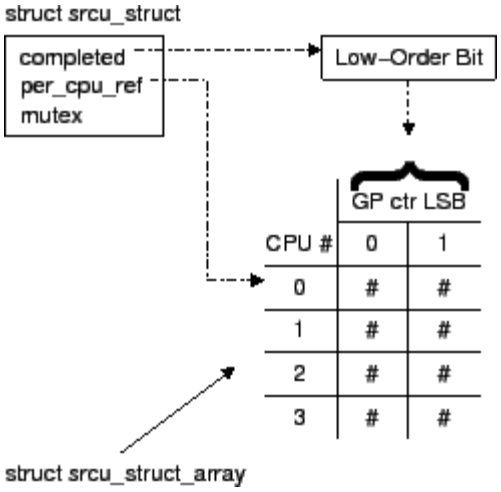


图 D.5: SRCU Data-Structure Diagram

D.1.3.2 初始化实现

SRCU 的初始化函数 init_srcu_struct()如图 D.6 所示。这个函数简单的初始化 struct srcu_struct，如果初始化成功就返回 0，否则返回 –ENOMEM。

```
1 int init_srcu_struct(struct srcu_struct *sp)
2 {
3     sp->completed = 0;
```

```
4     mutex_init(&sp->mutex);
5     sp->per_cpu_ref =
6     alloc_percpu(struct srcu_struct_array);
7     return (sp->per_cpu_ref ? 0 : -ENOMEM);
8 }
```

图 D.6: SRCU Initialization

SRCU 的清除函数如图 D.7。主要的清除函数 cleanup_srcu_struct()在本图的第 19-29 行。但是，立即调用 srcu_readers_active()，如图中第 13-17 行所示，以验证当前没有读者在使用 struct srcu_struct。

srcu_readers_active() 函数简单的返回所有可能索引上的 srcu_readers_active_idx()的总和。srcu_readers_active_idx()如第 1-11 所示，计算相应索引上的每 CPU 计数值的总和，然后返回结果。

如果 srcu_readers_active() 的返回值非 0，cleanup_srcu_struct()在第 24 行触发一个警告，并在第 25、26 行简单的返回。并且不释放正在使用的 struct srcu_struct。这样的警告总是表示存在一个 BUG，允许系统存在一点点内存泄漏而继续运行，这比可能的野指针访问要好一些。

否则，cleanup_srcu_struct() 在第 27、28 行释放 per-CPU 计数数组，并将指针设置为 NULL。

```
1 int srcu_readers_active_idx(struct srcu_struct *sp,
2 int idx)
3 {
4     int cpu;
5     int sum;
6
7     sum = 0;
8     for_each_possible_cpu(cpu)
9     sum += per_cpu_ptr(sp->per_cpu_ref, cpu)->c[idx];
10    return sum;
11 }
12
13 int srcu_readers_active(struct srcu_struct *sp)
14 {
15    return srcu_readers_active_idx(sp, 0) +
16    srcu_readers_active_idx(sp, 1);
17 }
18
19 void cleanup_srcu_struct(struct srcu_struct *sp)
20 {
```

```
21  int sum;
22
23  sum = srcu_readers_active(sp);
24  WARN_ON(sum);
25  if (sum != 0)
26          return;
27  free_percpu(sp->per_cpu_ref);
28  sp->per_cpu_ref = NULL;
29 }
```

图 D.7: SRCU Cleanup

D.1.3.3 读端实现

srcu_read_lock()的实现代码如图 D.8 所示。这个函数被小心的构建，以避免内存屏障和原子指令。

第 5、11 行禁止并打开抢占，以强制代码的执行顺序在单 CPU 上不能被抢占。第 6 行取得 grace-period 计数器的低位，它用来选择本 SRCU 读临界区使用哪一个每 CPU 计数器。第 7 行调用的 barrier() 指示编译器确保一次性获取索引号，这样第 9 行使用的索引号和第 11 行返回的索引号将是同一个值。第 8-9 行递增选中的计数器。D.3 第 10 行强制后面的执行在第 8-9 行以后发生，以防止在没有配置 CONFIG_PREEMPT 时，造成任何代码乱序。但是，在一个配置了 CONFIG_PREEMPT 的内核中，需要的 barrier() 调用已经包含在第 11 行的 preempt_enable() 中，因此，srcu_barrier()是一个空操作。最后，第 12 行返回索引号，该索引号被传递给相应的 srcu_read_unlock()。

```
1 int srcu_read_lock(struct srcu_struct *sp)
2 {
3   int idx;
4
5   preempt_disable();
6     idx = sp->completed & 0x1;
7   barrier();
8   per_cpu_ptr(sp->per_cpu_ref,
9   smp_processor_id())->c[idx]++;
10  srcu_barrier();
11    preempt_enable();
12    return idx;
13 }
```

图 D.8: SRCU Read-Side Acquisition

srcu_read_unlock() 的代码如图 D.9 所示。同样的，第 3 、 7 行禁止并打开抢占，这样，整个代码顺序在单核上将不可被抢占。在 CONFIG_PREEMPT 内核中，第 3 行的 preempt_disable() 包含一个 barrier() 原语。否则， barrier() 在第 4 行被调用。同样的，这使得在临界区后面执行的代码看起来象是插入了一个中断。第 5 、6 行递增本 CPU 的计数值，使用的索引号与相应的 srcu_read_lock() 是一样的。

```
1 void srcu_read_unlock(struct srcu_struct *sp, int idx)
2 {
3   preempt_disable();
4   srcu_barrier();
5   per_cpu_ptr(sp->per_cpu_ref,
6   smp_processor_id())->c[idx]--;
7   preempt_enable();
8 }
```

图 D.9: SRCU Read-Side Release

关键的一点是特定 CPU 的计数值仅仅能被其他 CPU 通过 CPU 中断处理程序看到。这些中断处理函数有责任确保所需的内存屏障得到执行。

D.1.3.4 写端实现

SRCU 的关键点是 synchronize_sched() 将会阻塞，直到所有当前执行在禁止抢占范围内的代码完成。synchronize_srcu() 原语利用了这个效果，如图 D.10。

第 5 行获得 grace-period 计数值的快照。第 6 行获得互斥锁，第 7-10 行检查自获得快照以来，是否至少经历了两个 grace-period。如果是这样，则释放锁并返回。在这种情况下，其他地方已经为我们做了相同的事情。否则，第 11 行确保任何其他 CPU 看到在 srcu_read_lock 中 grace-period 的递增值，也看到本 CPU 在进入 srcu_read_lock 前的任何变化。

第 12 行获得 grace-period 计数器的低位，用于以后作为每 CPU 计数变量的索引，第 13 行递增 grace-period 计数。第 14 行等待当前执行的 srcu_read_lock() 完成。这样，当运行到第 15 行时，所有 srcu_read_lock()实例都会使用

sp->completed 的新值。因此，第 15 行通过 srcu_readers_active_idx() 进行计数采样确保是单调递减的，这样当总和达到 0 时，它确保读者数不会再变化。

但是，srcu_read_unlock() 原语没有内存屏障，so the CPU is within its rights to reorder the counter decrement up into the SRCU critical section, so that references to an SRCU-protected data structure could in effect ``bleed out" of the SRCU critical section. 这个情况由 synchronize_sched() 第 17 行处理，它将阻塞直到其他在 preempt_disable() 中执行的代码完成 (正如 srcu_read_unlock()中一样)。由于特定的 preempt_disable() 代码序列是否完成是被 CPU 所注意到的，完成这些代码意味着以前的 SRCU 读临界区已经完成。任何需要的内存屏障都由这些代码实现了。

此时，在第 18 行释放 mutex 并返回调用者是安全的，调用者现在可以假定所有共享相同的 struct srcu_struc 的 SRCU 读临界区将观察到所有在调用 synchronize_srcu()之前进行的更新。

```
1 void synchronize_srcu(struct srcu_struct *sp)
2 {
3    int idx;
4
5    idx = sp->completed;
6    mutex_lock(&sp->mutex);
7    if ((sp->completed - idx) >= 2) {
8            mutex_unlock(&sp->mutex);
9    return;
10 }
11 synchronize_sched();
12   idx = sp->completed & 0x1;
13   sp->completed++;
14   synchronize_sched();
15   while (srcu_readers_active_idx(sp, idx))
16           schedule_timeout_interruptible(1);
17   synchronize_sched();
18   mutex_unlock(&sp->mutex);
19 }
```

图 D.10: SRCU Update-Side Implementation

问题 D.3：为什么我们可以假设：由 synchronize_sched()分开的更新将按顺序执行?

问题 D.4：为什么在第 18 行释放 mutex 前必须要在第 17 行执行 synchronize_srcu() (图 D.10)? 要交换这两行必须做什么修改? 这样的修改值得吗? 为什么?

D.1.4 SRCU 概述

SRCU 提供一种 RCU-like 原语集，以允许在 SRCU 读临界区中睡眠。但是，请注意 SRCU 仅仅被用于原型代码，虽然它经过了 RCU 压力测试。看看它使用了什么是有意思的。

D.2 分级 RCU 概述

虽然经典的 RCU 读端原语拥有出色的性能和扩展性，但是写端原语仅仅设计用于数十个 CPU，它判断预先存在的读临界区在什么时候完成。至少在每个 grace period，每个 CPU 必须获取一个全局锁，这使得它们的扩展性受到了限制。虽然经典 RCU 实际上能够比较困难的扩展到上千个 CPUs (但是增加了 grace periods 的代价)，多核系统需要更好的扩展它。

另外，经典 RCU 有一个不是最优的 dynticks 接口，导致经典 RCU 在每一个 grace period 都要唤醒每一个 CPU。要明白这个问题，考虑 16 核的系统，它只有四个 CPU 比较忙，其他 CPU 的负载都很轻。理想情况下，余下 12 个 CPUs 可以处于深度睡眠模式以节约能源。不幸的是，如果四个忙的 CPU 频繁的执行 RCU 更新，这 12 个空闲 CPU 会被周期性的唤醒，浪费了重要的能源。因此，对于经典 RCU 的任何主要变化都应当让睡眠的 CPU 处于睡眠状态。

经典 RCU 和分级 RCU 实现都有和经典 RCU 的语义相同的 APIs。但是，原有的实现被称为“经典 RCU”，新实现被称为“分级 RCU”。

@@@ roadmap @@@

D.2.1 RCU 基础回顾

从最基本的方面来说，RCU 是一种等待事务完成的方式。当然，还存在很多其他方法，包括引用计数，读写锁，事件等等。RCU 的一个大的优势是可以等待 20,000 个不同的事件，而不必明确的跟踪其中每一个事件，并且不用担心性能被降低，以及扩展性被限制，也不用担心复杂的死锁情况和内存泄漏的危险。

在 RCU 中，等待的事件被称为 "RCU 读临界区"。RCU 读临界区以 rcu_read_lock() 原语开始，以相应的 rcu_read_unlock() 原语结束。RCU 读临界

区可以嵌套，也可以包含相当多的代码，只要这些代码不阻塞或者睡眠(虽然有一种特殊的名为 SRCU 的 RCU，在 D.1 节中描述的，允许在 SRCU 读临界区中进行普通的睡眠)。如果您遵从这些约束，您可以使用 RCU 来等待任何想要的代码片段完成。

RCU 通过间接的确定其他事务何时完成来实现这一点。在其他地方描述了经典 RCU [MS98]，在 D.4 节描述了可抢占 RCU。

特别的，如图 8.11 中所示，RCU 是一种等待每一个存在的 RCU 读临界区完全完成的方法，也包含了这些临界区中执行的内存操作。

但是，请注意在特定的 grace period 周期后面开始的 RCU 读临界区能够、也将延伸 grace period 周期的结束点。

随后的章节给出了经典 RCU 实现的高级视图。

D.2.2 经典 RCU 实现概要

经典 RCU 实现的关键原理是：经典 RCU 读临界区限制内核代码不允许阻塞。这意味着在任意时刻，一个特定的 CPU 只要看起来处于阻塞状态、IDLE 循环、或者离开了内核后，我就知道所有 RCU 读临界区已经完成。这些状态被称为“静止状态”，当每一个 CPU 已经经历过至少一次静止状态时， RCU grace period 结束。

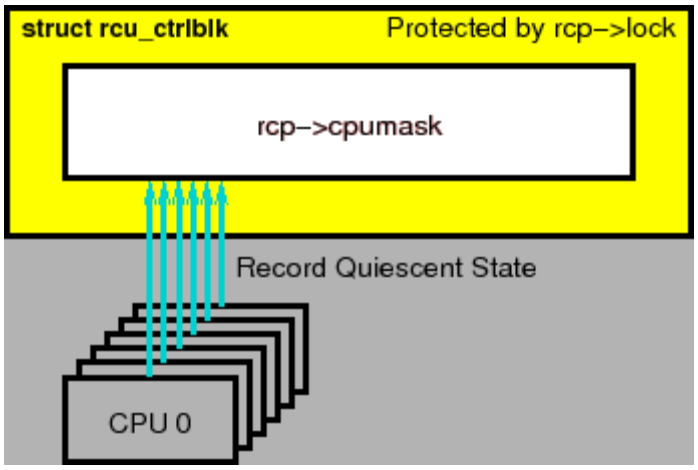


图 D.11: Flat Classic RCU State

经典 RCU 最重要的数据结构是 rcu_ctrlblk，包含了->cpumask 字段，每一个 CPU 包含一位，如图 D.11 所示。当每一个 grace period 开始时，每一个 CPU 相应的位被设置为 1，每一个 CPU 经过一次静止状态时，它必须清除相应的位。由于多个 CPU 可能希望同时清除它们的位，这将破坏->cpumask 字段，使用了一个 ->lock 自旋锁来保护 ->cpumask。不幸的是，当超过几千个 CPU 时，这个

自旋锁也备受争议。更糟糕的是，事实上所有 CPUs 必须清除它们的位，意味着在一个 grace period 内，CPU 不允许睡眠。这限制了 LINUX 节能的能力。

下一节将要展示一个新的非实时 RCU 实现需要什么？

D.2.3 RCU 迫切要解决的问题

实时 RCU 迫切要解决的问题列表 [MS05] 是一个好的开始：

- ✓ 延迟销毁, 这样一个 RCU grace period 不能结束，直到所有已经预先存在 RCU 读临界区已经完成。
- ✓ 可靠的，这样 RCU 支持 24x7 操作。
- ✓ 可以在 IRQ 处理函数中调用。
- ✓ 包含内存印记，这样，如果有很多回调过程，这种机制将加快 grace periods。 (在 LCA2005 列表中将削弱它.)
- ✓ 独立的内存块，这样 RCU 能够基于想要的内存分配工作。
- ✓ Synchronization-free 读端，这样仅仅允许正常的非原子指令操作于 CPU 或者任务局部内存。(对于 LCA2005 列表来说更是这样.)
- ✓ 无条件的 read-to-write 提升，这 LINUX 内核有几个地方需要这样使用。
- ✓ 兼容的 API.
- ✓ 由于这不是实时 RCU, 抢占 RCU 读临界区的要求可以被去掉。但是，我们需要增加以下新要求，以记录过去几年的变化。
- ✓ 极低的 internal-to-RCU 锁的扩展性争论。RCU 必须支持至少 1,024 CPUs，最好是至少 4,096 个 CPU。
- ✓ 节能： RCU 必须能够避免唤醒低电源状态的 dynticks-idle CPUs, 但是仍然能够判断当前的 grace period 何时结束. 这已经在实时 RCU 中实现，但是需要认真的简化。
- ✓ RCU 读临界区必须允许在 NMI 处理函数中使用，就如在中断处理函数中一样。注意，可抢占 RCU 能够避免这个需求，这是由于单独的实现了 synchronize_sched()。
- ✓ RCU 必须很好的管理不停的 CPU 热插拔操作。
- ✓ 必须能够等待所有事先注册的 RCU 回调完成，虽然这已经以 rcu_barrier() 的形式提供。
- ✓ 检测失去响应的 CPUs 是值得的，以帮助诊断 RCU 和死循环 BUG 及硬件错误，这能够防止 RCU grace periods 不能结束的情况。
- ✓ 加快 RCU grace periods 是值得的, 这样 RCU grace period 能够强制在数百微秒内完成，但是，这样的操作预期会带来严重的 CPU 负载。

最急迫的需求是第一条，可扩展性。因此下一节描述如何减少 RCU 的内部锁。

D.2.4 可扩展 RCU 实现

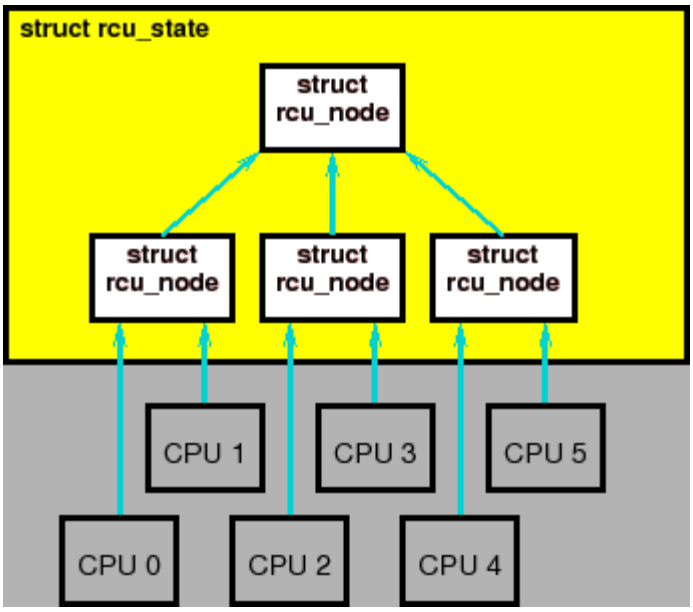


图 D.12: Hierarchical RCU State

减少锁竞争的一个有效方法是创建一个分级结构，如图 D.12 所示。在此，四个 rcu_node 结构中的每一个都有自己的锁，这样仅仅 CPUs 0 和 1 会获取最左边的 rcu_node 的锁，仅仅 CPUs 2 和 3 会获取中间的 rcu_node 的锁，仅仅 CPUs 4 和 5 会申请右边的 rcu_node 的锁。在任何 grace period 期间，仅仅某一个（xie.baoyou 注：经典 RCU 中每一个 CPU 会竞争一个全局锁，这里强调仅仅某一个，就排除了另外一个 CPU 去竞争上一层的锁）访问 rcu_node 结构的上一层的 rcu_node。

最终结果是减少了锁的竞争：不再是 6 个 CPUs 在每一个 grace period 内竞争同一个锁，而是竞争三个上层的 rcu_node 锁 (降低了 50%) ，并竞争两个下层的 rcu_nodes 锁(减少了 67%)。

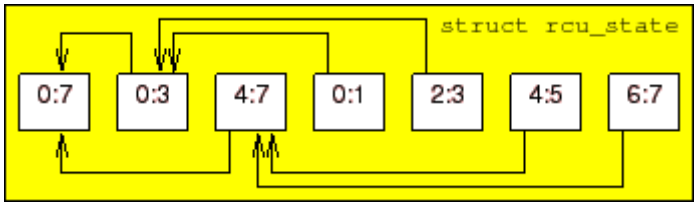


图 D.13: Mapping `rcu_node` Hierarchy Into Array

rcu_node 结构树被嵌入到 rcu_state 结构的一个线性数组，树根是结点 0，

如图 D.13 是一个 8-CPU 的系统。 每一个箭头链接一个 rcu_node 结构到它的父结点。与 rcu_node's ->parent 字段一致。每一个 rcu_node 标示 CPUs 覆盖范围，这样根结点覆盖了所有 CPUs，每一个二级结点覆盖了一半的 CPUs，每一个叶子结点覆盖了两个 CPUs。这个数组在编译时基于 NR_CPUS 的值静态分配。

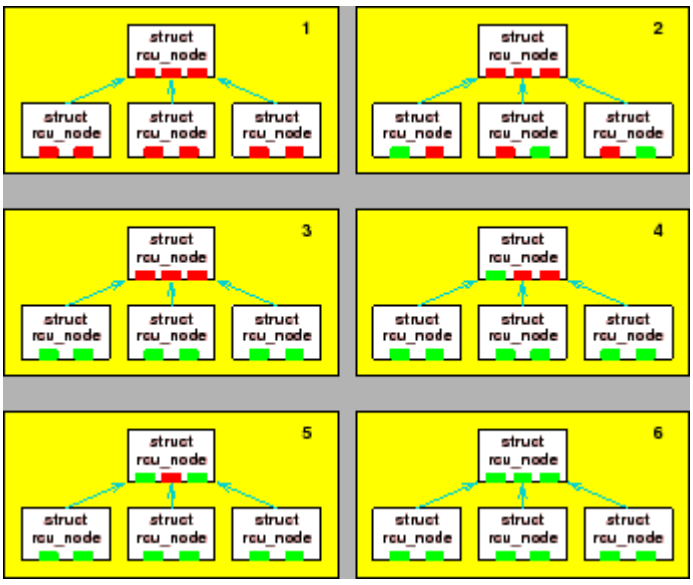


图 D.14: Hierarchical RCU Grace Period

图 D.14 显示了如何检测 grace periods 周期。在第一个图中，没有 CPU 经过静止状态，并用红块标示。假设所有 6 个 CPUs 同时试图告诉 RCU，它们已经经过一个静止状态。仅仅其中一对 CPU 能够获得低层的锁，如果 CPUs0、3、5 比较幸运，则第二图显示了其结果，标识为绿色块。一旦这些幸运的 CPU 完成了，那么其他 CPUs 将获得锁，如图 3 所示。每一个 CPU 将会发现它们是组内最后一个 CPU，因此所有三个 CPU 尝试移到上层 rcu_node。仅仅其中一个能获得上层 rcu_node 锁。假设 CPU1、2、4 依次获得了锁，第 4、5、6 图显示了相应的状态。最后第 6 图显示了所有 CPUs 已经经过一次静止状态，因此 grace period 结束。

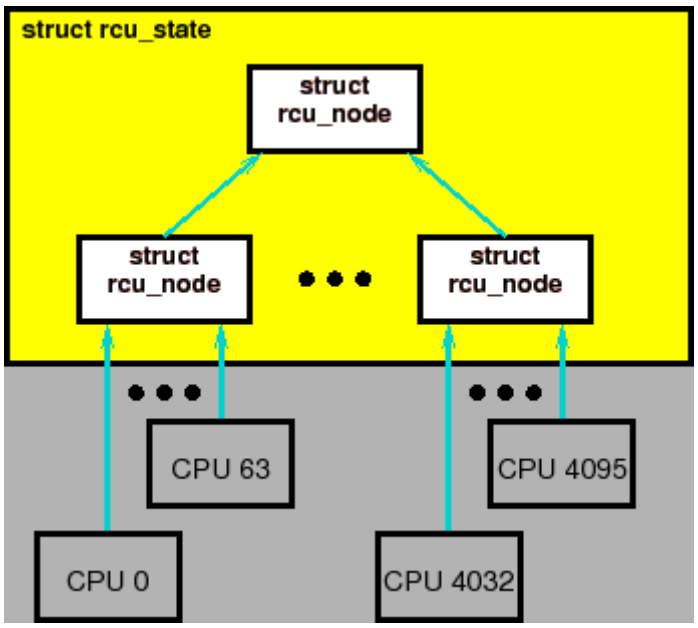


图 D.15: Hierarchical RCU State 4,096 CPUs

在上面的顺序中，没有超过 3 个 CPUs 为同一个锁产生竞争，与经典 RCU 进行对比，我们会高兴的发现，经典 RCU 中，所有 6 个 CPUs 可能冲突。但是，对更多的 CPU 来说，可以再减少锁之间的冲突。考虑有 64 个低级结构及 64*64=4,096 CPUs 个分组结构，如图 D.15。

在此，每一个低级 rcu_node 结构的锁被 64 个 CPUs 申请，将从经典 RCU 的 4096 个 CPUs 竞争一个单一的锁降为 64。在一个特定的 grace period 期间，仅仅一个低级 rcu_node 中的某一个 CPU 会申请上级 rcu_node 的锁。

问题 D.5：等一下！对这些锁来说，如何避免死锁?

问题 D.6：为什么最多减少 64 次？为什么不更多一点?

问题 D.7：为什么我不喜欢 McKenney 在问题 2 的答复中的不完善的理由!!!我希望将单一锁的竞争降低到更合理的地步，比如 16!!!

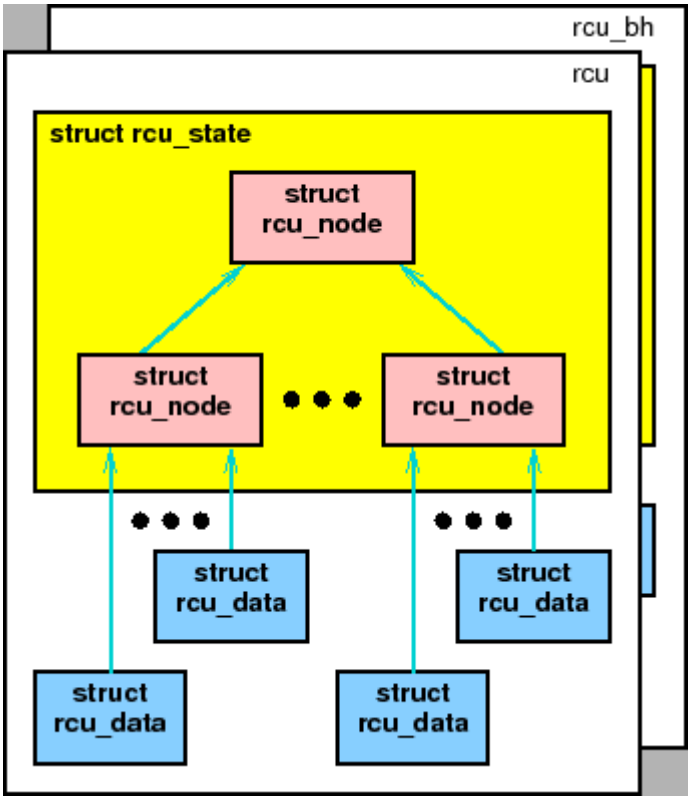


图 D.16: Hierarchical RCU State With BH

这个实现维护了一些 per-CPU 数据，如 RCU 回调列表，被组织在 rcu_data 结构中。另外， rcu (在 call_rcu()中) 和 rcu_bh (在 call_rcu_bh()中) 各自维护它们的分级结构，如图 D.16。

问题 D.8：OK, 这些颜色是干什么的?

下一节讨论节能。

D.2.5 迈向不成熟的 RCU 实现

正如较早前提示的一样，这些努力的一个重要目的是使一个处于睡眠状态的 CPU 保持它的状态，以促进节约能源。与之相对的，经典 RCU 至少会在一个 er grace period 周期内唤醒每一个 CPU。当很少的 CPUs 处于忙的状态，而其他大多数 CPU 都处于空闲状态时，这种处理方法不是最优的。这种情形将系统周期性的陷入高负载。而且，我们必须修复一个长期以来就存在的一个 BUG：一个处于 dynticks-idle 的 CPU 运行一个包含长时间运行的 RCU 读临界区的中断处理函数时，将不能阻止一个正处于结束状态的 RCU grace period（xie.baoyou：这是指一个困扰了作者几个月的 BUG。噫吁嘘，内核之难难于上青天。后面将看到，代码审查是一件多么重要的工作!）。

问题 D.9：对于这样一个不同寻常的 BUG，为什么 linux 能够运行?

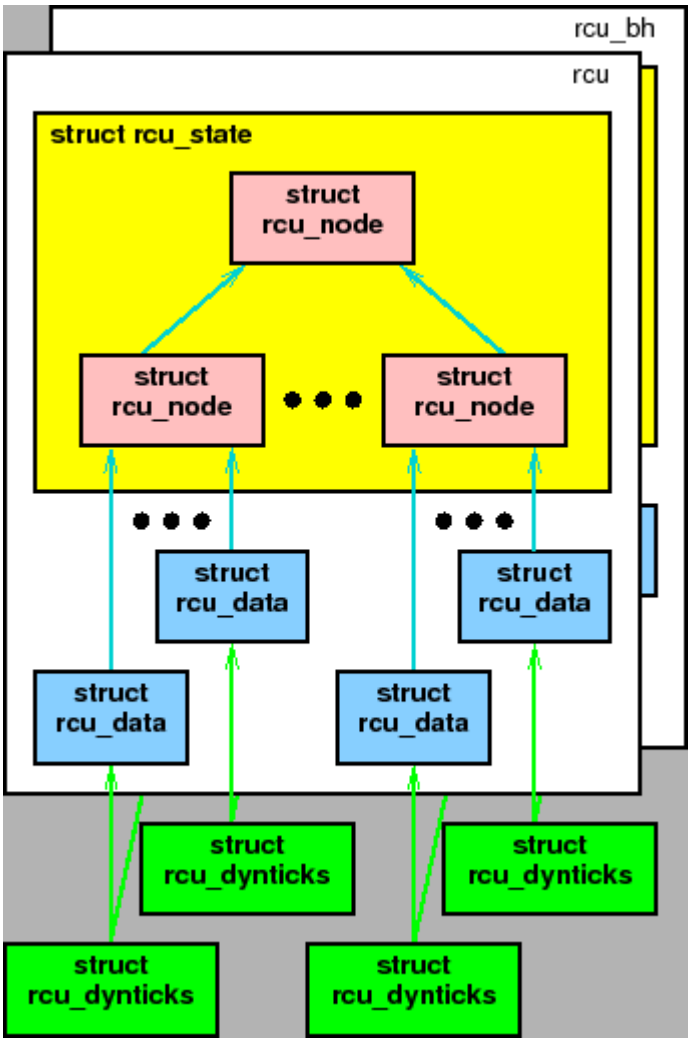


图 D.17: Hierarchical RCU State With Dynticks

这是通过要求所有 CPUs 操作位于一个 per-CPU rcu_dynticks 结构中的计数器来实现的。不严谨的讲，当相应的 CPU 处于 dynticks idle 模式时，计数器的值为偶数，否则是奇数。这样，RCU 仅仅需要等待 rcu_dynticks 计数值为奇数的 CPUs 经过静止状态，而不必唤醒正在睡眠的 CPUs。如图 D.17，每一个 per-CPU rcu_dynticks 结构被 ``rcu'' 和 ``rcu_bh'' 共享。

后面的章节描述 RCU 状态机视图。

D.2.6 状态机

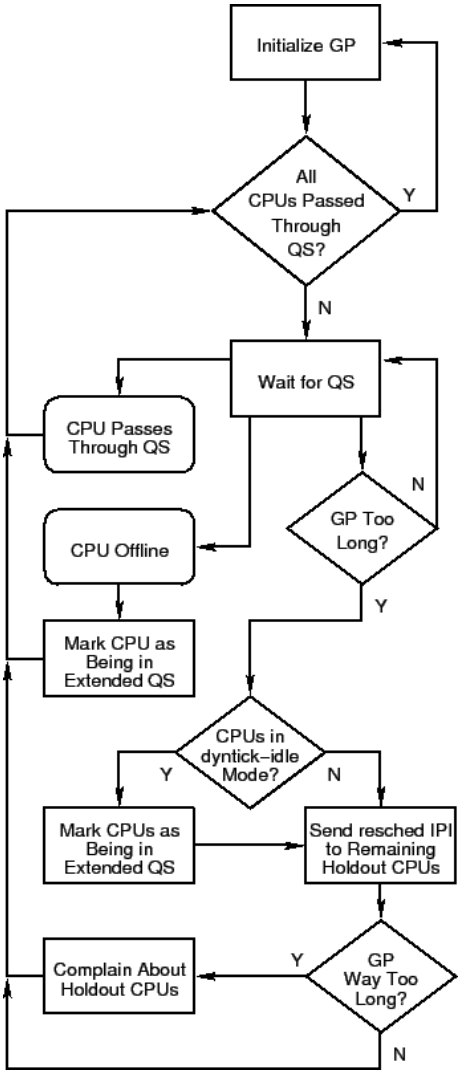


图 D.18: Generic RCU State Machine

从十分高层的角度来看，Linux-内核 RCU 实现可以被认为是一个高级状态机，如图 D.18。在一个很繁忙的系统上，通常的路径是最上面的两个循环。在每一个 grace period (GP)开始时进行初始化，等待静止状态 (QS)，在一个特定的 grace period 中，当每一个 CPU 都经历过静止状态时，就什么都不做。在这样一个系统中，每一次进程切换都产生一个静止状态，或者，在 CPU 进入 idle 状态或者执行用户态代码时，也产生一个静止状态。CPU-热插拔事件将使状态机进入 ``CPU Offline" 流程，而``holdout" CPUs 的出现，使得不能快速经历一次静止状态，这将使状态机进入 ``Send resched IPIs to Holdout CPUs" 流程。为了避免不必要的唤醒处于 dyntick-idle 状态的 CPU，RCU 实现将标记这些 CPUs 处于扩展的静止状态。最后，如果 CONFIG_RCU_CPU_STALL_DETECTOR 打开了，过迟的到达静止状态将使状态机进入``Complain About Holdout CPUs" 流程.

问题 D.10: 这个状态图不标出 dyntick-idle CPUs 将被 reschedule IPIs 唤醒? 为什么不唤醒它们?

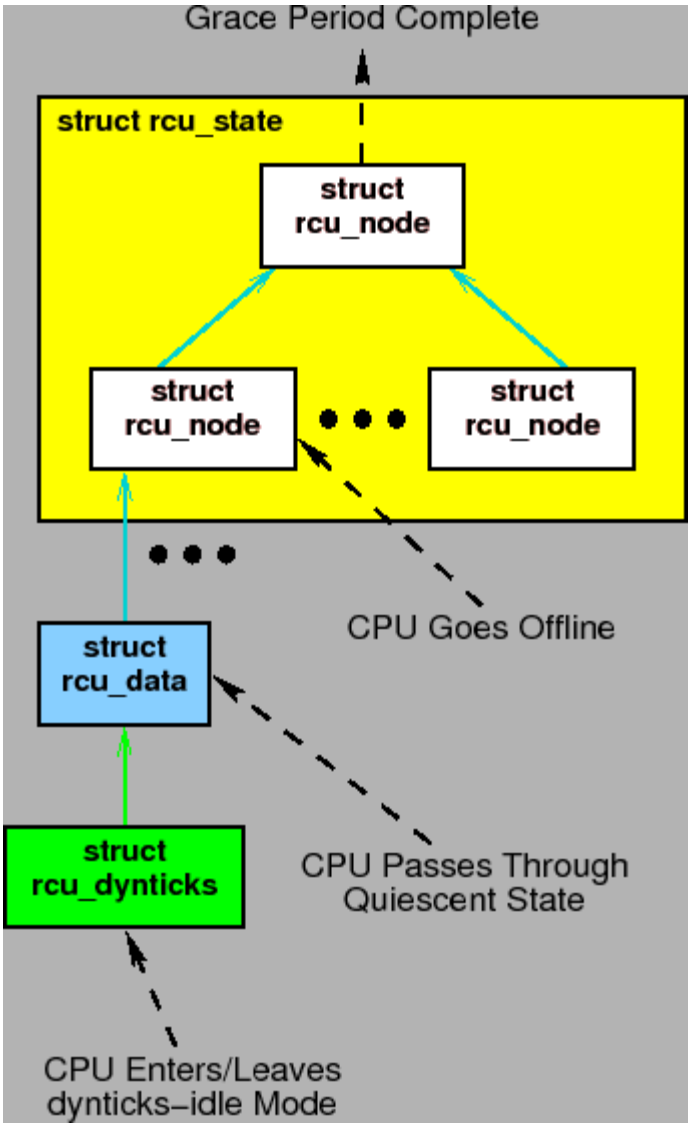


图 D.19: RCU State Machine and Hierarchical RCU Data Structures

上面的状态图中，事件会与不同的数据结构交互，如图 D.19。但是，状态图不会被任何 RCU 实现直接翻译为 C 代码。相反的，这些实现在内核中被编码为事件驱动的系统。随后的章节描述一些用例，以及 RCU 算法使用相关数据结构实现这些状态机的方法。

D.2.7 用例

本节给出一些 RCU 实现中，一些用例的概要，列出用到的数据结构和调用

的函数。这些用例如下：

- ✓ 开始一个新的 Grace Period (D.2.7.1 节)
- ✓ 经历一个静止状态 (D.2.7.2 节)
- ✓ 向 RCU 通告一个静止状态 (D.2.7.3 节)
- ✓ 进入、退出 Dynticks Idle 模式 (D.2.7.4 节)
- ✓ 从 Dynticks Idle 模式进入中断 (D.2.7.5 节)
- ✓ 从 Dynticks Idle 模式进入 NMI (D.2.7.6 节)
- ✓ 标记一个 CPU 处于 Dynticks Idle 模式 (D.2.7.7 节)
- ✓ CPU 离线 (D.2.7.8 节)
- ✓ CPU 上线 (D.2.7.9 节)
- ✓ 检测一个太长的 Grace Period (D.2.7.10 节)

后面的章节描述每一个用例。

D.2.7.1 开始一个新的 Grace Period

rcu_start_gp() 函数开始一个新的 grace period。当一个 CPU 有一个等待 grace period 的回调，但是没有 grace period 在运行时，就调用此函数。

rcu_start_gp() 函数更新 rcu_state 和 rcu_data 结构中的状态，以标识开始一个新的 grace period, 获取 ->onoff lock (并关中断) 以拒绝任何并发的 CPU 热插拔操作，在所有的 rcu_node 结构中设置位，以标识所有 CPUs (包括当前 CPU) 必须经历一次静止状态，最后释放->onoff 锁。

设置位操作分两个阶段进行。首先，在没有持有任何锁的情况下，非叶子节点 rcu_node 的位被设置，然后，在持有->lock 的情况下，每一个叶子节点的 rcu_node 结构的位被设置。

问题 D.11：如果在位已经设置完成的情况下，一个 CPU 报告它经历了一个静止状态，会发生什么？

问题 D.12：在位设置完成前，如果所有 CPU 都报告它们经历过一个静止状态会发生什么？

D.2.7.2 经历一次静止状态

rcu 和 rcu_bh 有各自的静止状态集合。RCU 的静止状态是进程切换，IDLE (不管是 dynticks 还是 IDLE loop)，以及执行用户态程序。但是 RCU-bh 的静止状态是在关中断状态下，退出软中断。需要注意的是，rcu 的静止状态也是 rcu_bh

的静止状态。Rcu 的静止状态通过调用 rcu_qsctr_inc()来记录。而 rcu_bh 的静止状态通过调用 rcu_bh_qsctr_inc()来记录。这两个函数将它们的状态记录到当前 CPU 的 rcu_data 结构中。

这些函数在调度器、__do_softirq()和 rcu_check_callbacks()中被调用。后面这个函数在 scheduling-clock 中断中调用，并分析状态以确定中断是否发生在一个静止状态中，以确定是调用 rcu_qsctr_inc() 或者 rcu_bh_qsctr_inc()。它也触发 RCU_SOFTIRQ，并导致 CPU 在随后的软中断上下文中调用 rcu_process_callbacks()。

D.2.7.3 向 RCU 宣告一次静止状态

前述的 rcu_process_callbacks() 函数要完成几个事情：

- ✓ 确定何时结束一个太长的 grace period (通过 force_quiescent_state()).
- ✓ 当其他 CPU 检测到 grace period 结束时，采用适当的动作。(通过 rcu_process_gp_end())。``适当的动作``包括加快本 CPU 的回调，以及记录新的 grace period。同一个函数也更新状态以响应其他 CPU。
- ✓ 向 RCU 核机制报告当前 CPU 的静止状态。(通过 rcu_check_quiescent_state(), 它会调用 cpu_quiet())。当然也会标记当前的 grace period 结束。
- ✓ 如果没有处理 grace period，并且这个 CPU 有 RCU 回调等待 grace period，则开始一个新的 grace period。(通过 cpu_needs_another_gp() 和 rcu_start_gp()).
- ✓ 当 grace period 结束时，调用这个 CPU 的回调 (通过 rcu_do_batch()).

这些接口都经过精心实现，以避免 BUG。

D.2.7.4 进入和退出 Dynticks Idle 模式

调度器调用 rcu_enter_nohz()进入 dynticks-idle 模式，并调用 rcu_exit_nohz()离开此模式。rcu_enter_nohz() 函数递增 per-CPU dynticks_nesting 变量，也递增 per-CPU dynticks 计数器，然后，后者必然拥有一个偶数值。rcu_exit_nohz() 函数递减 per-CPU dynticks_nesting 变量，并且再一次递增 per-CPU dynticks 计数器，后者将拥有一个奇数值。

dynticks 计数器可以被其他 CPUs 采样。如果其值是偶数，那么 CPU 处于扩展静止状态。类似的，如果计数器在一个特定的 grace period 内发生了改变，那么 CPU 必须在 grace period 期间的某个时间点上处于扩展静止状态。但是，

还需要采样另外一个 dynticks_nmi per-CPU 变量，随后我们将看到这个变量。

D.2.7.5 从 Dynticks Idle 模式进入中断

从 dynticks idle 模式进入中断由 rcu_irq_enter() 和 rcu_irq_exit()处理。rcu_irq_enter() 函数递增 per-CPU dynticks_nesting 变量。并且。如果先前的值是 0，也递增 dynticks per-CPU 变量 (它将拥有一个奇数值)。

rcu_irq_exit() 函数递减 per-CPU dynticks_nesting 变量。并且，如果新值是 0，也递减 dynticks per-CPU 变量 (它将拥有一个偶数值)。

注意：进入中断会处理退出 dynticks idle 模式。进入、退出之间不相符可能导致一些混淆。

D.2.7.6 从 Dynticks Idle 模式进入 NMI

从 dynticks idle 模式进入 NMI 由 rcu_nmi_enter() 和 rcu_nmi_exit()处理. 这些函数同时递增 dynticks_nmi 计数器，但仅仅是在前述 dynticks 计数是偶数时才进行递增。换句话说，如果 NMI 发生时，处于 non-dynticks-idle 模式或者处于中断状态，那么 NMI 将不操作 dynticks_nmi 计数器。

这两个函数之间唯一的差异在于错误检查， rcu_nmi_enter() 必须使 dynticks_nmi 计数器为奇数值，rcu_nmi_exit() 必须使这个计数器为偶数值。

D.2.7.7 标记 CPU 处于 Dynticks Idle 模式

force_quiescent_state() 函数实现一个三阶段的状态机。 第一个阶段 (RCU_INITIALIZING) 等待 rcu_start_gp()完成 grace-period 初始化。这个状态不是从 force_quiescent_state()退出，而是从 rcu_start_gp()退出。

在第二阶段(RCU_SAVE_DYNTICK)，dyntick_save_progress_counter() 函数扫描还没有报告静止状态的 CPUs，记录它们的 per-CPU dynticks 和 dynticks_nmi 计数器。如果这些计数器都是偶数值，那么相应的 CPU 处于 dynticks-idle 状态，因此标记它们为扩展静止状态 (通过 cpu_quiet_msk()报告)。

在第三阶段 (RCU_FORCE_QS), rcu_implicit_dynticks_qs() 函数再一次扫描仍然没有报告静止状态的 CPUs (既没有明确标示，也没有在 RCU_SAVE_DYNTICK 阶段隐含的标示)，再一次检查 per-CPU dynticks 和 dynticks_nmi 计数器。如果每一个值都变化，或者目前为偶数，那么相应的相应的 CPU 已经经过一次静止状态或者目前处于 dynticks idle 模式，也就是前述扩展

静止状态。

如果 rcu_implicit_dynticks_qs()发现特定 CPU 既没有处于 dynticks idle 模式，也没有报告一个静止状态，它调用 rcu_implicit_offline_qs()，这个函数检查 CPU 是否处于离线状态，如果是，那么也报告一个扩展静止状态。如果 CPU 在线，那么 rcu_implicit_offline_qs() 发送一个 reschedule IPI，尝试提醒该 CPU 应当报告一个静止状态。

force_quiescent_state() 既不直接调用 dyntick_save_progress_counter() ，也不直接调用 rcu_implicit_dynticks_qs()，而是将它们传递给 rcu_process_dyntick() 函数。

问题 D.13：如果一个 CPU 从 dyntick-idle 退出，然后经过一次静止状态，正好在此时，另一个 CPU 注意到它处于 dyntick-idle 模式，这将会发生什么?它们不会同时尝试报告一个静止状态，从而导致冲突吗?

问题 D.14：如果所有 CPU 都以 dyntick-idle 结束时，会怎样？不用防止当前 RCU grace period 永不结束?

问题 D.15：force_quiescent_state()是一个三阶段状态机，在扫描所有 CPU 时，不会产生三倍的调度延迟？

D.2.7.8 CPU 离线

CPU-离线事件导致 rcu_cpu_notify()调用 rcu_offline_cpu()，在 rcu 和 rcu_bh 上依次调用__rcu_offline_cpu()。这个函数清除离线 CPU 的位，这样，后面的 grace periods 将不再期望这个 CPU 宣告静止状态，随后调用 cpu_quiet()，以宣告离线扩展静止状态。这是在持有全局->onofflock 锁的情况下执行的，这是为了防止与 grace-period 初始化相冲突。

问题 D.16：其他持有 ->onofflock 锁的原因是防止多个并发的 online/offline 操作造成冲突，是这样吗?

D.2.7.9 CPU 上线

CPU-online 事件导致 rcu_cpu_notify()调用 rcu_online_cpu()，用于初始 CPU 的 dynticks 状态，然后调用 rcu_init_percpu_data()初始化 CPU 的 rcu_data 数据结构，也设置这个 CPU 的位(同样通过全局 ->onofflock 进行保护) ，这样后面的静止状态将等待这个 CPU 的静止状态。最后，rcu_online_cpu() 设置这个 CPU

的 RCU 软中断向量。

问题 D.17：对于所有这些获取全局 ->onofflock 的操作，在千个 CPU 的系统不会存在令人讨厌的锁竞争吗？

问题 D.18：为什么不合并检测 dyntick-idle CPUs 到 CPUs 离线处理代码中，以简化代码？

D.2.7.10 检测太长的 Grace Period

当配置了 CONFIG_RCU_CPU_STALL_DETECTOR 内核参数时，record_gp_stall_check_time() 函数记录时间，以及 3 秒以后的时间戳。如果当前 grace period 到时仍然没有结束，并且当前 CPU 是造成延迟的 cpu0，则调用 print_cpu_stall()，如果不是，则调用 print_other_cpu_stall()。

D.2.8 测试

RCU 是基本的同步代码，因此 RCU 的错误导致的后果是随机的，难于调试的内存错误。因此，高可靠的 RCU 是非常重要的。这些可靠性来自于小心的设计，但是最终还是需要依赖于高强度的压力测试 。

幸运的是，虽然有一些关于覆盖性方面的争论，但是仍然可以对软件进行一些压力测试。实际上，进行这些测试是被强烈建议的，因为不对你的软件进行折磨性测试的话，它就会反过来折磨你。

因此，我们使用 rcutorture 模块来对 RCU 进行折磨性的测试。

但是，它对通用情况下的 RCU 用法还不是很充分。也有必要对不常用的情况进行折磨性测试。例如，CPU 并发的上线或者离线，CPU 并发的进入及退出 dynticks idle 模式。我使用一个脚本 @@@ move to CodeSamples, ref @@@ ，并向模块 rcutorture 使用 test_no_idle_hz 模块参数对 dynticks idle 模式进行压力测试。有时我比较疑神疑鬼，因此有时并发的运行一个 kernbench 负载。在 128 路的机器上运行 10 个小时的压力测试，看起来是足够测试了几乎所有 BUGs 了。

实际上这还不算完。Alexey Dobriyan 和 Nick Piggin 早在 2008 年就证明过，以所有相关内核参数组合对 RCU 进行折磨测试是必要的。相关的内核参数可以使用另外一个脚本@@@ move to CodeSamples, ref @@@ 进行标识。

CONFIG_CLASSIC_RCU: 经典 RCU.

CONFIG_PREEMPT_RCU: 可抢占 (实时) RCU.

CONFIG_TREE_RCU: 用于大型 SMP 系统的经典 RCU.

CONFIG_RCU_FANOUT: 每一个 rcu_node 的 children 数量.

CONFIG_RCU_FANOUT_EXACT: rcu_node 树平衡.

CONFIG_HOTPLUG_CPU: 允许 CPUs 上线、离线.

CONFIG_NO_HZ: 打开 dyntick-idle 模式.

CONFIG_SMP: 打开 multi-CPU 选项.

CONFIG_RCU_CPU_STALL_DETECTOR: 当 CPUs 进入扩展静止状态时进行 RCU 检测

CONFIG_RCU_TRACE: 在 debugfs 中生成 RCU 跟踪文件

当忽略 CONFIG_DEBUG_LOCK_ALLOC 配置时，分级 RCU 不能打断 lockdep。有 10 个配置值，如果它们是独立的布尔值，则导致 1024 种组合。幸运的是，首先，其中三个是互斥的，这样可以将组合数量减少到 384 个，但是 CONFIG_RCU_FANOUT 可以取值 2-64，将组合数量增加到 12,096。这是一个不可实施的组合。

关键的一点是：如果 CONFIG_CLASSIC_RCU 或者 CONFIG_PREEMPT_RCU 有效时，预期仅仅 CONFIG_NO_HZ 和 CONFIG_PREEMPT 可能会改变其行为。

而且，并不是这些所有可能的 CONFIG_RCU_FANOUT 值都会产生有用的结果，实际上仅仅一部分情况需要分别测试：

单结点 ``tree".

两级平衡树.

三级平衡树

自动平衡树，当 CONFIG_RCU_FANOUT 指定一个不平衡树，但是没有 CONFIG_RCU_FANOUT_EXACT 时，进行自动平衡。

非平衡树.

更进一步说，CONFIG_HOTPLUG_CPU 仅仅在指定 CONFIG_SMP 时才有用，CONFIG_RCU_CPU_STALL_DETECTOR 是独立的，因此仅仅需要测试一次(虽然有时我太多疑了，因此决定在有 CONFIG_SMP 和没有 CONFIG_SMP 时，都测试它)。类似的，CONFIG_RCU_TRACE 也仅仅需要测试一次，但是象我一样多疑的人，会选择在有 CONFIG_NO_HZ 和没有 CONFIG_NO_HZ 时，都测试一下它。

这允许我们在 15 种测试情形下，得到一个覆盖率较好的 RCU 测试。所有这些测试情形都指定如下配置参数以运行 rcutorture，这样 CONFIG_HOTPLUG_CPU=n 会产生实际的效果：

```
CONFIG_RCU_TORTURE_TEST=m
CONFIG_MODULE_UNLOAD=y
CONFIG_SUSPEND=n
```

CONFIG_HIBERNATION=n

15 个测试情况如下：

强制单节点 ``tree" ，用于小型系统：

CONFIG_NR_CPUS=8
CONFIG_RCU_FANOUT=8
CONFIG_RCU_FANOUT_EXACT=n
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=y

强制两级节点用于大型系统：

CONFIG_NR_CPUS=8
CONFIG_RCU_FANOUT=4
CONFIG_RCU_FANOUT_EXACT=n
CONFIG_RCU_TRACE=n
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=y

强制三级节点，用于非常大型的系统:

CONFIG_NR_CPUS=8
CONFIG_RCU_FANOUT=2
CONFIG_RCU_FANOUT_EXACT=n
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=y

测试自动平衡:

CONFIG_NR_CPUS=8
CONFIG_RCU_FANOUT=6
CONFIG_RCU_FANOUT_EXACT=n
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=y

测试不平衡树:

CONFIG_NR_CPUS=8
CONFIG_RCU_FANOUT=6
CONFIG_RCU_FANOUT_EXACT=y
CONFIG_RCU_CPU_STALL_DETECTOR=y
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=y

禁止 CPU 延迟检测：

CONFIG_SMP=y
CONFIG_NO_HZ=y
CONFIG_RCU_CPU_STALL_DETECTOR=n
CONFIG_HOTPLUG_CPU=y
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=y

禁止 CPU 延迟检测及 dyntick idle 模式:

CONFIG_SMP=y
CONFIG_NO_HZ=n
CONFIG_RCU_CPU_STALL_DETECTOR=n
CONFIG_HOTPLUG_CPU=y
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=y

禁止 CPU 延迟检测及 cpu0 热插拔:

CONFIG_SMP=y
CONFIG_NO_HZ=y
CONFIG_RCU_CPU_STALL_DETECTOR=n
CONFIG_HOTPLUG_CPU=n
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=y

禁止 CPU 延迟检测, dyntick idle 模式, 及 CPU 热插拔:

CONFIG_SMP=y
CONFIG_NO_HZ=n
CONFIG_RCU_CPU_STALL_DETECTOR=n
CONFIG_HOTPLUG_CPU=n
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=y

禁止 SMP, CPU 延迟检测, dyntick idle 模式, 及 CPU 热插拔:

CONFIG_SMP=n
CONFIG_NO_HZ=n
CONFIG_RCU_CPU_STALL_DETECTOR=n
CONFIG_HOTPLUG_CPU=n
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n

CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=y

这个组合有一些编译警告.

Disable SMP and CPU hotplug:
CONFIG_SMP=n
CONFIG_NO_HZ=y
CONFIG_RCU_CPU_STALL_DETECTOR=y
CONFIG_HOTPLUG_CPU=n
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=y

有 dynticks idle 但是没有抢占的情况下，测试经典 RCU:

CONFIG_NO_HZ=y
CONFIG_PREEMPT=n
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=y
CONFIG_TREE_RCU=n

有抢占但是没有 dynticks idle 时，测试经典 RCU:

CONFIG_NO_HZ=n
CONFIG_PREEMPT=y
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=n
CONFIG_CLASSIC_RCU=y
CONFIG_TREE_RCU=n

在 dynticks idle 情况下，测试可抢占 RCU:

CONFIG_NO_HZ=y
CONFIG_PREEMPT=y
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=y
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=n

在没有 dynticks idle 时，测试可抢占 RCU:

CONFIG_NO_HZ=n
CONFIG_PREEMPT=y
CONFIG_RCU_TRACE=y
CONFIG_PREEMPT_RCU=y
CONFIG_CLASSIC_RCU=n
CONFIG_TREE_RCU=n

对于每一次大的 RCU 核心代码的变化，都应当以上面的组合运行 rcutorture，并且在 CONFIG_HOTPLUG_CPU 时，并发的进行 CPU 热插拔。对小的变化，

在每一种情况下运行 kernbench 就行了。当然，如果变化仅仅限于配置参数的部分子集，就可以减少测试情况的数量。

压力测试软件：Geneva Convention 仍然没有禁止它，我强烈推荐它!

D.2.9 结论

这个分级 RCU 实现减少了锁竞争，避免了不必要的唤醒 dyntick-idle 睡眠状态的 CPUs，因此有助于调试 Linux CPU 热插拔代码。这个实现被设计用于处理数千个 CPUs 的单个系统，并且在 64 位系统上，CPUs 数量限制是 250,000，在今后一段时间内，这个限制是足够的。

这个 RCU 实现当然也有一些局限：

force_quiescent_state()可能在关中断下扫描整个 CPUs 集。这在实时 RCU 实现中，是一个重大缺陷。因此，如果需要在可抢占 RCU 中加入分级，则需要其他方法。在 4096 个 CPU 的系统中，它可能会产生一些问题，但是需要在实际的系统中进行测试以证明真的有问题。

在繁忙的系统中，不能期望 force_quiescent_state() 扫描发生，CPUs 将在开始一个静止状态后，三个 jiffies 内经历一次静止状态。在半繁忙的系统中，仅仅处于 dynticks-idle 模式的 CPUs 需要扫描。其他情况下，例如，在一个 dynticks-idle CPU 在扫描过程中，处理一个中断时，后继的扫描是需要的。但是，这样的扫描是分别执行的，因此相应的调度延迟仅仅影响该扫描过程所在的 CPU 负载。

如果扫描被证明确实有问题，一个好的方法是进行递增扫描。这将稍微增加一点代码复杂性，也增加一点结束 grace period 的时间，但是这也确实算是一个好的方案。

rcu_node 分级在编译时创建，因此其长度是最大的 CPUs 数量 NR_CPUS。但是，即使在 4,096 CPUs 的系统，在 64 位系统上，rcu_node 分级也仅仅消耗 65 个缓存行。(即使在 32 们系统上包含 4,096 CPUs 也是这样!)。当然，在一个 16 CPU 的系统中，配置 NR_CPUS=4096 将使用一个二级树，实际上单节点树也会运行得很好。虽然这个配置会增加锁的负载，但是实际上不会影响经常执行的读端代码，因此事实上不会有太大的问题。

这个补丁会稍微增加内核代码及数据尺寸。在 NR_CPUS=4 的系统中，从经典 RCU 的 1,757 字节内核代码、456 字节数据，共 2213 字节的内核尺寸，增加到 4,006 字节的内核代码、624 字节的内核数据，共计 4,630 字节尺寸。即使对大多数嵌入式系统来说，这也不是一个问题。这些系统通常有上百 M 主内存。但是对特别小的系统来说，这可能就是一个问题了。

即使有这些问题，相对于经典 RCU 来说，在数百个 CPUs 的系统中，这个

分级 RCU 实现仍然是一个巨大的进步。最后，经典 RCU 设计用于 16-32 个 CPUs 的系统。

在某些地方，在可抢占 RCU 实现中使用分级是有必要的。有一点点挑战性，但是还是可以实现。

D.3 分级 RCU 代码走查

本节选择一些 linux 内核分级 RCU 代码进行走查。同样，本节是为那些希望非常深层次的理解 RCU 的骨灰级黑客准备的。这些黑客应当首先阅读 D.2 节. 骨灰级受虐狂也可能有兴趣看看这一节。当然，真正的骨灰级的受虐狂将在阅读 D.2 节前阅读本节。

D.3.1 节描述数据结构和内核参数， D.3.2 节包含外部函数接口， D.3.3 节提供了初始化过程，D.3.4 节解释 CPU 热插拔接口，D.3.5 节包含一些杂项函数，D.3.6 节描述 grace-period 检测机制，D.3.7 节描述 dynticks-idle 接口，D.3.8 接口包含处理离线及 dynticks-idle CPUs 的函数。 D.3.9 节描述报告 CPUs 延迟的函数。最后，D.3.10 节报告可能的设计缺陷和问题。

D.3.1 数据结构及内核参数

全面的理解分级 RCU 数据结构，对于理解其算法是十分重要的。D.3.1 节描述用来跟踪每一个 CPU 的 dyntick-idle 状态的数据结构，D.3.2 节描述 per-node 数据结构的每一个字段，D.3.3 节描述 per-CPU rcu_data 结构，D.3.4 节描述全局 rcu_state 数据结构，D.3.5 节描述控制分级 RCU 的内核参数。

图 D.17 以及图 D.26 通过数据流的方式对数据结构进行描述，它是非常有用的。

D.3.1.1 跟踪 Dyntick 状态

per-CPU rcu_dynticks 数据结构使用下面的字段跟踪 dynticks 状态：

dynticks_nesting：这个整型值是原因计数，表示相应的 CPU 应当监控的 RCU 读临界区的数量。如果 CPU 处于 dynticks-idle 模式，那么这个值是中断嵌套级别，否则它比 irq 中断嵌套级别大。

Dynticks：如果相应的 CPU 处于 dynticks-idle 模式，并且没有中断处理函数正在该 CPU 上运行，则这个计数值是偶数，否则是奇数。换句话说，如果计数值是奇数，那么相应的 CPU 可能处于 RCU 读临界区中。

dynticks_nmi：如果相应的 CPU 处于 NMI 处理函数中，则这个整形计数器的值是奇数，但是仅仅是在 CPU 处于 dyntick-idle 模式并且没有中断处理程序在运行时，NMI 到达时才是这样。否则，计数器是偶数。

rcu 和 rcu_bh 共享这个值。

D.3.1.2 分级实现中的节点

正如较早前提示的那样，rcu_node 是被放到 rcu_state 结构内的，如图 D.13 所示。每一个 rcu_node 有下面的字段：

Lock：这个 spinlock 保护这个结构中的非常量字段。这个锁在软中断上下文中获取，因此必须禁止中断。

问题 D.19：为什么在获取 rcu_data 结构的锁时，不简单的禁止下半部分（软中断）？这不是更快吗？

根 rcu_node 的 lock 字段有另一个作用：串行化 CPU-延迟检测，这样仅仅一个 CPU 报告 CPU 延迟事件。这在上千个 CPUs 的系统中是很重要的!

串行化启动一个新的 grace period，这样多个 CPU 不会同时开始 grace periods。

在开始 grace periods 时，防止开始一个新的 grace periods。

将状态机中强制产生静止状态的行为串行化，这样可以将重新调度 IPIs 降低到适当的数目。

Qsmask：这个位图掩码跟踪哪些 CPUs (rcu_node 叶子节点)或者 CPUs 组(rcu_node 非叶子节点)仍然需要经历一个静止状态，以结束当前 grace period。

Qsmaskinit：这个位图掩码跟踪哪些 CPUs (rcu_node 叶子节点)或者 CPUs 组(rcu_node 非叶子节点)仍然需要经历一个静止状态，以结束后续的 grace periods。CPU 热插拔代码维护 qsmaskinit 字段，在开始每一个 grace period 时，将它们复制到相应的 qsmask 字段。这个复制操作，是 grace period 初始化过程需要与 CPU 热插拔代码互斥的原因之一。

Grpmask：这个位图掩码中的位，与这个 rcu_node 结构在父 rcu_node 结构中 qsmask 和 qsmaskinit 中的位置是一致的。使用这个字段简化了静止状态处理，这是 Manfred Spraul 建议的。

问题 D.20：rcu_node 叶子节点的 qsmask 和 qsmaskinit 字段是如何的?

Grplo：这个字段表示这个 rcu_node 包含的编号最小的 CPU。

Grphi：这个字段表示这个 rcu_node 包含的编号最大的 CPU。

Grpnum：这个字段包含与这个 rcu_node 相应的父 rcu_node 结构的 qsmask 和 qsmaskinit 字段的位编号。换句话说，给定一个 rcu_node 结构指针 rnp，总有

1UL «rnp->grpnum == rnp->grpmask. grpnum 字段用于调试目的。

Level：对根 rcu_node 结构来说，这个字段是 0。根的子结点是 1，依此类推。

Parent：这个字段指向父 rcu_node 结构，对根结点来说，其值为 NULL。

D.3.1.3 Per-CPU 数据

rcu_data 数据结构包含 RCU 的每 CPU 状态。它包含管理 grace periods 和静止状态(completed, gpnum, passed_quiesc_completed, passed_quiesc, qs_pending, beenonline, mynode, 和 grpmask)的控制变量。rcu_data 数据结构也包含关于 RCU 回调的控制变量 (nxtlist, nxttail, qlen, 和 blimit)。打开 dynticks 的内核在 rcu_data 数据结构中也有相关的控制变量(dynticks, dynticks_snap, 和 dynticks_nmi_snap)。rcu_data 数据结构包含用于跟踪的事件计数器 (dynticks_fqs given dynticks, offline_fqs, and resched_ipi)。最后，还有一对字段，对调用 rcu_pending()进行计数，以确定何时强制进行静止状态(n_rcu_pending 和 n_rcu_pending_force_qs)，以及一个 cpu 字段标识哪一个 CPU 与该 rcu_data 结构对应。

每一个字段描述如下：

Completed：这个字段本 CPU 已经完成的 grace period 编号。

Gpnum：这个字段包含本 CPU 启动的 grace period 编号。

passed_quiesc_completed：本字段包含本 CPU 最近经过静止状态时，已经完成的 grace period 编号。最近完成的编号，其观察点在该 CPU 经历静止状态时：如果 CPU 没有观察到 grace period 42 完成，它将记录其值为 41。这是对的，因为 grace period 能够完成的唯一方法就是这个 CPU 已经经历过一次静止状态。这个字段被初始化为一个虚构的次数，以避免在 boot 和 CPU 上线时产生竞争条件。

passed_quiesc：这个字段表示自从存储在 passed_quiesc_completed 的 grace period 完成以来，本 CPU 是否经历过一次静止状态。

qs_pending：这个字段表示本 CPU 已经注意到 RCU 核心机制正在等待它经历一次静止状态。当 CPU 检测到一个新的 grace period，或者一个 CPU 上线时，将这个字段设置为 1。

问题 D.21：但是为什么当一个 CPU 上线时，将 qs_pending 设置为 1?

问题 D.22：为什么要记录最后一次 grace period 编号到 passed_quiesc_completed?

Beenonline：这个字段初始化为 0，当相应的 CPU 上线时被设置为 1，这用

于避免处理从没有上线的 CPU，当 NR_CPUS 大大超过实际的 CPU 数量时，这是有用。

问题 D.23：什么情况下，NR_CPUS 会大大超过实际的 CPUs 数量？

Mynode：这个字段指向处理相应 CPU 的 rcu_node 叶子节点。

Grpmask：这个掩码字段只有一个位，表示 mynode->qsmask 中哪一位与相应的 CPU 对应。

Nxtlist：这个字段指向本 CPU 最近一次的 RCU 回调(rcu_head 结构)，如果本 CPU 最近没有这样的回调，就设置为 NULL。其他的回调可能通过它们的 next 指针链接在一起。

Nxttail：这是一个指向 nxtlist 回调链表尾部的指针数组。如果 nxtlist 是空的，那么所有 nxttail 指针直接指向 nxtlist 字段。每一个 nxttail 数组节点有如下意义：

RCU_DONE_TAIL=0：这个元素是 CPU 在它经历 grace period 时最后调用的回调函数的->next 字段。如果没有这样的回调函数，则是 nxtlist 字段。

RCU_WAIT_TAIL=1：等待当前 grace period 结束时，最后一个回调函数的->next 指针，如果没有这样的回调函数，就等于 RCU_DONE_TAIL 元素。

RCU_NEXT_READY_TAIL=2：等待下一个 grace period 时的回调函数的 next 字段，如果没有这样的回调函数，则等于 RCU_WAIT_TAIL 元素。

RCU_NEXT_TAIL=3：链表中的最后一个回调函数的 next 指针，如果链表为空，就是 nxtlist 字段。

问题 D.24：为什么不简单的使用多个链表？

Qlen：在 nxtlist 链表中排队的回调函数数量。

Blimit：在某一个时刻可以调用的回调函数最大值。在高负载情况下，这个限制增强了系统响应性能。

Dynticks：与 cpu 对应的 rcu_dynticks 结构，在 D3.1.1 节中描述。

dynticks_snap：dynticks->dynticks 曾经经历过的值，在 CPU 在中断处理函数中检查 CPU 何时经历过一次 dynticks idle 状态。

dynticks_nmi_snap：dynticks->dynticks_nmi 曾经经过过的值，在 CPU NMI 处理函数中检查 CPU 何时经历过一次 dynticks idle 状态。

dynticks_fqs：其他 CPU 由于 dynticks idle 而标记一次静止状态的次数。

offline_fqs：其他 CPU 由于离线而标记一次静止状态的次数。

问题 D.25: So some poor CPU has to note quiescent states on behalf of each and every offline CPU? Yecch! Won't that result in excessive overheads in the not-uncommon case of a system with a small number of CPUs but a large value for

NR_CPUS? End Quick Quiz

resched_ipi：向相应 CPU 发送的重新调度 IPI 次数。向没有及时报告自己经历过静止状态的 CPU 发送这样的 IPI，但既不向离线 CPU，也不向处于 dynticks idle 状态的 CPU 发送这样的 IPI。

n_rcu_pending：调用 rcu_pending()的次数，在一个非 dynticks-idle 的 CPU 上，每一个 jiffy 将调用一次这个函数。

n_rcu_pending_force_qs：n_rcu_pending 上限。如果 n_rcu_pending 达到此值, 表示当前 grace period 延迟太久，则调用 force_quiescent_state()。

D.3.1.4 RCU 全局状态

rcu_state 结构包含每一个 RCU 实例（rcu 和 rcu_bh）的全局状态。包括与分级 rcu_node 相关的字段，包括结点数组本身，levelcnt 数组包含每一级结点计数，levelspread 数组包含每一个结点的子结点数量。rda 数组是每一个 CPU 的 rcu_data 结构指针。rcu_state 也包含一定数量的，与当前 grace period 相关的字段，以及与其他机制交互的字段 (signaled, gpnum, completed, onofflock, fqslock, jiffies_force_qs, n_force_qs, n_force_qs_lh, n_force_qs_ngp, gp_start, jiffies_stall, and dynticks_completed)。

每一个字段描述如下：

Node：这个字段是 rcu_node 结构数组，根节点位于->node[0]。其长度由 NUM_RCU_NODES C-预处理宏指定，根据 NR_CPUS 和 CONFIG_RCU_FANOUT 计算确定。CONFIG_RCU_FANOUT 在 D.3.1.5 节描述。注意，从元素 0 开始遍历->node 数组可以达到宽度优先搜索 rcu_node 分级树的效果。

Level：指向 node 数组的指针数组。分级树的根节点由->level[0]引用，第二级节点的第一个节点（如果有的话）由->level[1]引用，依此类推。第一个叶子节点由 ->level[NUM_RCU_LVLS-1]引用，level 数组的长度由 NUM_RCU_LVLS 指定，其计算方法在 D.3.1.5 节描述。->level 字段通常与->node 字段一起使用，用于分级扫描 rcu_node，例如，扫描所有叶子节点. ->level 由启动时的 rcu_init_one() 函数填充。

Levelcnt：这是一个数组，包含每一层 rcu_node 结构的数量，包括引用 rcu_node 叶子结构的 rcu_data 数据结构的数量，因此这个数组的元素比->level 数组多。注意 ->levelcnt[0]总是包含值 1，表示分级结构的根 rcu_node 只有一个. 这个数组的值被初始化为 NUM_RCU_LVL_0，NUM_RCU_LVL_1，NUM_RCU_LVL_2 和 NUM_RCU_LVL_3，这些预编译的宏在 D.3.1.5 节描述。

->levelcnt 字段用于初始化分级结构的其他部分，并用于调试目的。

Levelspread：这个字段的每一个元素包含 rcu_node 分级结构每一层期望的的子节点数量。这个数组的值由两个 rcu_init_levelspread()函数在运行时计算，具体选择哪一个函数由 CONFIG_RCU_FANOUT_EXACT 内核参数确定。

Rda：该字段的每一个元素包含相应 CPU 的 rcu_data 指针。这个数组在启动时由 RCU_DATA_PTR_INIT()宏初始化。

Signaled：这个字段用于维护 force_quiescent_state() 函数所使用的状态。该函数在 D.3.8 节描述. 这个字段可以有如下值：

RCU_GP_INIT：这个值表示当前 grace period 仍然在初始化过程中。因此 force_quiescent_state() 不应采取任何动作。当然，在 grace-period 真正开始前，grace-period 初始化最多可能需要三个 jiffies，如果你有大量的 CPUs，这个竞争可能会真实的发生。一旦完成 grace-period 初始化，这个值要么设置成 RCU_SAVE_DYNTICK (if CONFIG_NO_HZ) 要么设置成 RCU_FORCE_QS。

RCU_SAVE_DYNTICK：这个值表示 force_quiescent_state() 应当检查所有还没有报告静止状态的 CPU 的 dynticks 状态。报告静止状态表示 CPUs 已经处于 dyntick-idle 模式。

RCU_FORCE_QS：这个值表示 force_quiescent_state() 应当与在线、离线状态一起，重新检查还没有报告静止状态的 CPU 的 dynticks 状态。重新检测 dynticks 状态可以处理这样一种情况： 一个特定 CPU 可能处于 dynticks-idle 状态，但是在检测时，它同时处于中断和 NMI 处理中。

这个字段由根 rcu_node 锁保护。

问题 D.26：由什么来保护前面的字段?

Gpnum：当前 grace period 的值，如果当前没有 grace period，则是上一个 grace period 的值。由根 rcu_node 结构的锁保护。但是频繁的在没有这个锁保护的情况下访问（但是不修改）。

Completed：上一次 grace period 的值。同样的，如果当前没有 grace period，则它的值与->gpnum 是相等的。如果当前有 grace period 在处理，则比 ->gpnum 少 1。在某些 LINUX 版本的经典 RCU 中，这一对字段可以换成一个布尔变量。这个字段由根 rcu_node 结构的锁进行保护。但是频繁的在没有这个锁保护的情况下访问（但是不修改）。

Onofflock：防止在 grace period 初始化时，并发的处理上线、离线。但是有一个例外：如果 rcu_node 分级结构仅仅由一个单一个结构组成，那么单个结构的锁来代替这个任务。

Fqslock：这个字段用于在 force_quiescent_state()中，防止多个任务强制静止状态。

jiffies_force_qs：这是一个 以 jiffies 计算的时间，当需要调用 force_quiescent_state()以强制 CPUs 进入静止状态，或者报告一个静止状态时使用。这个字段由根 rcu_node 结构的锁保护。但是频繁的在没有这个锁保护的情况下访问（但是不修改）。

n_force_qs：调用 force_quiescent_state() 的次数。这个字段用于跟踪和调试，以 ->fqslock 锁进行保护。

n_force_qs_lh：由于->fqslock 被其他 CPU 获得而导致 force_quiescent_state()过早返回的次数的近似值。这个字段用于跟踪和调试，由于它是近似值，因此没有用任何锁进行保护。

n_force_qs_ngp：force_quiescent_state() 成功获得->fqslock 锁，但是随后发现没有 grace period 正在处理的次数。用于调试和跟踪，由->fqslock 进行保护。

gp_start：记录最近的 grace period 开始时间，以 jiffies 计数。这用于检测延迟的 CPUs，但是仅仅在 CONFIG_RCU_CPU_STALL_DETECTOR 内核参数选中时才有效。这个字段由根 rcu_node 的->lock 锁进行保护，但是有时不使用锁访问它。

jiffies_stall：这个时间值以 jiffies 计算，表示当前 grace period 什么时候会变得太长，此时将开始检查 CPU 延迟。与->gp_start 一样，仅仅在配置了 CONFIG_RCU_CPU_STALL_DETECTOR 内核参数时，这个字段才存在在。这个字段由根 rcu_node 保护，但是有时不使用这个锁而直接访问（不修改）。

dynticks_completed：当 force_quiescent_state()对 dyntick 进行快照时，这个字段记录->completed 的值。这个字段用于防止自前一个 grace period 到当前 grace period 的 dyntick-idle 静止状态。同样的，这个字段仅仅在配置了 CONFIG_NO_HZ 内核参数时才存在。这个字段由根 rcu_node 的锁进行保护，但有时也在没有锁保护的情况下进行访问（但不修改）。

D.3.1.5 内核参数

以下内核参数将影响 RCU：

NR_CPUS，系统中最大的 CPUs 数量。

CONFIG_RCU_FANOUT，在 rcu_node 分级体系中，期望的每一个节点的子节点数量。

CONFIG_RCU_FANOUT_EXACT，一个布尔值，防止 rcu_node 分组体系进行平衡操作。

CONFIG_HOTPLUG_CPU，允许 CPUs 上线、离线。

CONFIG_NO_HZ，表示支持 dynticks-idle 模式。

CONFIG_SMP，表示是否多核 CPUs。

CONFIG_RCU_CPU_STALL_DETECTOR，表示 RCU 将在 grace periods 太长时检查 CPUs 延迟。

CONFIG_RCU_TRACE，表示 RCU 将在 debugfs 中提供跟踪信息。

```
1 #define MAX_RCU_LVLS 3
2 #define RCU_FANOUT (CONFIG_RCU_FANOUT)
3 #define RCU_FANOUT_SQ (RCU_FANOUT * RCU_FANOUT)
4 #define RCU_FANOUT_CUBE (RCU_FANOUT_SQ * RCU_FANOUT)
5
6 #if NR_CPUS <= RCU_FANOUT
7 # define NUM_RCU_LVLS 1
8 # define NUM_RCU_LVL_0 1
9 # define NUM_RCU_LVL_1 (NR_CPUS)
10 # define NUM_RCU_LVL_2 0
11 # define NUM_RCU_LVL_3 0
12 #elif NR_CPUS <= RCU_FANOUT_SQ
13 # define NUM_RCU_LVLS 2
14 # define NUM_RCU_LVL_0 1
15 # define NUM_RCU_LVL_1 (((NR_CPUS) + RCU_FANOUT - 1) / RCU_FANOUT)
16 # define NUM_RCU_LVL_2 (NR_CPUS)
17 # define NUM_RCU_LVL_3 0
18 #elif NR_CPUS <= RCU_FANOUT_CUBE
19 # define NUM_RCU_LVLS 3
20 # define NUM_RCU_LVL_0 1
21 # define NUM_RCU_LVL_1 (((NR_CPUS) + RCU_FANOUT_SQ - 1) / RCU_FANOUT_SQ)
22 # define NUM_RCU_LVL_2 (((NR_CPUS) + (RCU_FANOUT) - 1) / (RCU_FANOUT))
23 # define NUM_RCU_LVL_3 NR_CPUS
24 #else
25 # error "CONFIG_RCU_FANOUT insufficient for NR_CPUS"
26 #endif /* #if (NR_CPUS) <= RCU_FANOUT */
27
28 #define RCU_SUM (NUM_RCU_LVL_0 + NUM_RCU_LVL_1 + NUM_RCU_LVL_2 + NUM_RCU_LVL_3)
29 #define NUM_RCU_NODES (RCU_SUM - NR_CPUS)
```

图 D.20: Determining Shape of RCU Hierarchy

CONFIG_RCU_FANOUT 和 NR_CPUS 参数用于在编译时确定 rcu_node 分级体系的形态。 如图 D.20 所示。第 1 行定义 rcu_node 分组体系的最大深度。注意增加最大深度需要修改其他地方，如，添加另外一个路径到第 6-26 行的#if

语句中。第 2-4 行计算 fanout, fanout 的平方, fanout 的立方。

然后将这些值与 NR_CPUS 进行比较，以确定 rcu_node 需要的深度，将其赋给 NUM_RCU_LVLS，用于 rcu_state 结构的数组长度。在根一层，总是只有一个节点，并且总有 NUM_CPUS 个 rcu_data 结构属于叶子节点。如果仅仅只比根层多一层，则叶子层的节点数是将 RCU_FANOUT 除以 NR_CPUS（向上对齐）。其他层的节点使用类似的方法进行计算，但是使用 RCU_FANOUT_SQ 代替 RCU_FANOUT。

随后第 28 行计算所有层的总和，结果是 rcu_node 结构的数量加上 rcu_data 的数量。最后，第 29 行从总和中减去 NR_CPUS (是 rcu_data 结构的数量) ，结果是 rcu_node 结构的数量，将其值保存在 NUM_RCU_NODES。这个值用于 rcu_state 结构的->nodes 数组的长度。

D.3.2 外部接口

RCU 的外部接口不仅仅包含标准的 RCU API，也包含向 RCU 自身需要的内部接口。这些接口是 rcu_read_lock()), rcu_read_unlock()), rcu_read_lock_bh()), rcu_read_unlock_bh()), call_rcu() (是对__call_rcu()), call_rcu_bh() (ditto), rcu_check_callbacks()的封装）, rcu_process_callbacks() (是对 __rcu_process_callbacks()的封装), rcu_pending() (是对 __rcu_pending()的封装), rcu_needs_cpu(), rcu_cpu_notify(), 和 __rcu_init()。注意 synchronize_rcu() 和 rcu_barrier() 通常用于所有 RCU 实现，并且以 call_rcu()的形式定义。类似的，rcu_barrier_bh() 通常用于所有 RCU 实现，并且以 call_rcu_bh()的形式定义。

外部接口分别在随后的章节中描述.

D.3.2.1 读端临界区

```
1 void __rcu_read_lock(void)
2 {
3 preempt_disable();
4 __acquire(RCU);
5 rcu_read_acquire();
6 }
7
8 void __rcu_read_unlock(void)
9 {
10 rcu_read_release();
11 __release(RCU);
12 preempt_enable();
```

```
13 }
14
15 void __rcu_read_lock_bh(void)
16 {
17 local_bh_disable();
18 __acquire(RCU_BH);
19 rcu_read_acquire();
20 }
21
22 void __rcu_read_unlock_bh(void)
23 {
24 rcu_read_release();
25 __release(RCU_BH);
26 local_bh_enable();
27 }
```

图 D.21: RCU Read-Side Critical Sections

图 D.21 显示了 RCU 读临界区的函数。第 1-6 行显示了__rcu_read_lock()，它开始一个``rcu"读临界区。第 3 行禁止抢占，第 4 行是一个弱的标记，表示开始一个 RCU 读临界区，第 5 行更新 lockdep 状态。第 8-13 行显示了__rcu_read_unlock()，它是__rcu_read_lock()的相对函数。第 15-20 显示__rcu_read_lock_bh() ，第 22-27 行显示 __rcu_read_unlock_bh()，它们与前两个函数类似，但是它们禁止、打开下半部分而不是抢占。

问题 D.27：我想 RCU 读端处理假设是非常快的！图 D.21 中显示的函数在读端很慢时显得无用了！这带来了什么问题?

D.3.2.2 call_rcu()

```
1 static void
2 __call_rcu(struct rcu_head *head,
3 void (*func)(struct rcu_head *rcu),
4 struct rcu_state *rsp)
5 {
6 unsigned long flags;
7 struct rcu_data *rdp;
8
9 head->func = func;
10 head->next = NULL;
11 smp_mb();
12 local_irq_save(flags);
13 rdp = rsp->rda[smp_processor_id()];
```

```
14 rcu_process_gp_end(rsp, rdp);
15 check_for_new_grace_period(rsp, rdp);
16 *rdp->nxttail[RCU_NEXT_TAIL] = head;
17 rdp->nxttail[RCU_NEXT_TAIL] = &head->next;
18 if (ACCESS_ONCE(rsp->completed) ==
19 ACCESS_ONCE(rsp->gpnum)) {
20 unsigned long nestflag;
21 struct rcu_node *rnp_root = rcu_get_root(rsp);
22
23 spin_lock_irqsave(&rnp_root->lock, nestflag);
24 rcu_start_gp(rsp, nestflag);
25 }
26 if (unlikely(++rdp->qlen > qhimark)) {
27 rdp->blimit = LONG_MAX;
28 force_quiescent_state(rsp, 0);
29 } else if ((long)(ACCESS_ONCE(rsp->jiffies_force_qs) -
30 jiffies) < 0 ||
31 (rdp->n_rcu_pending_force_qs -
32 rdp->n_rcu_pending) < 0)
33 force_quiescent_state(rsp, 1);
34 local_irq_restore(flags);
35 }
36
37 void call_rcu(struct rcu_head *head,
38 void (*func)(struct rcu_head *rcu))
39 {
40 __call_rcu(head, func, &rcu_state);
41 }
42
43 void call_rcu_bh(struct rcu_head *head,
44 void (*func)(struct rcu_head *rcu))
45 {
46 __call_rcu(head, func, &rcu_bh_state);
47 }
```

图 D.22: `call_rcu()` Code

图 D.22 显示了 __call_rcu()，call_rcu()和 call_rcu_bh()函数的代码。注意 call_rcu() 和 call_rcu_bh()是对 call_rcu()的简单封装，因此这里并不过多考虑它们。

将注意力转到__call_rcu()，第 9-10 行初始化指定的 rcu_head，第 11 行确保更新 RCU 保护数据结构先于调用__call_rcu() 回调函数前完成。第 12、34 行禁止并重新打开中断，以防止在一个中断处理函数中调用__call_rcu()而产生毁灭性的冲突。第 13 行得到当前 CPUs 的 rcu_data 引用，第 14 行调用

rcu_process_gp_end()，其目的是在当前 grace period 已经结束时，将回调函数提前。如果新的 grace period 已经开始，则第 15 行调用 check_for_new_grace_period() 记录状态。

问题 D.28：为什么在第 13 行不简单的使用__get_cpu_var() 来获得当前 CPU 的 rcu_data 结构的引用？

第 16 、17 行将新的回调函数加入队列。第 18、19 行检查是否有一个 grace period 正在处理中，如果没有，则第 23 行获得根 rcu_node 结构的锁，并在第 24 行调用 rcu_start_gp() 开始一个新的 grace period (也释放锁)。

第 26 行检查是否有太多的 RCU 回调在等待本 CPU，如果是这样，第 27 行递增->blimit，目的是提高正在处理的回调函数的速度。第 28 行调用 force_quiescent_state() ，其目的是试图使相应 CPU 经历一次静止状态。否则，第 29-32 行检查自 grace period 开始以来，是否已经经过太长时间，如果是这样，第 33 行调用非紧急的 force_quiescent_state()，也是为了使相应的 CPU 经历一次静止状态。

D.3.2.3 rcu_check_callbacks()

```
1 static int __rcu_pending(struct rcu_state *rsp,
2 struct rcu_data *rdp)
3 {
4 rdp->n_rcu_pending++;
5
6 check_cpu_stall(rsp, rdp);
7 if (rdp->qs_pending)
8 return 1;
9 if (cpu_has_callbacks_ready_to_invoke(rdp))
10 return 1;
11 if (cpu_needs_another_gp(rsp, rdp))
12 return 1;
13 if (ACCESS_ONCE(rsp->completed) != rdp->completed)
14 return 1;
15 if (ACCESS_ONCE(rsp->gpnum) != rdp->gpnum)
16 return 1;
17 if (ACCESS_ONCE(rsp->completed) !=
18 ACCESS_ONCE(rsp->gpnum) &&
19 ((long)(ACCESS_ONCE(rsp->jiffies_force_qs) -
20 jiffies) < 0 ||
21 (rdp->n_rcu_pending_force_qs -
```

```
22 rdp->n_rcu_pending) < 0))
23 return 1;
24 return 0;
25 }
26
27 int rcu_pending(int cpu)
28 {
29 return __rcu_pending(&rcu_state,
30 &per_cpu(rcu_data, cpu)) ||
31 __rcu_pending(&rcu_bh_state,
32 &per_cpu(rcu_bh_data, cpu));
33 }
34
35 void rcu_check_callbacks(int cpu, int user)
36 {
37 if (user ||
38 (idle_cpu(cpu) && !in_softirq() &&
39 hardirq_count() <= (1 << HARDIRQ_SHIFT))) {
40 rcu_qsctr_inc(cpu);
41 rcu_bh_qsctr_inc(cpu);
42 } else if (!in_softirq()) {
43 rcu_bh_qsctr_inc(cpu);
44 }
45 raise_softirq(RCU_SOFTIRQ);
46 }
```

图 D.23: rcu_check_callbacks() Code

图 D.23 显示的代码，在每一个 CPU 的每 jiffy 中，都会在 scheduling-clock 中断处理函数中调用。 rcu_pending() 函数 (是__rcu_pending()的一个封装函数) 被调用,如果它返回非 0，那么调用 rcu_check_callbacks()。 (注意有想法将 rcu_pending() 合并到 rcu_check_callbacks().)

从__rcu_pending()开始，第 4 行记录本调用次数，用于确定何时强制产生静止状态。第 6 行调用 check_cpu_stall()，目的是报告在内核中自旋的 CPUs，这些 CPUs 也许是存在硬件问题。如果配置了 CONFIG_RCU_CPU_STALL_DETECTOR，第 7-23 执行一系列的检查，如果 RCU 需要当前 CPU 做一些事情，就返回非 0 值。第 7 行检查当前 CPU 是否还缺少了一次静止状态，则在第 9 行调用 cpu_has_callbacks_ready_to_invoke()检查当前 CPU 是否有回调，并准备调用它们。第 11 行调用 cpu_needs_another_gp() 检查当前 CPU 是否有回调需要另外的 RCU grace period 完成。第 13 行检查当前 grace period 是否已经结束，第 15 行检查一个新 grace period 是否已经开始，最后，第 17-22 行检查是否应当强制其他 CPU 经历一次静止状态。最后一个检查可能会失

败： (1) 第 17-18 检查是否一个 breaks down 正在处理，如果是，则第 19-22 行检查是否 jiffies 足够长 (第 19-20 行) 或者调用 rcu_pending()次数足够多 (第 21-22) ，以确定是否应当调用 force_quiescent_state() 。如果这一系列的检查都没有触发，则第 24 行返回 0，表示不必调用 rcu_check_callbacks()。

第 27-33 行展示 rcu_pending()，它简单的调用__rcu_pending() 两次，一次是为``rcu"，另一次是为 ``rcu_bh"。

问题 D.29：在图 D.23 中，第 29-32 行 rcu_pending() 总是调用两次，没有办法合并对这两个结构的检查吗？

第 35-48 行展示 rcu_check_callbacks()，它检查 scheduling-clock 中断是否中止一个扩展静止状态，然后开始 RCU 软中断处理 (rcu_process_callbacks())。第 37-41 为“rcu” 执行这个检查，而第 42-43 行为``rcu_bh"执行这个检查。

第 37-39 行调度时钟中断是否打断了用户态执行 (第 37 行)，或者打断了 idle 循环 (第 38 行的 idle_cpu()) ，并且没有在中断上下文(第 38 行的剩余部分及整个第 39 行)。如果这个检查成功，则调度时钟中断来自于扩展静止状态，由于任何 rcu 的静止状态也是 rcu_bh 的静止状态，那么第 40、41 行报告静止状态。

类似于``rcu_bh"，第 42 行检查调度时钟中断是否来自于软中断允许区域，如果是，则第 43 行报告“rcu_bh”静止状态。

问题 D.30：图 D.23 第 42 行是否不应当检查 in_hardirq()?

其他情况下，第 45 行触发一个 RCU 软中断，将导致在将来的某个时刻，本 CPU 上的 rcu_process_callbacks() 将被调用。

D.3.2.4 rcu_process_callbacks()

```
1 static void
2 __rcu_process_callbacks(struct rcu_state *rsp,
3 struct rcu_data *rdp)
4 {
5 unsigned long flags;
6
7 if ((long)(ACCESS_ONCE(rsp->jiffies_force_qs) -
8 jiffies) < 0 ||
9 (rdp->n_rcu_pending_force_qs -
10 rdp->n_rcu_pending) < 0)
11 force_quiescent_state(rsp, 1);
12 rcu_process_gp_end(rsp, rdp);
13 rcu_check_quiescent_state(rsp, rdp);
14 if (cpu_needs_another_gp(rsp, rdp)) {
```

```
15 spin_lock_irqsave(&rcu_get_root(rsp)->lock, flags);
16 rcu_start_gp(rsp, flags);
17 }
18 rcu_do_batch(rdp);
19 }
20
21 static void
22 rcu_process_callbacks(struct softirq_action *unused)
23 {
24 smp_mb();
25 __rcu_process_callbacks(&rcu_state,
26 &__get_cpu_var(rcu_data));
27 __rcu_process_callbacks(&rcu_bh_state,
28 &__get_cpu_var(rcu_bh_data));
29 smp_mb();
30 }
```

图 D.24: rcu_process_callbacks() Code

图 D.24 展示了 rcu_process_callbacks()的代码，它是__rcu_process_callbacks()函数的封装。这些函数是调用 raise_softirq(RCU_SOFTIRQ)的函数的结果。例如，图 D.23 中第 47 行，当有某种原因使得有理由相信 RCU 核心需要当前 CPU 做某些事情的时候会触发此函数。

第 7-10 行检查自当前 grace period 启动以来，是否已经经历了一段时间。如果是这样，第 11 行调用 force_quiescent_state()，其目的是试图使相应的 CPUs 经历一次静止状态。

问题 D.31：但是我们在__rcu_process_callbacks 中不必检查 grace period 正在处理中吗?

在任何情况下，第 12 行调用 rcu_process_gp_end()，它检查其他 CPUs 是否结束了本 CPU 关注的 grace period。如果是，标记 grace period 结束并且加快调用本 CPUs 的 RCU 回调。第 13 行调用 rcu_check_quiescent_state()，该函数检查其他 CPUs 是否启动了一个新的 grace period，并检查当前 CPU 是否已经为这个 grace period 经历了一次静止状态。如果是这样，就更新相应的状态。第 14 行检查是否没有正在处理的 grace period，并且检查是否有另外的 grace period 所需要的回调函数，如果是这样，第 15 行获得根 rcu_node 的锁，第 17 行调用 rcu_start_gp()，开始一个新的 grace period(并且也释放根 rcu_node 的锁)。其他情况下，第 18 行调用 rcu_do_batch()，它调用本 CPU 的回调。

问题 D.32：如图 D.24，如果两个 CPU 尝试并发的开始 grace period 会发生什么?

第 21-30 行是 rcu_process_callbacks()，也是一个对__rcu_process_callbacks()的封装函数。第 24 行执行一个内存屏障，以确保前面的 RCU 读临界区在随后的 RCU 处理过程之前结束。第 25-26 行和 27-28 行分别为``rcu" 和 ``rcu_bh" 调用 __rcu_process_callbacks()，最后，第 29 行执行一个内存屏障，以确保任何 __rcu_process_callbacks() 完成的操作，都早于随后的 RCU 读临界区。

D.3.2.5 rcu_needs_cpu() 和 rcu_cpu_notify()

```
1 int rcu_needs_cpu(int cpu)
2 {
3 return per_cpu(rcu_data, cpu).nxtlist ||
4 per_cpu(rcu_bh_data, cpu).nxtlist;
5 }
6
7 static int __cpuinit
8 rcu_cpu_notify(struct notifier_block *self,
9 unsigned long action, void *hcpu)
10 {
11 long cpu = (long)hcpu;
12
13 switch (action) {
14 case CPU_UP_PREPARE:
15 case CPU_UP_PREPARE_FROZEN:
16 rcu_online_cpu(cpu);
17 break;
18 case CPU_DEAD:
19 case CPU_DEAD_FROZEN:
20 case CPU_UP_CANCELED:
21 case CPU_UP_CANCELED_FROZEN:
22 rcu_offline_cpu(cpu);
23 break;
24 default:
25 break;
26 }
27 return NOTIFY_OK;
28 }
```

图 D.25: rcu_needs_cpu() and rcu_cpu_notify Code

图 D.25 显示了 rcu_needs_cpu()和 rcu_cpu_notify()的代码，它们被 LINUX 内核调用，以检查 dynticks-idle 模式转换并处理 CPU 热插拔。

第 1-5 行显示了 rcu_needs_cpu()的代码，它简单的检查是否特定的 CPU 有

``rcu" (line 3) 或者 ``rcu_bh" (line 4) 回调。

第 7-28 显示了 rcu_cpu_notify()，它是一个非常典型的使用 switch 语句的 CPU 热插拔通知函数。如果特定的 CPU 上线，则在第 16 行调用 rcu_online_cpu() ，如果特定 CPU 离线，则在第 22 行调用 rcu_offline_cpu。请注意：CPU 热插拔不是原子的，可能穿越多个 grace periods。因此必须小心的处理 CPUs 热插拔事件。

D.3.3 初始化

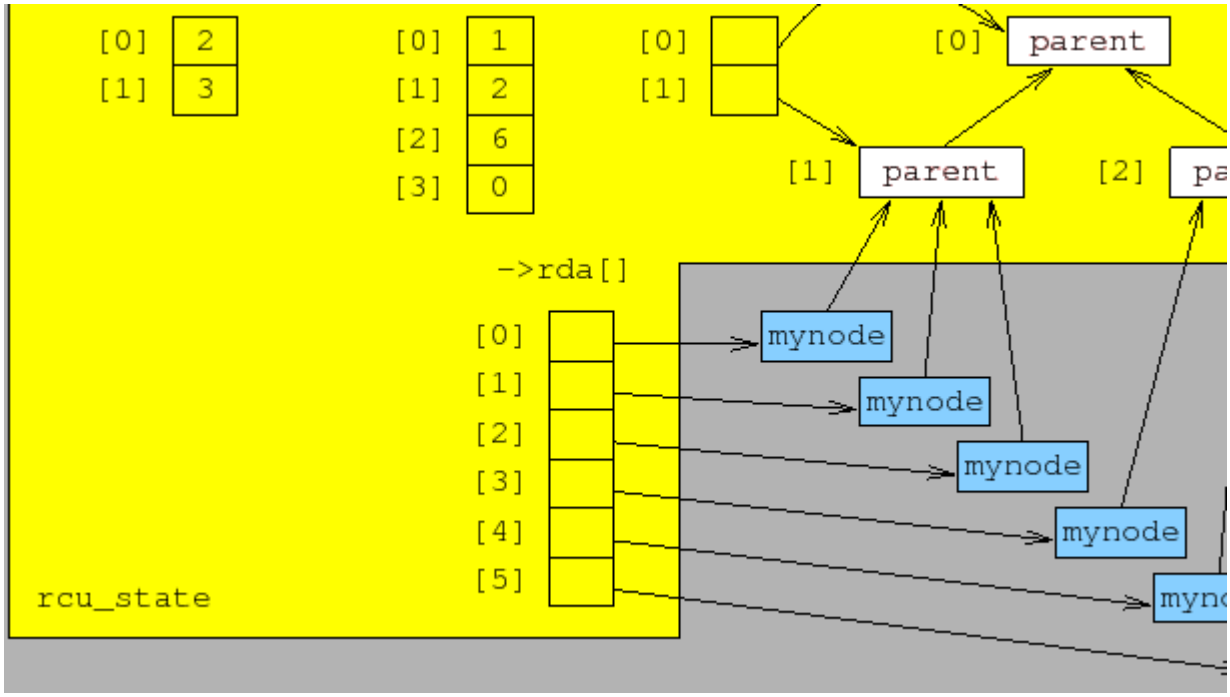


图 D.26: Initialized RCU Data Layout

本节浏览一下初始化代码，主要的关系如图 D.26 示。黄色区域是 rcu_state 中的字段，包括->node 数组，个别字段以粉红色显示，与 D.2 节的规范是一致的。蓝色框是 rcu_data 结构。每一个蓝色框表示每 CPU 的 rcu_data 集合。

->levelcnt[] 数组在编译时初始化，如 ->level[0],但是其他的值和指针由随后章节中的函数填充。该图显示了一个两级的分级结构，但是一级和三级的分级结构也是允许的。->levelspread[] 数组的每一个元素指出了相应层的子节点数量。因此，根节点有两个子结点，每一个叶子结点有三个子节点。每一个 levelcnt[] 数组元素表示相应层上有多少节点：根层是 1，叶子层是 2，rcu_data 层是 6 －其他额外的元素是无用的，其值为 0。每一个 ->level[]数组元素是相应的层的第一个 rcu_node。每一个->rda[]数组元素是相应的 CPU 的 rcu_data。每一个 rcu_node 结构的->parent 字段是它的父指针。根 rcu_node 的 parent 是 NULL。最后， 每一个 rcu_data 的->mynode 字段是它的父 rcu_data 字段。

问题 D.33：如何从根到叶子遍历 rcu_node？

随后的章节中，我们来看看如何构建这个结构。

D.3.3.1 rcu_init_levelspread()

```
#ifdef CONFIG_RCU_FANOUT_EXACT
2static void __init rcu_init_levelspread(struct rcu_state *rsp)
3{
4    int i;
5
6    for (i = NUM_RCU_LVLS - 1; i >= 0; i--)
7            rsp->levelspread[i] = CONFIG_RCU_FANOUT;
8}
9#else /* #ifdef CONFIG_RCU_FANOUT_EXACT */
10static void __init rcu_init_levelspread(struct rcu_state *rsp)
11{
12   int ccur;
13   int cprv;
14   int i;
15
16   cprv = NR_CPUS;
17   for (i = NUM_RCU_LVLS - 1; i >= 0; i--) {
18           ccur = rsp->levelcnt[i];
19           rsp->levelspread[i] = (cprv + ccur - 1) / ccur;
20           cprv = ccur;
21   }
22}
23#endif /* #else #ifdef CONFIG_RCU_FANOUT_EXACT */
```

图 D.27: rcu_init_levelspread() Code

图 D.27 展示了 rcu_init_levelspread() 函数的代码，它控制 fanout 的值，或者说在 rcu_node 分级体系中每一个父结点的子节点的数量。有两个版本的函数，其中一个在第 2-9 行展示了固定的 fanout (由 CONFIG_RCU_FANOUT 指定)，第 11-25 行根据 fanout 确定子节点的数量，然后平衡树。

精确 fanout 版本简单的指定 rcu_state 结构的->levelspread 数组的每一个元素赋值为 CONFIG_RCU_FANOUT 内核参数，如第 7、8 行的循环。

在第 11-24 行的可平衡分级版本使用一对局部变量 ccur 和 cprv ，这对变量跟踪当前层和前一层的 rcu_node 数量。这个函数从叶子开始向上级运行，因此 cprv 在第 18 行初始化为 NR_CPUS，适当数量的 rcu_data 结构被放到叶子层。第 19-23 行从叶子层到根层进行遍历处理。在这个循环中，第 20 行取得当前层

的 rcu_node 结构数量，并存到 ccur。第 21 行根据前一级的节点数量，按比例向上取整得到当前级的节点数量，将结果放到 rcu_state 结构的 ->levelspread 数组中。第 22 设置下一次循环的变量。

调用任一一个函数后， ->levelspread 数组包含 rcu_node 分级体系中每一层的子节点数目。

D.3.3.2 rcu_init_one()

```
1 static void __init rcu_init_one(struct rcu_state *rsp)
2 {
3 int cpustride = 1;
4 int i;
5 int j;
6 struct rcu_node *rnp;
7
8 for (i = 1; i < NUM_RCU_LVLS; i++)
9 rsp->level[i] = rsp->level[i - 1] +
10 rsp->levelcnt[i - 1];
11 rcu_init_levelspread(rsp);
12 for (i = NUM_RCU_LVLS - 1; i >= 0; i--) {
13 cpustride *= rsp->levelspread[i];
14 rnp = rsp->level[i];
15 for (j = 0; j < rsp->levelcnt[i]; j++, rnp++) {
16 spin_lock_init(&rnp->lock);
17 rnp->qsmask = 0;
18 rnp->qsmaskinit = 0;
19 rnp->grplo = j * cpustride;
20 rnp->grphi = (j + 1) * cpustride - 1;
21 if (rnp->grphi >= NR_CPUS)
22 rnp->grphi = NR_CPUS - 1;
23 if (i == 0) {
24 rnp->grpnum = 0;
25 rnp->grpmask = 0;
26 rnp->parent = NULL;
27 } else {
28 rnp->grpnum = j % rsp->levelspread[i - 1];
29 rnp->grpmask = 1UL << rnp->grpnum;
30 rnp->parent = rsp->level[i - 1] +
31 j / rsp->levelspread[i - 1];
32 }
33 rnp->level = i;
34 }
```

```
35 }
36 }
```

图 D.28: rcu_init_one() Code

图 D.28 展示了 rcu_init_one()的代码，它在启动时初始化特定 rcu_state 结构。

回想一下 D.3.1.4 节，rcu_state 数据结构的->levelcnt[] 数组是在编译时初始化为分级体系结构的每一层的节点数量。另外一个结点被初始化为最大可能的 CPUs 数量：NR_CPUS。另外， ->level[] 数组的第一个元素在编译时初始化为根 rcu_node 结构。这个数组以宽度优先进行排序。请一定注意这一点，第 8-10 的循环初始化剩余的->level[]数组，这个数组是 rcu_node 分级体系中，每一层的第一个 rcu_node 结构。

第 11 行调用 rcu_init_levelspread()，它填充->levelspread[] 数组，这个数组在 D.3.3.1 节中描述。然后全部初始化辅助数组，因此准备开始第 15-35 行的循环，每一次循环初始化 rcu_node 的每一层，从叶子层开始。

第 13 行计算当前层的每 CPU 的 rcu_node 数量。第 14 行当前层的第一个 rcu_node 结构的指针，这是为第 15-34 行的循环而准备的，每一次循环会初始化一个 rcu_node 结构。

第 16-18 行初始化 rcu_node 结构的自旋锁和它的 CPU 掩码。qsmaskinit 字段是在启动阶段在线的 CPUs 集合。qsmask 字段是 grace period 启动时在线的 CPU 掩码。第 19 行设置 ->grplo 字段为 rcu_node 结构的第一个 CPU 编号。第 20 行设置->grphi 为这个 rcu_node 结构的最后一个 CPU 编号。如果分级体系的某一层的最后一个 rcu_node 结构仅仅是半满的，第 21、22 行设置它的->grphi 字段为系统中最后一个可能的 CPU 号。

第 24-26 行为根 rcu_node 初始化 ->grpnum，->grpmask 和 ->parent 字段，它没有父节点，因此这些字段为 0 或者 NULL。第 28-31 行为余下的 rcu_node 初始化这些字段。第 28 行计算 ->grpnum 字段为 rcu_node 在父中的索引。第 29 行在->grpmask 中设置相应的位。最后，第 30-31 行设置父指针到 ->parent 字段。这三个字段用于在分级体系中向上传播向上静止状态。

最后，第 22 行在->level 中记录分级所在的层。

D.3.3.3 _rcu_init()

```
1 #define RCU_DATA_PTR_INIT(rsp, rcu_data) \
2 do { \
3 rnp = (rsp)->level[NUM_RCU_LVLS - 1]; \
4 j = 0; \
5 for_each_possible_cpu(i) { \
```

```
6 if (i > rnp[j].grphi) \
7 j++; \
8 per_cpu(rcu_data, i).mynode = &rnp[j]; \
9 (rsp)->rda[i] = &per_cpu(rcu_data, i); \
10 } \
11 } while (0)
12
13 void __init __rcu_init(void)
14 {
15 int i;
16 int j;
17 struct rcu_node *rnp;
18
19 rcu_init_one(&rcu_state);
20 RCU_DATA_PTR_INIT(&rcu_state, rcu_data);
21 rcu_init_one(&rcu_bh_state);
22 RCU_DATA_PTR_INIT(&rcu_bh_state, rcu_bh_data);
23
24 for_each_online_cpu(i)
25 rcu_cpu_notify(&rcu_nb, CPU_UP_PREPARE,
26 (void *)(long)i);
27 register_cpu_notifier(&rcu_nb);
28 }
```

图 D.29: `__rcu_init()` Code

图 D.29 展示了__rcu_init() 函数的代码和它的 RCU_DATA_PTR_INIT() 辅助宏。__rcu_init() 函数在早期的启动阶段被调用。早于调度器的初始化，并且早于 CPU 运行阶段。

RCU_DATA_PTR_INIT() 宏有一个指向 rcu_state 的指针参数，以及 rcu_data 每 CPU 变量的名称。这个宏扫描每 CPU rcu_data 结构，赋予每一个 rcu_data 结构的 ->mynode 指针指向相应的 rcu_node 叶子节点。它也填充特定 rcu_state 结构的 ->rda[] 数组元素为指向相应 rcu_data 结构的指针。第 3 行获得指向第一个叶子 rcu_node 的指针，并存到局部变量 rnp 中 (必须由调用这个宏的调用 者定义)，第 4 行设置局部变量 j 为相应的叶子节点号。第 5-10 行是每一次循环的执行体，它为相应的 CPU 进行初始化 (由 NR_CPUS 指定)。在为个循环中，第 6 行检查是否已经超过了当前叶子 rcu_node 节点的边界，如果是，则在第 7 行移动到下一个结构。仍然在这个循环中，第 8 行设置当前 CPU 的 rcu_data 结构的 ->mynode 指针为当前叶子 rcu_node 结构。第 9 行设置当前 CPU 的->rda[] 元素 (在 rcu_state 结构中) 为当前 CPU 的 rcu_data 结构。

问题 D.34：C 预编译宏是 1990s 的东东了！为什么不与时俱进的将其中

的 RCU_DATA_PTR_INIT()转换为一个函数？

__rcu_init() 函数首先在第 19 行调用 rcu_init_one()，然后调用 RCU_DATA_PTR_INIT()。然后在第 21-22 行为 rcu_bo_state 重复执行这几步。第 24-26 行的循环为每一个在线的 CPU 调用 rcu_cpu_notify() (应当仅仅是启动 CPU)，第 27 行注册一个通知者，这样在 CPU 上线时，rcu_cpu_notify() 将会得到调用，用于告知 RCU 某个 CPU 上线了。

问题 D.35：如果在第 25-26 行最近的在线 CPU 被处理，第 27 行调用 register_cpu_notifier()之间，某一个 CPU 上线了，将会发生什么情况？

rcu_cpu_notify() 以及相关的函数将在随后的 D.3.4 节进行讨论。

D.3.4 CPU 热插拔

随后章节中描述的 CPU 热插拔函数允许 RCU 跟踪哪些 CPU 在线，哪些不在线，而且也在 CPU 上线时，完成每一个 CPU 的 rcu_data 结构的初始化。

D.3.4.1 rcu_init_percpu_data()

```
1 static void
2 rcu_init_percpu_data(int cpu, struct rcu_state *rsp)
3 {
4 unsigned long flags;
5 int i;
6 long lastcomp;
7 unsigned long mask;
8 struct rcu_data *rdp = rsp->rda[cpu];
9 struct rcu_node *rnp = rcu_get_root(rsp);
10
11 spin_lock_irqsave(&rnp->lock, flags);
12 lastcomp = rsp->completed;
13 rdp->completed = lastcomp;
14 rdp->gpnum = lastcomp;
15 rdp->passed_quiesc = 0;
16 rdp->qs_pending = 1;
17 rdp->beenonline = 1;
18 rdp->passed_quiesc_completed = lastcomp - 1;
19 rdp->grpmask = 1UL << (cpu - rdp->mynode->grplo);
20 rdp->nxtlist = NULL;
21 for (i = 0; i < RCU_NEXT_SIZE; i++)
22 rdp->nxttail[i] = &rdp->nxtlist;
```

```
23 rdp->qlen = 0;
24 rdp->blimit = blimit;
25 #ifdef CONFIG_NO_HZ
26 rdp->dynticks = &per_cpu(rcu_dynticks, cpu);
27 #endif /* #ifdef CONFIG_NO_HZ */
28 rdp->cpu = cpu;
29 spin_unlock(&rnp->lock);
30 spin_lock(&rsp->onofflock);
31 rnp = rdp->mynode;
32 mask = rdp->grpmask;
33 do {
34 spin_lock(&rnp->lock);
35 rnp->qsmaskinit |= mask;
36 mask = rnp->grpmask;
37 spin_unlock(&rnp->lock);
38 rnp = rnp->parent;
39 } while (rnp != NULL && !(rnp->qsmaskinit & mask));
40 spin_unlock(&rsp->onofflock);
41 cpu_quiet(cpu, rsp, rdp, lastcomp);
42 local_irq_restore(flags);
43 }
```

图 D.30: rcu_init_percpu_data() Code

图 D.30 展示了 rcu_init_percpu_data()的代码，它初始化特定 CPU 的 rcu_data 结构，以响应 CPU 启动或者 CPU 上线。它也设置 rcu_node 结构，这样该 CPU 将参与将来的 grace periods。

第 8 行获得该 CPU 的 rcu_data 结构指针，并将这个指针设置到局部变量 rdp 中。第 9 行获得根 rcu_node 结构指针，并设置到局部变量 rnp 中。

第 11-29 行在根 rcu_node 的锁保护下初始化 rcu_node 的字段，使用锁的目的是确保一致的数据。第 17 行对于跟踪来说是重要的，因为很多 LINUX 发行版将 NR_CPUS 设置为一个非常大的值，在跟踪 rcu_data 结构时，将导致过多的输出。->beenonline 字段用来解决这个问题。当相应的 CPU 上线时，它将设置这个字段为 1，其他 rcu_data 结构则是 0。这允许跟踪代码简单的忽略不相关的 CPUs。

第 30-40 行在 rcu_node 分级体系中传播 CPU 在线位。直到到达根 rcu_hode，或者相应的位已经设置。这个设置操作在->onofflock 的保护下进行，这样可以与新 grace period 的初始化进行互斥。另外，每一个 rcu_node 结构在它的锁保护下进行初始化。第 41 行调用 cpu_quiet() 来通知 RCU：该 CPU 处于扩展静止状态，最后，第 42 行重新打开中断。

问题 D.36：为什么在第 41 行调用 cpu_quiet() ？我们已经在几个锁的保

护下与 grace periods 进行了互斥，并且，更早的 grace periods 将不会等待较早前还处于离线状态的 CPU。

请注意：rcu_init_percpu_data() 不仅仅在启动时调用，也在 CPU 每次上线时调用。

D.3.4.2 rcu_online_cpu()

```
1 static void __cpuinit rcu_online_cpu(int cpu)
2 {
3 #ifdef CONFIG_NO_HZ
4 struct rcu_dynticks *rdtp;
5
6 rdtp = &per_cpu(rcu_dynticks, cpu);
7 rdtp->dynticks_nesting = 1;
8 rdtp->dynticks |= 1;
9 rdtp->dynticks_nmi = (rdtp->dynticks_nmi + 1) & ~0x1;
10 #endif /* #ifdef CONFIG_NO_HZ */
11 rcu_init_percpu_data(cpu, &rcu_state);
12 rcu_init_percpu_data(cpu, &rcu_bh_state);
13 open_softirq(RCU_SOFTIRQ, rcu_process_callbacks);
14 }
```

图 D.31: rcu_online_cpu() Code

图 D.31 展示了 rcu_online_cpu()的代码，它告知 RCU 相应的 CPU 准备上线。

当打开 dynticks 时 (CONFIG_NO_HZ)，第 6 行获得特定 CPU 的 rcu_dynticks 结构的引用，它在``rcu" 和 ``rcu_bh"实现中是共享的。第 7 行设置 ->dynticks_nesting 字段为 1，它反映了新上线 CPU 不处于 dynticks-idle 模式的事实 (回忆一下，->dynticks_nesting 字段跟踪了相应的 CPU 需要被 RCU 读临界区跟踪的原因数，由于当前能够运行进程级代码，因此将它置 1)。第 8 行强制 ->dynticks 字段为一个奇数，它至少与上一次 CPU 在线时的值一样大，它也反映了刚上线的 CPUs 目前不在 dynticks-idle 模式这个事实，第 9 行强制 ->dynticks_nmi 字段为一个偶数，并且至少与上一次在线时的值一样大，反映当前 CPU 不处于 NMI 中断处理函数的事实。

不管 CONFIG_NO_HZ 内核参数的值是什么，第 11-13 都会得到执行。第 11 行为“rcu”初始化特定 CPU 的 rcu_data 结构，第 12 行则是为``rcu_bh"。最后，第 13 行注册 rcu_process_callbacks()，在随后的本 CPU 上 raise_softirq()会调用它。

D.3.4.3 rcu_offline_cpu()

```
1 static void
2 __rcu_offline_cpu(int cpu, struct rcu_state *rsp)
3 {
4 int i;
5 unsigned long flags;
6 long lastcomp;
7 unsigned long mask;
8 struct rcu_data *rdp = rsp->rda[cpu];
9 struct rcu_data *rdp_me;
10 struct rcu_node *rnp;
11
12 spin_lock_irqsave(&rsp->onofflock, flags);
13 rnp = rdp->mynode;
14 mask = rdp->grpmask;
15 do {
16 spin_lock(&rnp->lock);
17 rnp->qsmaskinit &= ~mask;
18 if (rnp->qsmaskinit != 0) {
19 spin_unlock(&rnp->lock);
20 break;
21 }
22 mask = rnp->grpmask;
23 spin_unlock(&rnp->lock);
24 rnp = rnp->parent;
25 } while (rnp != NULL);
26 lastcomp = rsp->completed;
27 spin_unlock(&rsp->onofflock);
28 cpu_quiet(cpu, rsp, rdp, lastcomp);
29 rdp_me = rsp->rda[smp_processor_id()];
30 if (rdp->nxtlist != NULL) {
31 *rdp_me->nxttail[RCU_NEXT_TAIL] = rdp->nxtlist;
32 rdp_me->nxttail[RCU_NEXT_TAIL] =
33 rdp->nxttail[RCU_NEXT_TAIL];
34 rdp->nxtlist = NULL;
35 for (i = 0; i < RCU_NEXT_SIZE; i++)
36 rdp->nxttail[i] = &rdp->nxtlist;
37 rdp_me->qlen += rdp->qlen;
38 rdp->qlen = 0;
39 }
40 local_irq_restore(flags);
41 }
```

```
42
43 static void rcu_offline_cpu(int cpu)
44 {
45 __rcu_offline_cpu(cpu, &rcu_state);
46 __rcu_offline_cpu(cpu, &rcu_bh_state);
47 }
```

图 D.32: rcu_offline_cpu() Code

图 D.32 展示了__rcu_offline_cpu()和它的封装函数 rcu_offline_cpu().这个封装函数的目的是简单的调用两次 (图中第 43-47 行) __rcu_offline_cpu(),一次是为 ``rcu" ，另一次是为``rcu_bh"。__rcu_offline_cpu() 函数的目的是防止后面的 grace periods 等待将要离线的 CPU，标记扩展的静止状态，并且为该 CPU 上正在处理的 RCU 回调找到一个新的地方。

转到 __rcu_offline_cpu()，如图第 1-41 行所示。第 12 行获取特定 rcu_state 的 ->onofflock，这是为了与 grace-period 初始化进行互斥。

问题 D.37：但是如果 rcu_node 分级体系只有一个结构会如何，比如在一个非常小的系统中？在这种情况下如何防止与 grace-period 初始化相冲突。

第 13 行得到当前 CPU 相应的叶子 rcu_node 结构的指针，使用这个 CPU 的 rcu_data 结构的 ->mynode 指针。(参见图 D.26)。第 14 行使用叶子节点 rcu_node 结构的 qsmask 字段获得这个 CPU 的掩码。

第 15-25 行的循环体向上清除 rcu_node 分级体系结构的位，从这个 CPU 的叶子 rcu_node 结构开始。第 16 行获得当前 rcu_node 结构的->lock 锁，第 17 行从->qsmaskinit 字段中清除本 CPU 相应的位。这样后面的 grace periods 将不再等待本 CPU 的静止状态。如果象第 18 行中检测到的，->qsmaskinit 值不为 0，那么当前 rcu_node 结构还有其他在线 CPUs 需要跟踪，则么第 19 行释放当前 rcu_node 结构的锁，并且第 20 行退出循环。否则，我们需要继续向上遍历 rcu_node 分级体系。在这种情况下，第 22 行获得下一级需要用的掩码，第 23 行释放当前 rcu_node 结构的锁，第 24 行移到下一级。在到达顶层后，第 25 行退出循环。

问题 D.38：但是第 25 行会真的退出循环吗？为什么？

第 26 行获得特定 rcu_state 结构的->completed 字段，并存到局部变量 lastcomp 中，第 27 行释放->onofflock (但是中断仍然是关闭的)，第 28 行调用 cpu_quiet()，其目的是表示离线 CPU 现在处于一个扩展静止状态，传递 lastcomp 以避免向不同的 grace period 报告这个静止状态。

问题 D.39：假设第 26 行的执行顺序不是如图 D.32，lastcomp 被设置为前一个 grace period，这样当前的 grace period 会等待离线的 CPU 吗? 在这种情况

下，调用 cpu_quiet()不会错误的报告静止状态，并导致 grace period 永远等待离线 CPU 吗？

问题 D.40：一个离线 CPU 处于扩展静止状态，为什么第 28 行需要小心的处理 grace period？

第 29-39 行将离线 CPU 上的所有 RCU 回调函数移到正在运行的 CPU 上。这个操作必须避免将被移动的回调乱序。否则 rcu_barrier()将不能正常运行。第 29 行将指向当前运行 CPU 的 rcu_data 结构的指针放到局部变量 rdp_me 中。第 30 行检查要离线的的 CPU 是否有需要处理的 RCU 回调。如果有，第 31-38 行移动它们。第 31 行将回调列表连接到运行 CPU 列表的尾部。第 32-33 行设置正在运行的 CPU 的回调尾部指针为离线 CPU 的尾部指针。然后，第 34-36 行初始化离线 CPU 的回调列表为空。第 37 行将离线 CPU 的列表长度加到正在运行的 CPU 中，最后，第 38 行将离线 CPU 列表长度设置为 0。

问题 D.41：但是这个列表移动操作使得离线 CPU 的回调需要经历下一个 grace period，即使正准备调用它们，这会不会没有效率?而且，如果一个 CPU 离线又上线，是否不能防止一个特定的已经被调用的回调？

最后，第 40 行重新打开中断。

D.3.5 杂项函数

本节描述杂项函数：

rcu_batches_completed

rcu_batches_completed_bh

cpu_has_callbacks_ready_to_invoke

cpu_needs_another_gp

rcu_get_root

```
1 long rcu_batches_completed(void)
2 {
3 return rcu_state.completed;
4 }
5
6 long rcu_batches_completed_bh(void)
7 {
8 return rcu_bh_state.completed;
9 }
10
11 static int
12 cpu_has_callbacks_ready_to_invoke(struct rcu_data *rdp)
```

```
13 {
14 return &rdp->nxtlist != rdp->nxttail[RCU_DONE_TAIL];
15 }
16
17 static int
18 cpu_needs_another_gp(struct rcu_state *rsp,
19 struct rcu_data *rdp)
20 {
21 return *rdp->nxttail[RCU_DONE_TAIL] &&
22 ACCESS_ONCE(rsp->completed) ==
23 ACCESS_ONCE(rsp->gpnum);
24 }
25
26 static struct rcu_node
27 *rcu_get_root(struct rcu_state *rsp)
28 {
29 return &rsp->node[0];
30 }
```

图 D.33: Miscellaneous Functions

图 D.33 展示了一些杂项函数。第 19 行显示了 rcu_batches_completed()和 rcu_batches_completed_bh(),用于 rcutorture 测试套件.第 11-15 行显示了 cpu_has_callbacks_ready_to_invoke(),它表示特定 rcu_data 是否有 RCU 回调需要经历它们的 grace period。第 17-24 行显示了 cpu_needs_another_gp()。最后，第 26-30 行展示了 rcu_get_root()，它返回与特定 rcu_state 结构相关的根 rcu_node。

D.3.6 Grace-Period 检测函数

本节包含的函数被 grace periods 开始结束检测函数直接调用。这当然包含开始结束 grace periods 的函数。

D.3.6.1 标记新 Grace Periods

分级 RCU 的一个主要目的是检测 grace periods，直接调用的函数在本节中描述。D.3.6.1 节包含允许 CPus 标记一个新 grace period 已经开始的函数，D.3.6.2 节的函数允许 CPUs 标记一个存在的 grace period 已经结束，D.3.6.3 节的函数 rcu_start_gp()，开始一个新的 grace period，D.3.6.4 节的函数包括报告 CPUs 的静止状态。

```
1 static void note_new_gpnum(struct rcu_state *rsp,
2 struct rcu_data *rdp)
```

```
3 {
4 rdp->qs_pending = 1;
5 rdp->passed_quiesc = 0;
6 rdp->gpnum = rsp->gpnum;
7 rdp->n_rcu_pending_force_qs = rdp->n_rcu_pending +
8 RCU_JIFFIES_TILL_FORCE_QS;
9 }
10
11 static int
12 check_for_new_grace_period(struct rcu_state *rsp,
13 struct rcu_data *rdp)
14 {
15 unsigned long flags;
16 int ret = 0;
17
18 local_irq_save(flags);
19 if (rdp->gpnum != rsp->gpnum) {
20 note_new_gpnum(rsp, rdp);
21 ret = 1;
22 }
23 local_irq_restore(flags);
24 return ret;
25 }
```

图 D.34: Noting New Grace Periods

图 D.34 显示了 note_new_gpnum()的代码，它更新状态以反映一个新的 grace period，也包含函数 check_for_new_grace_period()，被 CPUs 用来检测其他 CPUs 何时开始一个新 grace period。

note_new_gpnum()的第 4 行设置->qs_pending 标志，表示本 CPU 的 RCU 需要一个静止状态。第 5 行清除 ->passed_quiesc 标志，表示本 CPU 还没有经历过这样一个静止状态。第 6 行从全局 rcu_state 结构中复制 grace-period 号到该 CPU 的 rcu_data 结构中，这样，本 CPU 将记下它已经标记了新 grace period。最后，第 7-8 行记录 CPU 试图强制其他 CPU 经历静止状态(通过调用 force_quiescent_state())的时间，以 jiffies 计算。假设 grace period 没有提前结束。

check_for_new_grace_period()的第 18、23 行禁止并重新打开中断。第 19 行检查当前 CPU 是否还有新 grace period 没有标记，如果是，则在第 20 行调用 note_new_gpnum()，其目的是标记新 grace period，第 21 行设置适当的返回值。第 24 行返回状态：如果新 grace period 已经开始，则返回非 0，否则返回 0。

问题 D.42：为什么不仅仅将 note_new_gpnum() 内联到 check_for_new_grace_period() ？

D.3.6.2 标记旧 Grace Periods 结束

```
1 static void
2 rcu_process_gp_end(struct rcu_state *rsp,
3 struct rcu_data *rdp)
4 {
5 long completed_snap;
6 unsigned long flags;
7
8 local_irq_save(flags);
9 completed_snap = ACCESS_ONCE(rsp->completed);
10 if (rdp->completed != completed_snap) {
11 rdp->nxttail[RCU_DONE_TAIL] =
12 rdp->nxttail[RCU_WAIT_TAIL];
13 rdp->nxttail[RCU_WAIT_TAIL] =
14 rdp->nxttail[RCU_NEXT_READY_TAIL];
15 rdp->nxttail[RCU_NEXT_READY_TAIL] =
16 rdp->nxttail[RCU_NEXT_TAIL];
17 rdp->completed = completed_snap;
18 }
19 local_irq_restore(flags);
20 }
```

图 D.35: Noting End of Old Grace Periods

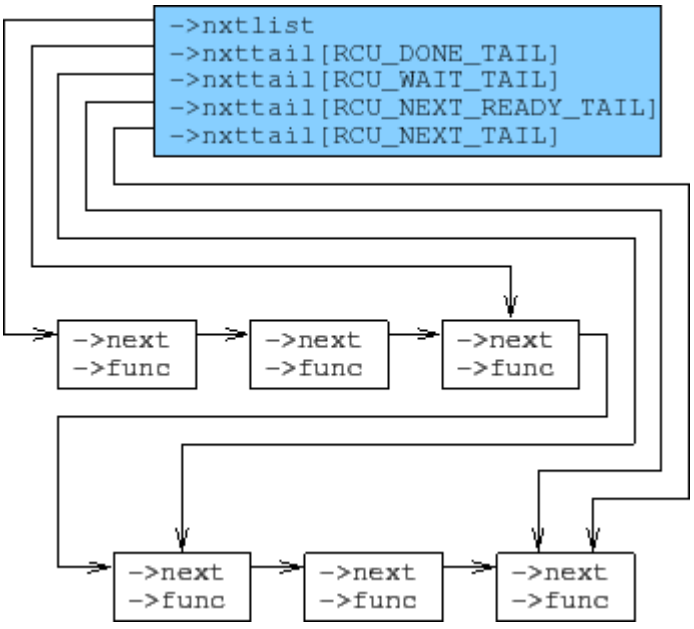


图 D.36: RCU Callback List

图 D.35 展示了 rcu_process_gp_end()，当 CPU 察觉 grace period 已经结束时

调用此函数。如果 grace period 已经结束，那么这个函数提前运行当前 CPU 的 RCU 回调，这些回调由一个单向链表管理，这个单向链表有多个尾指针，如图。多尾指针布局最初由 Lai Jiangshan 提出，简化了链表处理 [Jia08]。在这个图中，蓝色表示一个 CPU 的 rcu_data 结构，底部 6 个白色框表示 6 个 RCU 回调的链表(rcu_head)。在这个链表中，前三个回调已经经历了它们的 grace period，因此正等待被调用。第 4 个回调（第二行的第一个）等待当前 grace period 完成，最后两个等待下一个 grace period。最后两个尾指针是最后一个元素，因此，还没有与特定 grace period 相关的回调是空的。

图 D.35 的第 8、19 行禁止并重新打开中断。第 9 行获得 rcu_state 结构的->completed 字段的快照值，并存储到局部变量 completed_snap 中。第 10 行检查当前 CPU 是否仍然没有注意到 grace period 已经结束。如果没有注意到这一点，则第 11-16 行通过操作尾指针提前处理这个 CPU 的 RCU 回调。第 17 行记录最近完成的 grace period 编号到该 CPU 的 rcu_data 结构的->completed 字段中。

D.3.6.3 启动 Grace Period

```
1 static void
2 rcu_start_gp(struct rcu_state *rsp, unsigned long flags)
3 __releases(rcu_get_root(rsp)->lock)
4 {
5 struct rcu_data *rdp = rsp->rda[smp_processor_id()];
6 struct rcu_node *rnp = rcu_get_root(rsp);
7 struct rcu_node *rnp_cur;
8 struct rcu_node *rnp_end;
9
10 if (!cpu_needs_another_gp(rsp, rdp)) {
11 spin_unlock_irqrestore(&rnp->lock, flags);
12 return;
13 }
14 rsp->gpnum++;
15 rsp->signaled = RCU_GP_INIT;
16 rsp->jiffies_force_qs = jiffies +
17 RCU_JIFFIES_TILL_FORCE_QS;
18 rdp->n_rcu_pending_force_qs = rdp->n_rcu_pending +
19 RCU_JIFFIES_TILL_FORCE_QS;
20 record_gp_stall_check_time(rsp);
21 dyntick_record_completed(rsp, rsp->completed - 1);
22 note_new_gpnum(rsp, rdp);
23 rdp->nxttail[RCU_NEXT_READY_TAIL] =
24 rdp->nxttail[RCU_NEXT_TAIL];
```

```
25 rdp->nxttail[RCU_WAIT_TAIL] =
26 rdp->nxttail[RCU_NEXT_TAIL];
27 if (NUM_RCU_NODES == 1) {
28 rnp->qsmask = rnp->qsmaskinit;
29 spin_unlock_irqrestore(&rnp->lock, flags);
30 return;
31 }
32 spin_unlock(&rnp->lock);
33 spin_lock(&rsp->onofflock);
34 rnp_end = rsp->level[NUM_RCU_LVLS - 1];
35 rnp_cur = &rsp->node[0];
36 for (; rnp_cur < rnp_end; rnp_cur++)
37 rnp_cur->qsmask = rnp_cur->qsmaskinit;
38 rnp_end = &rsp->node[NUM_RCU_NODES];
39 rnp_cur = rsp->level[NUM_RCU_LVLS - 1];
40 for (; rnp_cur < rnp_end; rnp_cur++) {
41 spin_lock(&rnp_cur->lock);
42 rnp_cur->qsmask = rnp_cur->qsmaskinit;
43 spin_unlock(&rnp_cur->lock);
44 }
45 rsp->signaled = RCU_SIGNAL_INIT;
46 spin_unlock_irqrestore(&rsp->onofflock, flags);
47 }
```

图 D.37: Starting a Grace Period

图 D.37 显示了 rcu_start_gp(),它开始一个新的 grace period,也释放根 root_node 结构的锁，这个锁必须由调用者获取。

局部变量 rdp 是当前 CPU 的 rcu_data 结构，rnp 是根 rcu_node 结构，rnp_cur 和 rnp_end 作为指针，用来遍历 rcu_nod 分级体系。

第 10 行调用 cpu_needs_another_gp()来检查当前 CPU 是否需要启动另外一个 grace period，如果不需要，第 11 行释放根 rcu_node 结构的锁，并在第 12 行返回。由于多个 CPUs 试图同时启动一个 grace period，就可能运行这个代码分支。在这种情况下，胜利者将启动 grace period，其他将通过这个代码路径退出。

否则，第 14 行递增特定 CPU 的 rcu_state 结构的 ->gpnum 字段，正式的表示新 grace period 开始了。

问题 D.43：但在第 15 行没有初始化过程!如果一个 CPU 宣告一个新 grace period 开始，并且试图立即报告一个静止状态将发生什么？这不会变得让人困惑吗?

第 15 行设置->signaled 字段为 RCU_GP_INIT ，其目的是防止其他 CPU 试图在初始化完成前强制结束新 grace period。第 16-18 对下一次强制结束 grace

period 进行调度，首先是设置 jiffies，其次是设置调用 rcu_pending 的次数。当然，如果 grace period 正常结束，就没有必要强制结束它。第 20 行调用 record_gp_stall_check_time() 来调度长周期检测—如果 grace period 超过设定的时间，则认为出现一个错误。第 22 行调用 note_new_gpnum() 来初始化这个 CPU 的 rcu_data 结构。

第 23-26 行前移本 CPU 的所有回调，这样在新 grace period 结束时，它们将被调用。这将使得回调被加快执行，其他 CPU 仅仅为当前 grace period 移动 RCU_NEXT_READY_TAIL 组。RCU_NEXT_TAIL 组将前移为 RCU_NEXT_READY_组。本 CPU 可以加快 RCU_NEXT_TAIL 组是由于它知道何时开始新 grace period。相对的，其他 CPUs 不能正确的处理启动新 grace period、新 RCU 到达方面的竞争。

第 27 行检查在分级体系中，是否仅仅只有一个 rcu_node 结构，如果是这样，第 28 行向所有在线 CPU 设置 ->qsmask 位，换句话说，为了使 grace period 结束，这些相应的 CPU 必须经历一次静止状态。第 29 行释放根 rcu_node 结构的锁，并在第 30 行返回。在这种情况下，可以期望 gcc 废代码去除功能消除第 32-46 行。

否则，rcu_node 分级体系拥有多个结构，这需要更复杂的初始化。第 32 行释放根 rcu_node 的锁，但是中断仍然被禁止。然后第 33 行获得特定 rcu_state 的>onofflock，防止 CPU 热插拔在 RCU 状态维护操作之间的冲突。

第 34 行设置局部变量 rnp_end 为第一个叶子 rcu_node 结构。第 35 行设置局部变量 rnp_cur local 为根 rcu_node 结构，它也是-node 数组的第一个结构。第 36 、 37 行遍历所有非叶子 rcu_node 结构，设置那些必须经历静止状态的 CPUs 的下级 rcu_node 相应的位。

问题 D.44：嗨！当第 37 行处理状态时，不应当获得非叶子 rcu_node 结构的锁吗？？？

第 38 行设置局部变量 rnp_end 为最后一个 rcu_node 结构，第 39 行设置局部变量 rnp_cur 为第一个叶子 rcu_node 结构， 这样第 40-44 行的循环将遍历所有 rcu_node 分级体系的叶子结点。在每一次循环中，第 41 行获得当前叶子 rcu_node 结构的锁，第 42 行设置在线 CPUs 的相应位，第 43 行释放锁。

问题 D.45：为什么我们不能合并第 36-37 行以及第 40-44 行的循环？

第 45 行设置特定 rcu_state 结构的 ->signaled 字段以允许强制静止状态，第 46 行释放->onofflock 以允许 CPU 热插拔操作维护 RCU 状态。

D.3.6.4 报告静止状态

分级 RCU 实现了一种分级报告静止状态的方法，使用了以下函数：

当一个为``rcu" 和 ``rcu_bh"经历一次静止状态时，分别调用 rcu_qsctr_inc()和 rcu_bh_qsctr_inc()。注意 dynticks-idle 和 CPU-离线这两种静止状态是特殊处理的，因为事实上这样的 CPU 没有在运行状态，因此不能主动报告自己处于静止状态。

rcu_check_quiescent_state()检查当前 CPU 是否已经经历一次静止状态，如果是这样，就调用 cpu_quiet()。

cpu_quiet() 通过调用 cpu_quiet_msk()来报告指定 CPU 已经经过了静止状态。特定 CPU 必须是当前 CPU 或者一个离线 CPU。

cpu_quiet_msk() 报告一组经历了静止状态的 CPU。这些 CPUs 必须是当前 CPU 或者它都离线了。

下面描述这些函数.

```
1 void rcu_qsctr_inc(int cpu)
2 {
3 struct rcu_data *rdp = &per_cpu(rcu_data, cpu);
4 rdp->passed_quiesc = 1;
5 rdp->passed_quiesc_completed = rdp->completed;
6 }
7
8 void rcu_bh_qsctr_inc(int cpu)
9 {
10 struct rcu_data *rdp = &per_cpu(rcu_bh_data, cpu);
11 rdp->passed_quiesc = 1;
12 rdp->passed_quiesc_completed = rdp->completed;
13 }
```

图 D.38: Code for Recording Quiescent States

图 D.38 展示了 rcu_qsctr_inc() 和 rcu_bh_qsctr_inc()的代码，它们标记当前 CPU 经过了一次静止状态。

rcu_qsctr_inc()的第 3 行获得特定 CPU 的 rcu_data 结构指针。第 4 行设置->passed_quiesc 字段，以记录静止状态，第 5 行设置->passed_quiesc_completed 字段为上一次完成的 grace period 编号 (存储在 rcu_data 结构的 ->completed 字段)。

rcu_bh_qsctr_inc() 函数完全一样，仅仅是第 10 行从 rcu_bh_data 每 CPU 变量中获得 rcu_data 指针，而不是 rcu_data 每 CPU 变量。

```
1 static void
2 rcu_check_quiescent_state(struct rcu_state *rsp,
3 struct rcu_data *rdp)
```

```
4 {
5 if (check_for_new_grace_period(rsp, rdp))
6 return;
7 if (!rdp->qs_pending)
8 return;
9 if (!rdp->passed_quiesc)
10 return;
11 cpu_quiet(rdp->cpu, rsp, rdp,
12 rdp->passed_quiesc_completed);
13 }
```

图 D.39: Code for rcu_check_quiescent_state()

图 D.39 展示了 rcu_check_quiescent_state()的代码，它被 from rcu_process_callbacks()调用(在 D.3.2.4 节中描述)，以确定其他 CPUs 何时开始了一个新 grace period，并向本 CPU 通告最近的静止状态。

第 5 行调用 check_for_new_grace_period() 来检查一个新的 grace period 已经被其他 CPU 启动，并且更新本 CPU 的局部状态以表明这个新的 grace period 。如果新的 grace period 已经启动，第 6 行则返回。第 7 行检查 RCU 是否仍然期望当前 CPU 的静止状态。如果不是，则返回。第 9 行检查自当前 grace period 启动以来，本 CPU 是否已经经历了一次静止状态。如果没有，则第 10 行返回。

因此，当到达第 11 行时，表示前面的 grace period 仍然有效，如果本 CPU 需要经历一次静止状态以允许这个 grace period 结束，并且如果这个 CPU 已经经历了这样一个静止状态。这种情况下，第 11-12 行调用 cpu_quiet() 向 RCU 报告本次静止状态。

问题 D.46: 什么可以防止图 D.39 的第 11-12 行错误的将上一个 grace period 的静止状态向当前 grace period 报告？

```
1 static void
2 cpu_quiet(int cpu, struct rcu_state *rsp,
3 struct rcu_data *rdp, long lastcomp)
4 {
5 unsigned long flags;
6 unsigned long mask;
7 struct rcu_node *rnp;
8
9 rnp = rdp->mynode;
10 spin_lock_irqsave(&rnp->lock, flags);
11 if (lastcomp != ACCESS_ONCE(rsp->completed)) {
12 rdp->passed_quiesc = 0;
13 spin_unlock_irqrestore(&rnp->lock, flags);
```

```
14 return;
15 }
16 mask = rdp->grpmask;
17 if ((rnp->qsmask & mask) == 0) {
18 spin_unlock_irqrestore(&rnp->lock, flags);
19 } else {
20 rdp->qs_pending = 0;
21 rdp = rsp->rda[smp_processor_id()];
22 rdp->nxttail[RCU_NEXT_READY_TAIL] =
23 rdp->nxttail[RCU_NEXT_TAIL];
24 cpu_quiet_msk(mask, rsp, rnp, flags);
25 }
26 }
```

图 D.40: Code for `cpu_quiet()`

图 D.40 展示了 cpu_quiet，这用来为特定 CPU 报告一次静止状态。正如较早前看到的一样，它必须是当前运行的 CPU 或者是离线 CPU。

第 9 行获得本 CPU 的 rcu_node 叶子节点结构指针。第 10 行获得叶子 rcu_node 的锁并禁止中断。第 11 行进行检查，以确保特定 grace period 仍然有效，如果是这样的话，第 11 行清除标志，以表示该 CPU 经历了一次静止状态。第 13 行释放锁并重新打开中断，第 14 行返回调用者。

否则，第 16 行为特定 CPU 生成一个掩码。第 17 行检查相应的位是否仍然在叶子 rcu_node 结构中，如果不在结构中，则在第 18 行释放锁并重新打开中断。

另一方面，如果 CPU 位仍然存在在，第 20 行清除->qs_pending，这反映了该 CPU 已经为该 grace period 经历了一次静止状态。第 21 行改写局部变量 rdp 为当前 CPU 的 rcu_data 结构指针。第 22-23 行更新运行 CPU 的回调。最后，第 24 行从下往上清除 rcu_node 分级体系的位。注意 cpu_quiet() 释放锁并重新打开中断。

问题 D.47：第 22-23 行如何知道提前运行 CPU 的 RCU 回调是安全的？

```
1 static void
2 cpu_quiet_msk(unsigned long mask, struct rcu_state *rsp,
3 struct rcu_node *rnp, unsigned long flags)
4 __releases(rnp->lock)
5 {
6 for (;;) {
7 if (!(rnp->qsmask & mask)) {
8 spin_unlock_irqrestore(&rnp->lock, flags);
9 return;
10 }
11 rnp->qsmask &= ~mask;
```

```
12 if (rnp->qsmask != 0) {
13 spin_unlock_irqrestore(&rnp->lock, flags);
14 return;
15 }
16 mask = rnp->grpmask;
17 if (rnp->parent == NULL) {
18 break;
19 }
20 spin_unlock_irqrestore(&rnp->lock, flags);
21 rnp = rnp->parent;
22 spin_lock_irqsave(&rnp->lock, flags);
23 }
24 rsp->completed = rsp->gpnum;
25 rcu_process_gp_end(rsp, rsp->rda[smp_processor_id()]);
26 rcu_start_gp(rsp, flags);
27 }
```

图 D.41: Code for `cpu_quiet_msk()`

图 D.41 展示了 cpu_quiet_msk()，它更新 rcu_node 分级体系以反映掩码参数表示的 CPUs 经历了静止状态。注意参数 rnp 是相应 CPUs 的叶子 rcu_node。

问题 D.48：第 2 行的掩码参数是 unsigned long，它在超过 64 个 CPU 的系统上是如何处理的？

第 4 行是一个注解，表示 cpu_quiet_msk() 释放叶子 rcu_node 结构的锁。

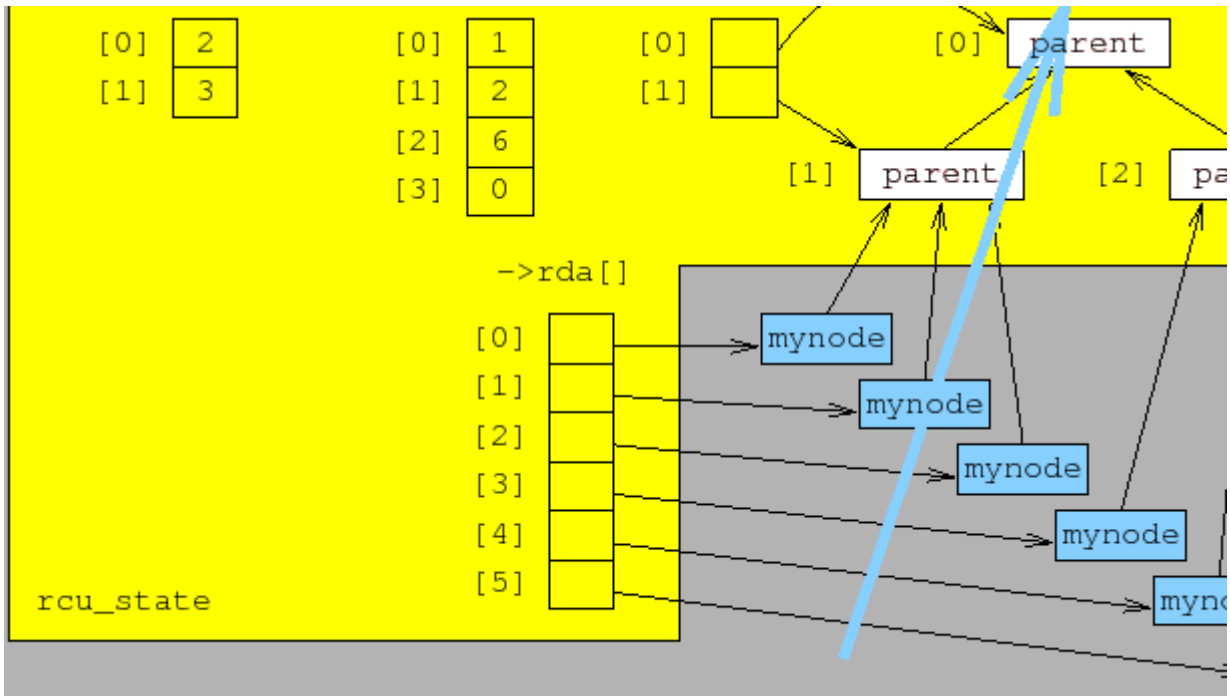


图 D.42: Scanning `rcu_node` Structures When Applying

Quiescent States

第 6-23 行的每一次循环处理需要 rcu_node 分级体系中的一级，遍历图 D.42 中蓝色箭头表示的结构。

第 7 行检查当前 rcu_node 结构的->qsmask 字段中的所有位是否已经清除，如果是，则第 8 行释放锁并重新打开中断。并在第 9 行返回调用者。如果不是，第 11 行从 rcu_node 结构的 qsmask 字段中清除掩码对应的位。第 12 行检查->psmask 中是否还存在剩余的位，如果是，第 13 行释放锁并重新打开中断，并在第 14 位返回调用者。

否则，有必要处理 rcu_node 分级体系中的下一级。为了准备对下一级的处理，第 16 行将掩码设置为当前 rcu_node 结构的上一级。第 17 行检查当前 rcu_node 结构是否存在父结构，如果不存在，则第 18 行退出循环。另一方面，如果父结构存在，第 20 行释放当前 rcu_node 结构的锁，第 21 行将 rnp 设置为它的父，第 22 行获得父的锁。然后继续从第 7 行开始执行。

如果第 18 行退出循环，我们就知道当前 grace period 已经结束, 因为只有所有 CPUs 已经经历过静止状态，才能清除根 rcu_node 的位。在这种情况下，第 24 行更新 rcu_state 结构的 ->completed 字段，以匹配最近结束的 grace period 编号，这表示 grace period 事实上已经结束。第 24 行调用 rcu_process_gp_end() 以提前运行 CPU 的 RCU 回调，最后，第 26 行调用 rcu_start_gp()，其目的是启动一个新的 grace period，任何当前运行 CPU 上余下的回调需要这样一个 grace period。

```
1 static void rcu_do_batch(struct rcu_data *rdp)
2 {
3 unsigned long flags;
4 struct rcu_head *next, *list, **tail;
5 int count;
6
7 if (!cpu_has_callbacks_ready_to_invoke(rdp))
8 return;
9 local_irq_save(flags);
10 list = rdp->nxtlist;
11 rdp->nxtlist = *rdp->nxttail[RCU_DONE_TAIL];
12 *rdp->nxttail[RCU_DONE_TAIL] = NULL;
13 tail = rdp->nxttail[RCU_DONE_TAIL];
14 for (count = RCU_NEXT_SIZE - 1; count >= 0; count--)
15 if (rdp->nxttail[count] ==
16 rdp->nxttail[RCU_DONE_TAIL])
17 rdp->nxttail[count] = &rdp->nxtlist;
18 local_irq_restore(flags);
19 count = 0;
```

```
20 while (list) {
21 next = list->next;
22 prefetch(next);
23 list->func(list);
24 list = next;
25 if (++count >= rdp->blimit)
26 break;
27 }
28 local_irq_save(flags);
29 rdp->qlen -= count;
30 if (list != NULL) {
31 *tail = rdp->nxtlist;
32 rdp->nxtlist = list;
33 for (count = 0; count < RCU_NEXT_SIZE; count++)
34 if (&rdp->nxtlist == rdp->nxttail[count])
35 rdp->nxttail[count] = tail;
36 else
37 break;
38 }
39 if (rdp->blimit == LONG_MAX && rdp->qlen <= qlowmark)
40 rdp->blimit = blimit;
41 local_irq_restore(flags);
42 if (cpu_has_callbacks_ready_to_invoke(rdp))
43 raise_softirq(RCU_SOFTIRQ);
44 }
```

图 D.43: Code for `rcu_do_batch()`

图 D.43 展示了 rcu_do_batch()，它在 grace periods 结束时调用 RCU 回调。仅仅是正在运行的 CPU 才会调用它们的回调 －其实 CPUs 必须调用它们自己的回调。

问题 D.49： dynticks-idle 或者离线 CPU 上的 RCU 回调是如何得到调用的？

第 7 行调用 cpu_has_callbacks_ready_to_invoke()以确定当前 CPU 是否有 RCU 回调需要完成，如果不是，则第 8 行返回。第 9、18 行禁止并重新打开中断。第 11-13 行从->nxtlist 中移除准备调用的回调，第 14-17 行调整相应的尾指针。

问题 D.50：为什么第 14-17 行需要调整它们的尾指针？

第 19 行初始化局部变量 count 为 0，准备为了实际调用的回调函数进行计数。第 20-27 行的循环体调用一个回调并对其进行计数，如果调用了太多的回调，那么在第 25-26 行退出循环。剩余的函数由于这个限制而不能被调用。

第 28、41 行禁止并重新打开中断。第 29 行更新->qlen 字段，它为本 CPU 维护一个 RCU 回调函数总数计数。第 30 行检查是否有因为调用限制的原因而不能调用的回调。如果有这样的回调，重新调整尾指针。第 39-40 行恢复限制数。如果有剩余的回调，则第 42-43 行触发 RCU 处理。

D.3.7 Dyntick-Idle 函数

本节中的函数仅仅在 CONFIG_NO_HZ 在内核中构建时，才会被定义，考虑到某些情况，在没有定义这个内核参数时，将扩展出一些空操作的版本。这些函数控制 RCU 是否应当关注某个特定的 CPU。处于 dynticks-idle 模式的 CPU 被忽略，但是仅仅是没有在中断和 NMI 处理函数中才忽略它们。本节中的函数向 RCU 传递 CPU 状态。

这一组函数用于可抢占 RCU 时十分简单，参见 E.7 节对较早前的更复杂模式的描述。Manfred Spraul 在他的基于状态 RCU 补丁中给出了这个简化的接口 [Spr08b,Spr08a].

D.3.7.1 节描述了从进程上下文进入、退出 dynticks-idle 模式，D.3.7.2 节描述从 dynticks-idle 模式进入 NMI 处理函数，D.3.7.3 节包含从 dynticks-idle 模式进入中断处理，D.3.7.4 节描述了检查其他 CPU 是否处于 dynticks-idle 模式的函数。

D.3.7.1 进入和退出 Dyntick-Idle 模式

```
1 void rcu_enter_nohz(void)
2 {
3 unsigned long flags;
4 struct rcu_dynticks *rdtp;
5
6 smp_mb();
7 local_irq_save(flags);
8 rdtp = &__get_cpu_var(rcu_dynticks);
9 rdtp->dynticks++;
10 rdtp->dynticks_nesting--;
11 local_irq_restore(flags);
12 }
13
14 void rcu_exit_nohz(void)
15 {
16 unsigned long flags;
17 struct rcu_dynticks *rdtp;
```

```
18
19 local_irq_save(flags);
20 rdtp = &__get_cpu_var(rcu_dynticks);
21 rdtp->dynticks++;
22 rdtp->dynticks_nesting++;
23 local_irq_restore(flags);
24 smp_mb();
25 }
```

图 D.44: Entering and Exiting Dyntick-Idle Mode

图 D.44 展示了 rcu_enter_nohz()和 rcu_exit_nohz()函数，它允许调度器进入、退出 dynticks-idle 模式。因此，调用 rcu_enter_nohz() 后,，RCU 将忽略它，直到下次调用 rcu_exit_nohz()，或者下一个中断，或者下一个 NMI。

rcu_enter_nohz()的第 6 行执行一个内存屏障，确保前面的 RCU 读临界区看起来在随后的代码之前发生。随后的代码通知 RCU 忽略这个 CPU。第 7、11 行禁止并恢复中断，以避免状态改变冲突。第 98 行获得当前运行 CPU 的 rcu_dynticks 结构，第 9 行递增 ->dynticks 字段 (这样它必然是偶数，表示这个 CPU 将被忽略)，最后第 10 行递减->dynticks_nesting 字段 (现在它必然是 0，表示不必关注这个 CPU)。

rcu_exit_nohz()的第 19 、23 行禁止并重新打开中断，也是为了避免冲突。第 20 行得到这个 CPU 的 rcu_dynticks 结构指针，第 21 行递增 ->dynticks 字段 (现在它必然是奇数，表示 RCU 必须再次关注这个 CPU)，第 22 行递增 ->dynticks_nesting 字段 (这样它必然是 1，表示有一个原因需要关注这个 CPU)。

D.3.7.2 从 Dyntick-Idle 模式进入 NM

```
1 void rcu_nmi_enter(void)
2 {
3 struct rcu_dynticks *rdtp;
4
5 rdtp = &__get_cpu_var(rcu_dynticks);
6 if (rdtp->dynticks & 0x1)
7 return;
8 rdtp->dynticks_nmi++;
9 smp_mb();
10 }
11
12 void rcu_nmi_exit(void)
13 {
14 struct rcu_dynticks *rdtp;
```

```
15
16 rdtp = &__get_cpu_var(rcu_dynticks);
17 if (rdtp->dynticks & 0x1)
18 return;
19 smp_mb();
20 rdtp->dynticks_nmi++;
```

图 D.45: NMIs from Dyntick-Idle Mode

图 D.45 显示了 rcu_nmi_enter() 和 rcu_nmi_exit(),它们处理进入、退出 NMI。退出 dyntick-idle 模式进入 NMI 处理需要特别留意，换句话说， RCU 必须关注处于 NMI 处理函数中的 CPU 要求回到 dyntick-idle 模式的 CPU。因为 NMI 处理程序可能包含 RCU 读临界区。这有点令人迷惑，所以必须小心。

rcu_nmi_enter()的第 5 行获得这个 CPU 的 rcu_dynticks 结构指针，第 6 行检查该 CPU 是否已经处于 RCU 中，如果是，则在第 7 行直接返回。否则，第 8 行递增->dynticks_nmi 字段，这样它应该是一个奇数值。最后，第 19 行执行一个内存屏障，以确保前面对->dynticks_nmi 的递增会在随后的 RCU 读临界区之前发生。

rcu_nmi_exit()的第 16 行再次获得 CPU 的 rcu_dynticks 结构指针，第 17 行检查 RCU 是否已经关注该 CPU，如果不是，则在第 18 行返回。第 19 行执行一个内存屏障，确保 NMI 处理函数中的任何 RCU 读临界区在第 20 行递增->dynticks_nmi 字段前，被其他 CPU 看见。这个字段的新值必然是偶数。

问题 D.51：但是图 D.45 中的代码如何处理嵌套 NMIs?

D.3.7.3 从 Dyntick-Idle 模式进入中断

```
1 void rcu_irq_enter(void)
2 {
3 struct rcu_dynticks *rdtp;
4
5 rdtp = &__get_cpu_var(rcu_dynticks);
6 if (rdtp->dynticks_nesting++)
7 return;
8 rdtp->dynticks++;
9 smp_mb();
10 }
11
12 void rcu_irq_exit(void)
13 {
14 struct rcu_dynticks *rdtp;
```

```
15
16 rdtp = &__get_cpu_var(rcu_dynticks);
17 if (--rdtp->dynticks_nesting)
18 return;
19 smp_mb();
20 rdtp->dynticks++;
21 if (__get_cpu_var(rcu_data).nxtlist ||
22 __get_cpu_var(rcu_bh_data).nxtlist)
23 set_need_resched();
24 }
```

图 D.46: Interrupts from Dyntick-Idle Mode

图 D.46 展示了 rcu_irq_enter() 和 rcu_irq_exit(),它们处理进入、退出中断。

rcu_irq_enter()的第 5 行也获得当前 CPU 的 rcu_dynticks 结构指针。第 6 行递增->dynticks_nesting 字段，如果旧值是非 0 值(换句话说，RCU 已经在关注这个 CPU 了)，第 7 行则退出。否则，第 8 行递增 ->dynticks 字段，那么它现在必然是一个奇数值。第 9 行执行一个内存屏障，这样，这次递增会在中断处理函数中的所有 RCU 读临界区之前被其他 CPU 看到。

rcu_irq_exit()的第 16 行获得当前 CPU 的 rcu_dynticks 结构的引用。17 行递减 ->dynticks_nesting 字段，如果结果非 0 (换句话说, 虽然离开了中断处理函数，但是 RCU 仍然在关注这个 CPU)，那么第 18 行退出。否则，第 19 行执行一个内存屏障，这样在中断处理函数中的读临界区将在第 20 行的递增之前被其他 CPU 看到。第 20 行递增 ->dynticks 字段 (现在必然有一个偶数值)。第 21、22 行检查中断处理函数是否触发了任何 ``rcu'' 或者 ``rcu_bh'' 回调，如果是，则第 23 行强制 CPU 进行调度,它有一个附加的效果是强制 CPU 退出 dynticks-idle 模式，它允许 RCU 处理这些回调函数需要的 grace period。

D.3.7.4 检查 Dyntick-Idle 模式

dyntick_save_progress_counter() 和 rcu_implicit_dynticks_qs() 函数用来检查一个 CPU 是否处于 dynticks-idle 模式。dyntick_save_progress_counter() 函数首先被调用，如果 CPU 当前处于 dynticks-idle 模式，则返回非 0 值。如果 CPU 没有处于 dynticks-idle 模式，例如，由于当前正在处理中断或者 NMI，那么在随后一些 jiffies 中，将调用 rcu_implicit_dynticks_qs()函数。这个函数查看当前状态，如果 CPU 在 dynticks-idle 状态则返回非 0。rcu_implicit_dynticks_qs() 函数可能被反复调用，直到它返回真。

```
1 static int
2 dyntick_save_progress_counter(struct rcu_data *rdp)
```

```
3 {
4 int ret;
5 int snap;
6 int snap_nmi;
7
8 snap = rdp->dynticks->dynticks;
9 snap_nmi = rdp->dynticks->dynticks_nmi;
10 smp_mb();
11 rdp->dynticks_snap = snap;
12 rdp->dynticks_nmi_snap = snap_nmi;
13 ret = ((snap & 0x1) == 0) && ((snap_nmi & 0x1) == 0);
14 if (ret)
15 rdp->dynticks_fqs++;
16 return ret;
17 }
```

图 D.47: Code for dyntick_save_progress_counter()

图 D.47 展示了 dyntick_save_progress_counter()的代码，传递了一个特定 CPU 的一对 rcu_state 结构的 rcu_data。第 8、9 行获得 CPU 的 rcu_dynticks 的->dynticks 和->dynticks_nmi 字段快照，然后在第 10 行执行一个内存屏障，以确保这两个快照在随后依赖于这两个值的处理过程之前被其他 CPU 看见。这个内存屏障与 rcu_enter_nohz()，rcu_exit_nohz()，rcu_nmi_enter()，rcu_nmi_exit()，rcu_irq_enter()和 rcu_irq_exit()中的内存屏障是成对的。第 11、12 行存储这两个快照，这样它们可以被随后调用的 rcu_implicit_dynticks_qs()访问。第 13 行检查两个快照是否都是偶数，这表示该 CPU 既不处于 non-idle 进程，中断处理，也不处于 NMI 处理中。如果是这样，第 14、15 行递增统计计数->dynticks_fqs，它仅仅用于调试目的。不管怎样，第 16 行返回的值表示 CPU 是否处于 dynticks-idle 模式。

问题 D.52：为什么在第 8、9 行之间没有内存屏障?

```
1 static int
2 rcu_implicit_dynticks_qs(struct rcu_data *rdp)
3 {
4 long curr;
5 long curr_nmi;
6 long snap;
7 long snap_nmi;
8
9 curr = rdp->dynticks->dynticks;
10 snap = rdp->dynticks_snap;
11 curr_nmi = rdp->dynticks->dynticks_nmi;
12 snap_nmi = rdp->dynticks_nmi_snap;
```

```
13 smp_mb();
14 if ((curr != snap || (curr & 0x1) == 0) &&
15 (curr_nmi != snap_nmi || (curr_nmi & 0x1) == 0)) {
16 rdp->dynticks_fqs++;
17 return 1;
18 }
19 return rcu_implicit_offline_qs(rdp);
20 }
```

图 D.48: Code for rcu_implicit_dynticks_qs()

图 D.48 展示了 rcu_implicit_dynticks_qs()的代码。第 9-12 行获得 CPU 的 rcu_dynticks 结构的->dynticks 和 ->dynticks_nmi 字段的新值，和上一个 dyntick_save_progress_counter()函数一样。第 13 行执行一个内存屏障，以确保其值在后续的 RCU 处理过程之前被其他 CPU 看到。与 dyntick_save_progress_counter()一样，这个内存屏障与 rcu_enter_nohz()，rcu_exit_nohz()，rcu_nmi_enter()，rcu_nmi_exit()，rcu_irq_enter()和 rcu_irq_exit()是配对的。第 14-15 行进行检查，以确保该 CPU 当前要么处于 dynticks-idle 模式 ((curr & 0x1) == 0 and (curr_nmi & 0x1) == 0) ，要么自调用 dyntick_save_progress_counter()以来，已经经历过一次静止状态 (curr != snap and curr_nmi != snap_nmi)。如果是这样，第 16 行递增 ->dynticks_fqs 统计计数 (也是用于调试目的) ，第 17 行返回非 0 值表示指定 CPU 已经经历过一次静止状态。否则，第 19 行调用 rcu_implicit_offline_qs() (在 D.3.8 节中描述) 以检查指定 CPU 当前是否已经离线。

D.3.8 强制静止状态

通常，经过静止状态的 CPUs 将适当的记录下来，因此 grace periods 将及时的结束。但是, 下面三个条件中的任意一个都可能阻止 CPUs 经历静止状态：

- ✓ CPU 处于 dyntick-idle 状态，因此在低电源状态睡眠。虽然这样的 CPU 实际上是处于扩展静止状态，不过由于它不能执行指令，因此它不能自行做任何事情。
- ✓ CPU 正在上线过程中，RCU 已经得到通知，知道它已经上线。但是这个 CPU 仍然没有执行代码，也没有在 cpu_online_map 中标记在线标志。当前 grace period 因此会等待它，但是它不能自行经历一次静止状态。
- ✓ CPU 在运行用户态代码，还没有进入调度器。

在每一种情况下，RCU 需要对未响应的 CPU 采取行动。以下节描述这些函数。D.3.8.1 节描述记录 dynticks-idle grace-period 编号的函数，D.3.8.2 节描述检

查离线及久未响应的 CPU 的函数，D.3.8.3 节描述 rcu_process_dyntick()，它扫描久未响应的 CPUs，D.3.8.4 节描述 force_quiescent_state()，驱动对扩展静止状态的检测，以及在久未响应的 CPU 上强制产生一次静止状态。

D.3.8.1 记录 Dynticks-Idle Grace Period

```
1 static void
2 dyntick_record_completed(struct rcu_state *rsp,
3 long comp)
4 {
5 rsp->dynticks_completed = comp;
6 }
7
8 static long
9 dyntick_recall_completed(struct rcu_state *rsp)
10 {
11 return rsp->dynticks_completed;
12 }
```

图 D.49: Recording and Recalling Dynticks-Idle Grace Period

图 D.49 展示了 dyntick_record_completed()和 dyntick_recall_completed()的代码。上面的函数仅仅在 dynticks 打开时才定义(换句话说，CONFIG_NO_HZ 内核参数被选中)，否则它们就是空操作。

第 1-6 行展示了 dyntick_record_completed()，它将 comp 参数的值保存到 rcu_state 结构的 ->dynticks_completed 字段。第 8-12 行展示了 dyntick_recall_completed()，它返回最近调用 dyntick_record_completed() 时，存储在 rcu_state 结构中的值。

D.3.8.2 处理离线及久未响应的 CPU

```
1 static int rcu_implicit_offline_qs(struct rcu_data *rdp)
2 {
3 if (cpu_is_offline(rdp->cpu)) {
4 rdp->offline_fqs++;
5 return 1;
6 }
7 if (rdp->cpu != smp_processor_id())
8 smp_send_reschedule(rdp->cpu);
9 else
10 set_need_resched();
```

```
11 rdp->resched_ipi++;
12 return 0;
13 }
```

图 D.50: Handling Offline and Holdout CPUs

图 D.50 展示了 rcu_implicit_offline_qs()的代码，它检测离线 CPUs，并强制在线但是久未响应的 CPUs 进入一个静止状态。

第 3 行检查特定 CPU 是否离线，如果是，第 4 行递增统计计数->offline_fqs (仅仅用于调试)，第 5 行返回非 0 值表示 CPU 处于扩展静止状态。

否则，CPU 是在线的，不处于 dynticks-idle 模式，并且还没有经历一次静止状态。第 7 行检查久未响应 CPU 是否为当前 cpu，如果不是，第 8 行发送重新调度 IPI。否则，第 10 行设置 TIF_NEED_RESCHED 标志，强制当前 CPU 进入调度。任一情况下，CPU 将很快进入静止状态。第 11 行递增统计计数 resched_ipi，它也用于调度目的。最后，第 12 行返回 0，表示久未响应 CPU 仍然没有经历静止状态。

D.3.8.3 扫描久未响应 CPUs

```
1 static int
2 rcu_process_dyntick(struct rcu_state *rsp,
3 long lastcomp,
4 int (*f)(struct rcu_data *))
5 {
6 unsigned long bit;
7 int cpu;
8 unsigned long flags;
9 unsigned long mask;
10 struct rcu_node *rnp_cur;
11 struct rcu_node *rnp_end;
12
13 rnp_cur = rsp->level[NUM_RCU_LVLS - 1];
14 rnp_end = &rsp->node[NUM_RCU_NODES];
15 for (; rnp_cur < rnp_end; rnp_cur++) {
16 mask = 0;
17 spin_lock_irqsave(&rnp_cur->lock, flags);
18 if (rsp->completed != lastcomp) {
19 spin_unlock_irqrestore(&rnp_cur->lock, flags);
20 return 1;
21 }
22 if (rnp_cur->qsmask == 0) {
23 spin_unlock_irqrestore(&rnp_cur->lock, flags);
```

```
24 continue;
25 }
26 cpu = rnp_cur->grplo;
27 bit = 1;
28 for (; cpu <= rnp_cur->grphi; cpu++, bit <<= 1) {
29 if ((rnp_cur->qsmask & bit) != 0 &&
30 f(rsp->rda[cpu]))
31 mask |= bit;
32 }
33 if (mask != 0 && rsp->completed == lastcomp) {
34 cpu_quiet_msk(mask, rsp, rnp_cur, flags);
35 continue;
36 }
37 spin_unlock_irqrestore(&rnp_cur->lock, flags);
38 }
39 return 0;
40 }
```

图 D.51: Scanning for Holdout CPUs

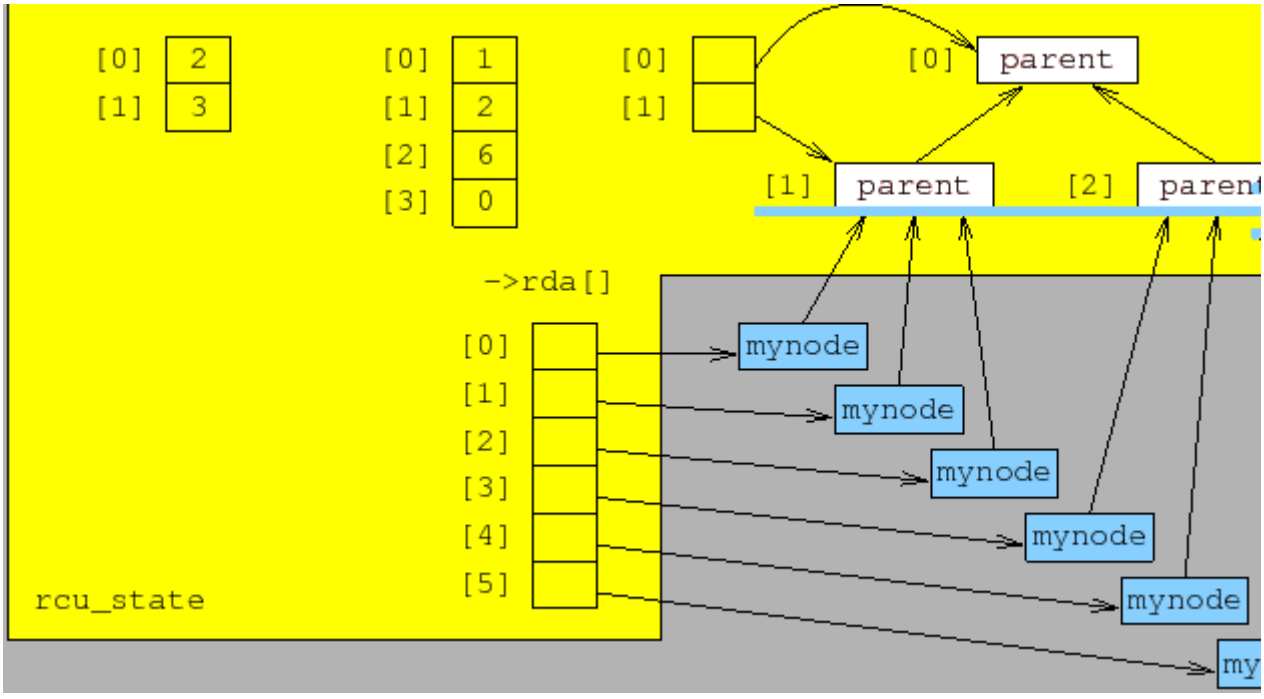


图 D.52: Scanning Leaf `rcu_node` Structures

图 D.51 展示了 rcu_process_dyntick()的代码，它扫描叶子 rcu_node ，以搜索久未响应 CPUs，由图 D.52 中蓝色箭头标识。它为每一个 CPU 的 rcu_data 结构调用传入参数 f ，如果由 lastcomp 参数指定的 grace period 已经结束，就返回非 0。

第 13、14 行获得第一个和最后一个 rcu_node 结构的引用。第 15-38 行的循

环体处理其中一个叶子 rcu_node 结构。

第 16 行设置局部变量 mask 为 0。这个变量用于保存已经处于扩展静止状态的 CPU。第 17 行获得当前 rcu_node 结构的锁，第 18 行检查当前 grace period 是否已经完成。如果是，第 19 行释放锁，并在第 20 行返回非 0。否则，第 22 行检查与这个 rcu_node 相关联的久未响应的 CPU。如果没有这样的 CPU， 第 23 行释放锁，并且在第 24 行重启循环，并从下一个 rcu_node 结构开始。

如果这个 rcu_node 结构至少有一个久未响应的 CPU，则开始第 26 行的执行。第 26、27 行设置局部变量 cpu 和 bit 为与这个 rcu_node 相关的最小的值。第 28-32 行的循环体检查与当前 rcu_node 结构相关的一个 CPUs。第 29 行检查该 CPU 是否仍然处于未响应状态或者它是否经历了一次静止状态。如果仍然未响应，第 30 行调用指定的函数(dyntick_save_progress_counter()或者 rcu_implicit_dynticks_qs()，由调用者指定)，如果函数返回非 0 (表示当前 CPU 处于扩展静止状态)，那么第 31 行设置当前 CPU 的位到 mask 中。

第 33 行检查是否有 CPU 处于扩展静止状态，如果是，第 34 行调用 cpu_quiet_msk() 报告 grace period 不必再等待这些 CPUs，第 35 行重新开始循环。(注意 cpu_quiet_msk() 释放了当前 rcu_node 的锁, 因此可能结束当前 current grace period.) 否则，如果所有未响应 CPUs 仍然没有响应，第 37 行释放当前 rcu_node 的锁。

一旦所有叶子 rcu_node 已经处理，就退出循环。第 39 行返回 0 表示当前 grace period 仍然需要强制静止状态。

D.3.8.4 force_quiescent_state()代码

```
1 static void
2 force_quiescent_state(struct rcu_state *rsp, int relaxed)
3 {
4 unsigned long flags;
5 long lastcomp;
6 struct rcu_data *rdp = rsp->rda[smp_processor_id()];
7 struct rcu_node *rnp = rcu_get_root(rsp);
8 u8 signaled;
9
10 if (ACCESS_ONCE(rsp->completed) ==
11 ACCESS_ONCE(rsp->gpnum))
12 return;
13 if (!spin_trylock_irqsave(&rsp->fqslock, flags)) {
14 rsp->n_force_qs_lh++;
15 return;
16 }
```

```
17 if (relaxed &&
18 (long)(rsp->jiffies_force_qs - jiffies) >= 0 &&
19 (rdp->n_rcu_pending_force_qs -
20 rdp->n_rcu_pending) >= 0)
21 goto unlock_ret;
22 rsp->n_force_qs++;
23 spin_lock(&rnp->lock);
24 lastcomp = rsp->completed;
25 signaled = rsp->signaled;
26 rsp->jiffies_force_qs =
27 jiffies + RCU_JIFFIES_TILL_FORCE_QS;
28 rdp->n_rcu_pending_force_qs =
29 rdp->n_rcu_pending +
30 RCU_JIFFIES_TILL_FORCE_QS;
31 if (lastcomp == rsp->gpnum) {
32 rsp->n_force_qs_ngp++;
33 spin_unlock(&rnp->lock);
34 goto unlock_ret;
35 }
36 spin_unlock(&rnp->lock);
37 switch (signaled) {
38 case RCU_GP_INIT:
39 break;
40 case RCU_SAVE_DYNTICK:
41 if (RCU_SIGNAL_INIT != RCU_SAVE_DYNTICK)
42 break;
43 if (rcu_process_dyntick(rsp, lastcomp,
44 dyntick_save_progress_counter))
45 goto unlock_ret;
46 spin_lock(&rnp->lock);
47 if (lastcomp == rsp->completed) {
48 rsp->signaled = RCU_FORCE_QS;
49 dyntick_record_completed(rsp, lastcomp);
50 }
51 spin_unlock(&rnp->lock);
52 break;
53 case RCU_FORCE_QS:
54 if (rcu_process_dyntick(rsp,
55 dyntick_recall_completed(rsp),
56 rcu_implicit_dynticks_qs))
57 goto unlock_ret;
58 break;
59 }
60 unlock_ret:
```

```
61 spin_unlock_irqrestore(&rsp->fqslock, flags);
62 }
```

图 D.53: force_quiescent_state() Code

图 D.53 展示了在配置 CONFIG_SMP 时，force_quiescent_state() 的代码。当 RCU 认为需要通过强制 CPUs 通过静止状态以加快当前 grace period 时，调用此函数。当以下情况发生时，RCU 认为需要这样做：

当前 grace period 已经过去超过三个 jiffies (或者由编译时指定的值 RCU_JIFFIES_TILL_FORCE_QS)，或者 CPU 通过 call_rcu() 或者 call_rcu_bh() 将一个 RCU 回调入队时，发现超过 10,000 回调已经在队列中 (或者是启动时指定的参数 qhimark)。

第 10-12 行是否有一个 grace period 正在处理，如果没有则退出。第 13-16 行尝试获得->fqslock，它防止与其他尝试加快 grace period 操作之间的冲突。当锁已经获得时，递增->n_force_qs_lh 计数，当 CONFIG_RCU_TRACE 打开时，可以通过 fqlh= field 查看。第 17-21 行是否真的有必要加快当前 grace period。换句话说，当前 CPU 是否有 10,000 RCU 回调处于等待状态，或者自当前 grace period 启动以来，或者至上次试图加快当前 grace period 以来，是否至少经过三个 jiffies。第 22 行对尝试加快 grace periods 进行计数。

第 23-36 行在 rcu_node 结构的锁保护下执行，其目的是防止在我们试图加快它时，当前 grace periods 已经结束。第 24、25 行对 ->completed 和 ->signaled 字段进行快照，第 26-30 行设置后续的 force_quiescent_state()允许执行的最早的时间。第 31-35 行检查在我们获得 rcu_node 锁期间，grace period 是否已经结束，如果是这样，则释放锁并返回。

第 37-59 行驱动 force_quiescent_state() 状态机。如果 grace period 仍然在初始化，第 41、42 行简单的返回。这允许随后的时间再次调用 force_quiescent_state()，假设此时初始化已经完成。如果打开了 dynticks (通过 CONFIG_NO_HZ 内核参数，第一次调用 force_quiescent_state()将执行第 40-52 行，第二次及随后的调用将执行第 53-59 行。另一方面，如果 dynticks 没有打开，那么每一次调用都将执行第 53-59 行。

第 40-52 行的目的是记录所有没有经历静止状态的 CPU 的当前 dynticks-idle 状态。并且记录当前在 dynticks-idle 状态(但是当前没有在中断和 NMI 处理函数中)的 CPU 的静止状态。第 41-42 行 通知 gcc ，当前分支语句在没有配置 CONFIG_NO_HZ 时，是无用代码。第 43-45 行调用 rcu_process_dyntick()，其目的是为每一个没有经历静止状态的 CPU 调用 dyntick_save_progress_counter()，如果 grace period 结束，则退出 force_quiescent_state() (可能是由于已经发现所有没有经历静止状态的 CPU 已经处于 dyntick-idle 模式)。第 46、51 行获取并释

放根 rcu_node 结构的锁。第 47 行检查当前 grace period 是否仍然需要强制其他 CPU 进入静止状态。如果是，第 48 行将状态机推进到 RCU_FORCE_QS 状态，第 49 行保存当前 grace-period 编号，这对下一个 force_quiescent_state()是有用的。

正如较早前所述，在配置 CONFIG_NO_HZ 时，第 53-58 行处理第二次及后面对 force_quiescent_state()的调用，在没有配置 CONFIG_NO_HZ 时，它处理每一次对 force_quiescent_state()的调用。第 54 、58 行调用 rcu_process_dyntick()，它遍历所有仍然没有经历静止状态的 CPUs，在其上调用 rcu_implicit_dynticks_qs()，它依次检查这些 CPU 是否经历 dyntick-idle 状态 (如果 CONFIG_NO_HZ 被打开)，检查我们是否正在等待离线 CPU，最后发送一个重新调度 IPI 给余下的 CPUs。

D.3.9 CPU-延迟检测

当配置了 CONFIG_RCU_CPU_STALL_DETECTOR 内核参数时，RCU 进行延迟 CPU 检查。``延迟 CPUs" 是那些关抢占后自旋的 CPUs，它们降低了系统响应时间。这些检查是通过 record_gp_stall_check_time(), check_cpu_stall(), print_cpu_stall(), 和 print_other_cpu_stall() 函数来实现的，每一个函数将在下面描述。在没有配置 CONFIG_RCU_CPU_STALL_DETECTOR 时，这些函数是空操作。

```
1 static void
2 record_gp_stall_check_time(struct rcu_state *rsp)
3 {
4 rsp->gp_start = jiffies;
5 rsp->jiffies_stall =
6 jiffies + RCU_SECONDS_TILL_STALL_CHECK;
7 }
```

图 D.54: record_gp_stall_check_time() Code

图 D.54 展示了 record_gp_stall_check_time()函数的代码。第 4 行记录当前时间（grace period 启动时间，以 jiffies 计算），第 5-6 行记录应当检查 CPU 延迟的时间，此时 grace period 应该已经运行一大段时间。

```
1 static void
2 check_cpu_stall(struct rcu_state *rsp,
3 struct rcu_data *rdp)
4 {
5 long delta;
6 struct rcu_node *rnp;
7
```

```
8 delta = jiffies - rsp->jiffies_stall;
9 rnp = rdp->mynode;
10 if ((rnp->qsmask & rdp->grpmask) && delta >= 0) {
11 print_cpu_stall(rsp);
12 } else if (rsp->gpnum != rsp->completed &&
13 delta >= RCU_STALL_RAT_DELAY) {
14 print_other_cpu_stall(rsp);
15 }
16 }
```

图 D.55: check_cpu_stall() Code

图 D.55 展示了 check_cpu_stall 的代码，它检查 grace period 是否已经启动太长时间，如果是，则调用 print_cpu_stall() 或者 print_other_cpu_stall()来打印一个 CPU 延迟警告信息。

第 8 行计算自延迟警告信息应当被打印以来，经过的 jiffies 数，如果还没有到打印警告信息的时间，那么其值应当为负。第 9 行获得当前 CPU 相应的叶子 rcu_node 结构的指针，第 10 行检查当前 CPU 是否已经经历过一次静止状态或者 grace period 是否超时太久(换句话说,当前 CPU 是否延迟)，如果是，则第 11 行调用 print_cpu_stall()。

否则，第 12-13 行检查当前 grace period 是否仍然有效，并且是否已经延迟了 RCU_STALL_RAT_DELAY 这么久的时间，如果是，则在第 14 行调用 print_other_cpu_stall()。

问题 D.53：为什么在第 12-13 行要等待额外的 jiffies?

```
1 static void print_cpu_stall(struct rcu_state *rsp)
2 {
3 unsigned long flags;
4 struct rcu_node *rnp = rcu_get_root(rsp);
5
6 printk(KERN_ERR
7 "INFO: RCU detected CPU %d stall "
8 "(t=%lu jiffies)\n",
9 smp_processor_id(),
10 jiffies - rsp->gp_start);
11 dump_stack();
12 spin_lock_irqsave(&rnp->lock, flags);
13 if ((long)(jiffies - rsp->jiffies_stall) >= 0)
14 rsp->jiffies_stall =
15 jiffies + RCU_SECONDS_TILL_STALL_RECHECK;
16 spin_unlock_irqrestore(&rnp->lock, flags);
17 set_need_resched();
18 }
```

图 D.56: print_cpu_stall() Code

图 D.56 展示了 print_cpu_stall()的代码。

第 6-11 打印控制台消息并打印当前 CPU 的堆栈，而第 12-17 行计算下次 CPU 延迟警告信息的时间。

问题 D.54：在打印延迟警告信息前，要防止 grace period 结束时发生什么事情 ？

```
1 static void print_other_cpu_stall(struct rcu_state *rsp)
2 {
3 int cpu;
4 long delta;
5 unsigned long flags;
6 struct rcu_node *rnp = rcu_get_root(rsp);
7 struct rcu_node *rnp_cur;
8 struct rcu_node *rnp_end;
9
10 rnp_cur = rsp->level[NUM_RCU_LVLS - 1];
11 rnp_end = &rsp->node[NUM_RCU_NODES];
12 spin_lock_irqsave(&rnp->lock, flags);
13 delta = jiffies - rsp->jiffies_stall;
14 if (delta < RCU_STALL_RAT_DELAY ||
15 rsp->gpnum == rsp->completed) {
16 spin_unlock_irqrestore(&rnp->lock, flags);
17 return;
18 }
19 rsp->jiffies_stall = jiffies +
20 RCU_SECONDS_TILL_STALL_RECHECK;
21 spin_unlock_irqrestore(&rnp->lock, flags);
22 printk(KERN_ERR "INFO: RCU detected CPU stalls:");
23 for (; rnp_cur < rnp_end; rnp_cur++) {
24 if (rnp_cur->qsmask == 0)
25 continue;
26 cpu = 0;
27 for (; cpu <= rnp_cur->grphi - rnp_cur->grplo; cpu++)
28 if (rnp_cur->qsmask & (1UL << cpu))
29 printk(" %d", rnp_cur->grplo + cpu);
30 }
31 printk(" (detected by %d, t=%ld jiffies)\n",
32 smp_processor_id(),
33 (long)(jiffies - rsp->gp_start));
34 force_quiescent_state(rsp, 0);
35 }
```

图 D.57: print_other_cpu_stall() Code

图 D.57 展示了 print_other_cpu_stall()的代码,它打印除当前 CPUs 外的其他 CPUs 的延迟警告信息。

第 10、11 行获得第一个叶子 rcu_node 结构的引用，以及最后一个叶子 rcu_node 结构的引用。第 12 行获得根 rcu_node 结构的锁，并且禁止中断。第 13 行计算 CPU 延迟警告信息已经过去多久 (如果还没有打印过则时间为负),第 14、15 行检查 CPU 延迟警告是否已经过去并且 grace period 还没有结束，如果是这样，则第 16 行释放锁并在第 17 行返回。

问题 D.55：为什么 print_other_cpu_stall() 需要检查 grace period 是否结束?

否则，第 19、20 行计算下次打印警告信息的时间，第 21 行释放锁并重新打开中断。第 23-33 行打印延迟 CPUs 列表，最后，第 34 行调用 force_quiescent_state()，来推动延迟 CPU 经历一次静止状态。

D.3.10 可能的缺陷及变更

分级 RCU 最大的问题,可能是调用 force_quiescent_state() 要遍历所有 CPUs 的 rcu_data 结构。在一个有上千 CPUs 的系统中，这可能引起较大的调度延迟，这个扫描过程是在关中断情况下进行的。

这会变成一个实实在在的问题,一个修复方法是分离 force_quiescent_state()。这将大大减少调度延迟.

此时, 那些高度关注调度延迟的系统中，可以限制 CPUs 的数目或者使用可抢占 RCU。

D.4 可抢占 RCU

可抢占 RCU 是不寻常的，它允许读临界区被抢占，并可以阻塞以等待锁。但是，它不处理通常的阻塞(例如通过 wait_event() 原语进行阻塞)：如果你需要这一点，应当使用 SRCU，在附录中描述。与 SRCU 相对的，可抢占 RCU 仅仅允许遵从优先级继承的阻塞原语，在没有配置 CONFIG_PREEMPT 的内核中，不能阻塞。在 RCU 读临界区获得阻塞锁及可抢占会损害到 Ingo Molnar 的 -rt 补丁。但是，最初的可抢占 RCU 实现 [McK05c] 有一些限制，包括：

- ✓ 读端原语不能在不可屏障中断和系统管理中断中调用。
- ✓ 读端原语同时使用原子指令及内存屏障，这都会带来额外的负载。

✓ RCU 读临界区不再优先 [McK07d]。

2.6.26 内核中的新的可抢占 RCU 实现去除了这些限制，附录描述其设计 [McK07a]。但是，请注意：在 2 6.32 中，已经将它替换成一个更快更简单的实现了。

问题 D.56：为什么在可抢占 RCU 读临界区中的阻塞原语应当遵从优先级继承？

问题 D.57：Could the prohibition against using primitives that would block in a non-CONFIG_PREEMPT kernel be lifted, and if so, under what conditions? End Quick Quiz

D.4.1 RCU 概念

有一个低级的 RCU 视图，理解并验证一个 RCU 实现就非常简单了。本节给出最基本的 RCU 必须支持的要求概述。更详细的描述，请参见 8.3.1 节。

RCU 实现必须遵从以下规则：如果 RCU 读临界区中的任何语句在一个 grace period 之前，那么 RCU 读临界区中的所有语句都必须在 grace period 结束前完成。

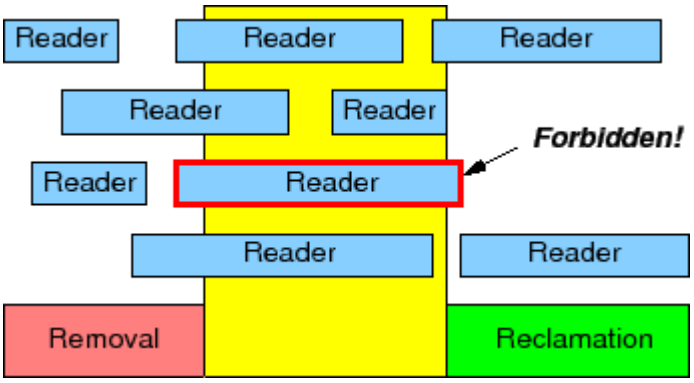


图 D.58: Buggy Grace Period From Broken RCU

以图 D.58 来举例，其时间从左到右推进。红色的 "Removal" 框表示写临界区，它修改 RCU 保护的数据结构。例如， via list_del_rcu()；大的黄色 "Grace Period" 框表示一个 grace period (surprise!) ，它可能通过 synchronize_rcu()调用，绿色"Reclamation" 框表示释放受影响的数据元素，也许是通过 kfree()释放。每一个蓝色"Reader" 框表示一个读临界区，例如，由 rcu_read_lock() 开始并由 rcu_read_unlock()结束的代码段。红色边框 "Reader" 框是一个错误例子：任何这样的 RCU 实现是错误的，它允许一个读临界区完全覆盖一个 grace period，因为这样的话，在读者仍然在使用内存的时候，内存可能已经被释放。

所以说，会有这样的 RCU 出现这样的情况吗?

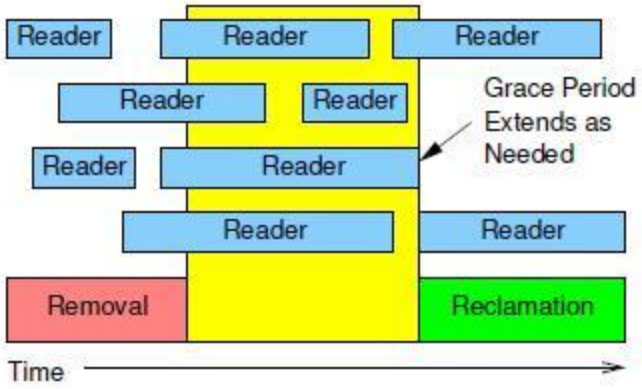


图 D.59: Good Grace Period From Correct RCU

必须扩展 grace period，可能如图 D.59 所示。简单的说，RCU 实现必须确保任何开始于特定 grace period 的 RCU 读临界区必须在 grace period 允许完成前全部结束。这个事实可以允许 RCU 验证专注于如下一点：简单的证明任何开始于 grace period 的 RCU 读临界区必须在 grace period 结束前结束。这需要充足的屏障来防止编译器和 CPU 破坏 RCU 的作用。

D.4.2 可抢占 RCU 算法概述

本节专注于可抢占 RCU 的特定实现。很多其他实现是可能的，它们在其他地方描述 [MSMB06,MS05]。本文档专注于这个实现的通用方法，数据结构，grace-period 状态机，并走查读端原语实现。

D.4.2.1 通用方法

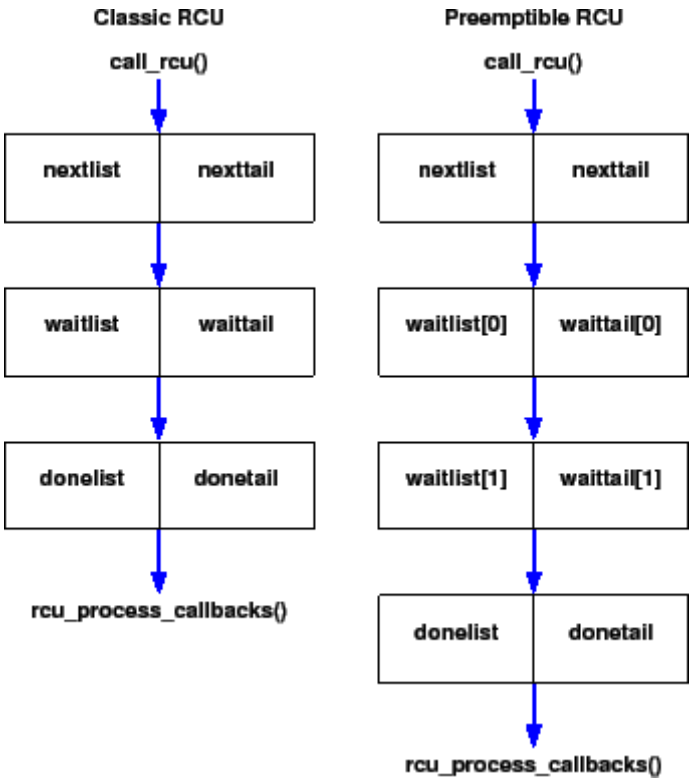


图 D.60: Classic vs. Preemptible RCU Callback Processing

由于可抢占 RCU 在 rcu_read_lock() 和 rcu_read_unlock()中不需要内存屏障，因此需要一个多级 grace-period 检测算法。不是使用一个单一的回调函数等待队列(这在较早前的 RCU 实现中是足够的)，可抢占 RCU 使用一个等待队列数组，因此 RCU 回调依次加入这个数组的每一个元素中。回调流程的差异如图 D.60，在这个可抢占 RCU 实现中，每一个 grace period 有两级等待链表(相对的，2007/09/10 的实时补丁 [McK07c] 使用四级等待链表)。

对于每 grace period 两级链表来说，一对链表形成了一个完成的 grace period。

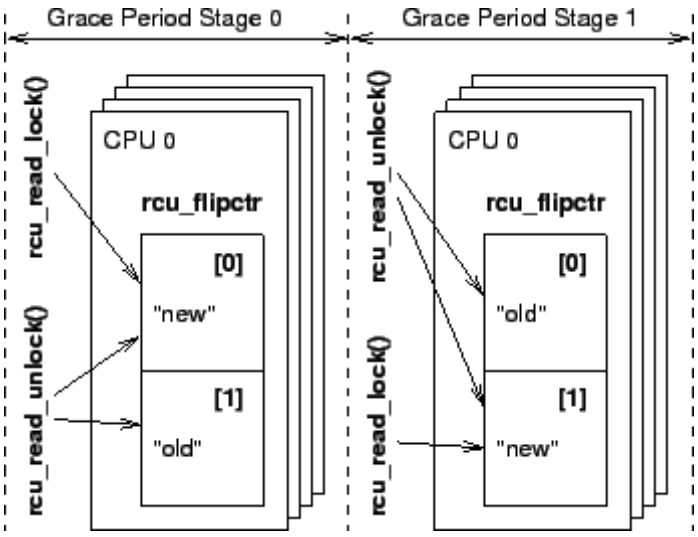


图 D.61: Preemptible RCU Counter Flip Operation

为了确定一个 grace period 何时能够结束，可抢占 RCU 使用了一个每 CPU 的两元素 rcu_flipctr 数组来跟踪正在处理的 RCU 读临界区。其中一个元素跟踪旧的 RCU 读临界区，换句话说，这些临界区开始于当前 grace period 之前。另外的元素跟踪新的 RCU 读临界区，也就是说这些临界区开始于当前 grace period 过程中。数组元素在开始新 grace period 时转换角色，如图 D.61 所示。

由于上图中左边的 rcu_flipctr[0] 跟踪新的 RCU 读临界区，因此它由 rcu_read_lock() 递增，并由 rcu_read_unlock()递减。类似的，rcu_flipctr[1] 跟踪旧 RCU 读临界区因此它由 rcu_read_unlock() 递减，并从不递增。

由于每一个 CPU 的旧 rcu_flipctr[1] 元素从不增加，它们的和最终会等于 0，虽然在 RCU 读临界区中间产生的抢占会导致个别计数值非 0，甚至变为负数。例如，假设某个 CPU 上的任务调用 rcu_read_lock()，被抢占了，它在另外的 CPU 上重新开始，然后调用 rcu_read_unlock()。那么第一个 CPU 的计数将为 +1 ，并且第二个 CPU 的计数值将为-1。但是，它们的和仍然为 0。不管可能产生的抢占，当旧计数值总和变为 0 时，就可以安全的转移到下一个阶段，如上图右边所示。

在第二阶段，每一个 CPU 的 rcu_flipctr 计数数组元素转换角色。rcu_flipctr[0] 计数值现在跟踪旧的 RCU 读临界区，换句话说，跟踪在 grace period 阶段 0 过程中开始的临界区。类似的，rcu_flipctr[1] 现在跟踪新 RCU 读临界区，它们开始于 grace period 阶段 1。因此， rcu_read_lock() 递增 rcu_flipctr[1]，而 rcu_read_unlock() 仍然递减计数。特别的，如果相应的 rcu_read_lock()在 grace-period 阶段 0 开始，那么 rcu_read_unlock() 必须递减 rcu_flipctr[0]，但是如果相应的 rcu_read_lock() 开始于 grace-period 阶段 1，那么 rcu_read_unlock() 必须递减 rcu_flipctr[1]。

关键的一点在于：跟踪旧 RCU 读临界区的 rcu_flipctr 元素必须严格递减。因此，一早它们的和变为 0，就不再变化。

rcu_read_lock() 原语使用当前 grace-period 计数的低位 (rcu_ctrlblk.completed & 0x1) 来对 rcu_flipctr 数组进行索引，并且在任务结构中记录这个索引。相应的 rcu_read_unlock() 使用它来确保递减相应的 rcu_read_lock()计数。当然，如果 RCU 读临界区被抢占，rcu_read_lock() 可能不同 CPU 的计数，而不是递减相应 rcu_read_lock()发生时所在的 CPU 计数。

每一个 CPU 也维护 rcu_flip_flag 和 rcu_mb_flag 每 CPU 变量。rcu_flip_flag 变量用于对每一个 grace period 阶段的开始进行同步：一旦指定 CPU 已经响应了它的 rcu_flip_flag, 它必须不再递增 rcu_flip 数组元素，该元素现在已经用于旧 grace-period 阶段。

rcu_mb_flag 变量用于强制每一个 CPU 在 grace period 阶段结束时执行一个内存屏障。这些内存屏障用于确保在某一个特定 grace period 阶段结束的 RCU 读临界区，其内存访问顺序早于该阶段结束。这个方法在每一个 RCU 读临界区开始和结束时获得一个好处，它们不必真正执行这些代价高昂的内存屏障。rcu_mb_flag 被检查总和为 0 的 CPU 设置为 rcu_mb_needed，但是特定 rcu_mb_flag 仅仅被相应的 CPU 改回为 rcu_mb_done，此时该 CPU 可能正好执行了一个内存屏障。

D.4.2.2 数据结构

本节描述可抢占 RCU 的主要数据结构，包括 rcu_ctrlblk, rcu_data, rcu_flipctr, rcu_try_flip_state, rcu_try_flip_flag 和 rcu_mb_flag。

D.4.2.2.1 `rcu_ctrlblk`

rcu_ctrlblk 结构是全局的，有保护 grace-period 处理的锁 (fliplock) ，还有全局 grace-period 计数 (completed)。Completed 的最后一位用于 rcu_read_lock() 选择对哪一个计数进行递增。

D.4.2.2.2 rcu_data

rcu_data 结构是一个每 CPU 结构，包含以下字段：

lock 保护本结构的其他字段。

completed 用来同步 CPU 局部操作和 rcu_ctrlblk 中的全局计数。

waitlistcount 用来维护非空等待列表计数。这个字段被 rcu_pending()用来确定本 CPU 是否有 RCU 相关的事务需要完成。

nextlist, nextail, waitlist, waittail, donelist, 和 donetail 管理 RCU 回调，这些回调函数正在等待 grace-period 结束。第一个链表有一个尾指针，允许添加。RCU 回调如下图所示进行移动。

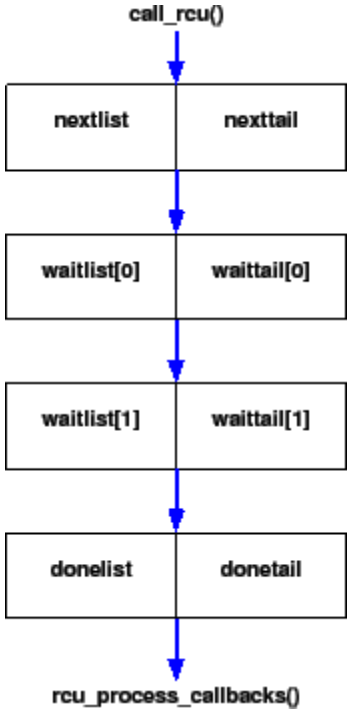


图 D.62: Preemptible RCU Callback Flow

图 D.62 展示了 RCU 回调如何在一个特定的 rcu_data 结构列表中移动。 从 call_rcu() 中被创建，直到 rcu_process_callbacks()被执行.每一个蓝色的箭头表示一次 grace-period 状态机引起的变化，这在随后的章节中进行描述。

D.4.2.2.3 rcu_flipct

正如较早前提示的那样，rcu_flipctr 每 CPU 计数数组包含一对计数器，用来跟踪未完成的 RCU 读临界区。这个数组中任何特定的计数器可能变为负数，例如，当一个任务在 RCU 读临界区中被迁移到其他 CPU 时。但是，计数器的总和在相应的 grace-period 期间至始至终都保持正数，而且在 grace-period 结束时会变为 0。

D.4.2.2.4 rcu_try_flip_state

rcu_try_flip_state 变量跟踪 grace-period 状态机的当前状态，将在下一节中

描述。

D.4.2.2.5 rcu_try_flip_flag

rcu_try_flip_flag 每 CPU 变量告知相应的 CPU，这些 CPU 的 grace-period 计数已经被递增，也记录 CPU 的应答。一旦特定 CPU 已经应答了计数器变化事件，所有后续 rcu_read_lock 操作必须计到新的 grace-period 计数器中。

D.4.2.2.6 rcu_mb_flag

rcu_mb_flag 每 CPU 变量告知相应的 CPU，它必须执行一个内存屏障，这是为了处理 grace-period 状态机。它也记录 CPU 的应答。一旦特定 CPU 已经执行了它的内存屏障，所前之前的 RCU 读临界区的内存操作，将被相应 grace-period 后面的代码所看见。

D.4.2.3 Grace-Period 状态机

本节给出 grace-period 状态机的概况，然后走查相关的代码。

D.4.2.3.1 Grace-Period 状态机概况

相应的状态 (在 rcu_try_flip_state 中记录) 可能有以下的值：

rcu_try_flip_idle_state：在没有 RCU grace-period 期间， grace-period state 状态机是空闲状态。rcu_ctrlblk.completed grace-period 计数器在退出此状态后递增。所有每 CPU rcu_flip_flag 变量被设置为 rcu_flipped。

rcu_try_flip_waitack_state：等待所有 CPU 应答，以表示它们已经看到前一个状态的递增，它们设置 rcu_flip_flag 变量为 rcu_flip_seen。一旦所有 CPUs 都已经应答，我们就知道旧的计数器将不再递增。

rcu_try_flip_waitzero_state：等待旧的计数值总和变为 0。一旦计数值总和为 0，所有每 CPU 变量 rcu_mb_flag 变量被设置为 rcu_mb_needed。

rcu_try_flip_waitmb_state：等待所有 CPUs 执行一个内存屏障指令，这由它们将 rcu_mb_flag 变量设置为 rcu_mb_done 来表示执行了一个内存屏障。一旦所有 CPUs 已经这样做了，所 CPUs 都保证能够看到在相应的 grace-period 开始前启动的 RCU 临界区所做的修改。即使在弱序的机器上也是如此。

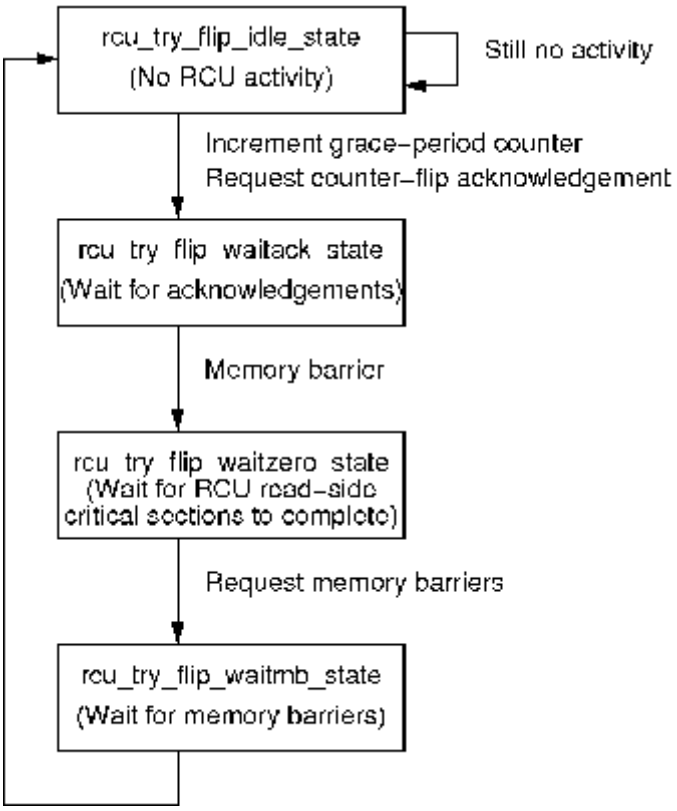


图 D.63: Preemptible RCU State Machine

grace period 状态机周期在这些连续状态之间循环，如下图所示。

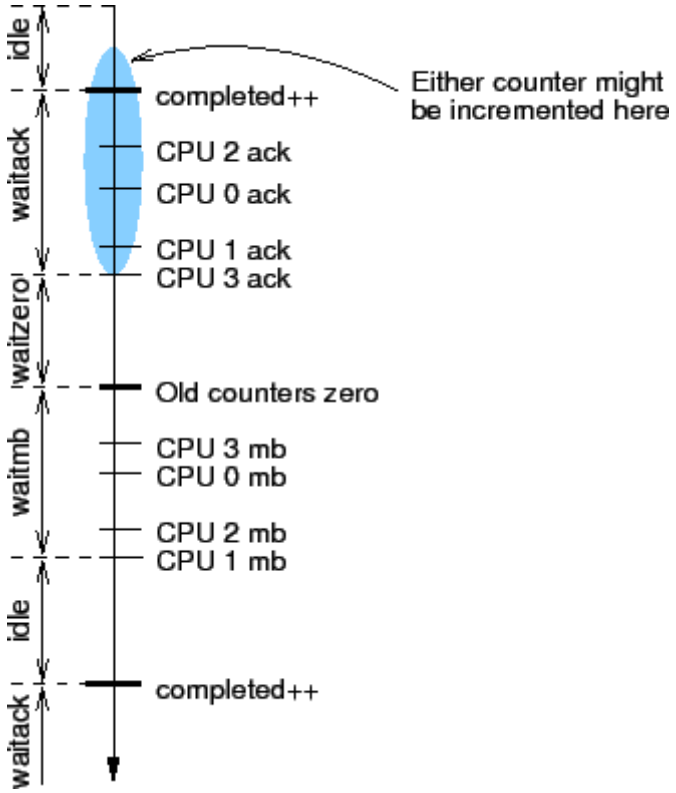


图 D.64: Preemptible RCU State Machine Timeline

图 D.64 展示了状态机是如何运行的。这些状态显示在图的左边，相关的事件显示在时间轴，时间处理是向下面的方向进行的。在后面验证算法时，我们将详细阐述这个图。

在此, 有一些重要的东西需要注意：

对 rcu_ctrlblk.completed 计数器的递增可能被不同 CPU 在不同的时间看到，由绿色的椭圆表示。但是，在一个特定 CPU 已经应答这个递增事件后，它需要使用新的计数器。因此，一旦所有 CPUs 已经进行了应答，旧的计数器仅仅能够被递减。

一个特定 CPU 仅仅在应答计数器递增前移动它的回调链表。

蓝色椭圆表示一个事实：内存重排可能导致不同 CPU 在不同时间看到递增操作。这意味着特定 CPU 可以认为其他一些 CPU 已经向前运行了，在计数器还没有实际递增前，就使用计数器的新值。实际上，从原理上来说，一个特定 CPU 可能早在上一次内存屏障之前，就看到 rcu_ctrlblk.completed 的下一个递增。(注意这句话并不是非常严密，如果想正确了解内存屏障，请参见附录)。

由于 rcu_read_lock()不包含内存屏障，相应的 RCU 读临界区可能被 CPU 重排到 rcu_read_unlock()后。因此，需要内存屏障来确保 RCU 读临界区的操作确实完成。

正如我们看到的，不同 CPU 可能在不同时间看到 flip 变量的变化，这个事实意味着一次状态机的转换对一个 grace period 来说是不够的：需要多次转换。

D.4.2.3.2 Grace-Period 状态机走查

```
1 void rcu_check_callbacks(int cpu, int user)
2 {
3 unsigned long flags;
4 struct rcu_data *rdp = RCU_DATA_CPU(cpu);
5
6 rcu_check_mb(cpu);
7 if (rcu_ctrlblk.completed == rdp->completed)
8 rcu_try_flip();
9 spin_lock_irqsave(&rdp->lock, flags);
10 RCU_TRACE_RDP(rcupreempt_trace_check_callbacks, rdp);
11 __rcu_advance_callbacks(rdp);
12 spin_unlock_irqrestore(&rdp->lock, flags);
13 }
```

图 D.65: `rcu_check_callbacks()` Implementatio

本节走查实现 RCU grace-period 状态机的 C 代码，它们被调度时钟中断调

用，在关中断情况下调用 rcu_check_callbacks()。这个函数的实现如图 D.65。第 4 行获得当前 CPU 相关的 rcu_data 结构，第 6 行检查这个 CPU 是否有必要执行一个内存屏障，将状态机带出 rcu_try_flip_waitmb_state 状态。第 7 行检查该 CPU 是否已经意识到当前 grace-period 阶段，如果是，则第 8 行尝试将状态机向前推进。第 9、12 行获得并释放 rcu_data 的锁，如果可以，则在第 11 行加快回调函数的执行。在通过 CONFIG_RCU_TRACE 打开调试的情况下，第 10 行更新 RCU 调试统计信息。

```
1 static void rcu_check_mb(int cpu)
2 {
3 if (per_cpu(rcu_mb_flag, cpu) == rcu_mb_needed) {
4 smp_mb();
5 per_cpu(rcu_mb_flag, cpu) = rcu_mb_done;
6 }
7 }
```

图 D.66: `rcu_check_mb()` Implementation

rcu_check_mb() 函数在必要时执行一个内存屏障，如上图 D.66。第 3 行检查当前 CPU 是否有必要执行一个内存屏障，如果是，则第 4 行执行它，第 5 行告知状态机。注意：这个内存屏障确保任何看到新 rcu_mb_flag 的 RCU 也将看到这个 CPU 之前的 RCU 读临界区的内存操作。

```
1 static void rcu_try_flip(void)
2 {
3 unsigned long flags;
4
5 RCU_TRACE_ME(rcupreempt_trace_try_flip_1);
6 if (!spin_trylock_irqsave(&rcu_ctrlblk.fliplock, flags)) {
7 RCU_TRACE_ME(rcupreempt_trace_try_flip_e1);
8 return;
9 }
10 switch (rcu_try_flip_state) {
11 case rcu_try_flip_idle_state:
12 if (rcu_try_flip_idle())
13 rcu_try_flip_state = rcu_try_flip_waitack_state;
14 break;
15 case rcu_try_flip_waitack_state:
16 if (rcu_try_flip_waitack())
17 rcu_try_flip_state = rcu_try_flip_waitzero_state;
18 break;
19 case rcu_try_flip_waitzero_state:
20 if (rcu_try_flip_waitzero())
21 rcu_try_flip_state = rcu_try_flip_waitmb_state;
22 break;
```

```
23 case rcu_try_flip_waitmb_state:
24 if (rcu_try_flip_waitmb())
25 rcu_try_flip_state = rcu_try_flip_idle_state;
26 }
27 spin_unlock_irqrestore(&rcu_ctrlblk.fliplock, flags);
28 }
```

图 D.67: `rcu_try_flip()` Implementation

rcu_try_flip() 函数实现顶层的 RCU grace-period 状态机，如图 D.67 所示。第 6 行尝试获得全局 RCU 状态机锁，如果不成功则返回。第 5、7 行增加 RCU 状态统计 (也是在 CONFIG_RCU_TRACE 打开的情况下)。第 10- 26 行执行状态机，每一个状态调用一个特定函数。如果需要转换状态，每一个函数将返回，否则返回 0。原则上，下一个状态应当被立即执行，但是在实践中，我们选择不这样做，以降低延迟。最后，第 27 行释放全局 RCU 状态机锁，它在第 6 行获得。

```
1 static int rcu_try_flip_idle(void)
2 {
3 int cpu;
4
5 RCU_TRACE_ME(rcupreempt_trace_try_flip_i1);
6 if (!rcu_pending(smp_processor_id())) {
7 RCU_TRACE_ME(rcupreempt_trace_try_flip_ie1);
8 return 0;
9 }
10 RCU_TRACE_ME(rcupreempt_trace_try_flip_g1);
11 rcu_ctrlblk.completed++;
12 smp_mb();
13 for_each_cpu_mask(cpu, rcu_cpu_online_map)
14 per_cpu(rcu_flip_flag, cpu) = rcu_flipped;
15 return 1;
16 }
```

图 D.68: `rcu_try_flip_idle()` Implementation

当 RCU grace-period 状态机是空闲状态时，rcu_try_flip_idle() 函数被调用。因此当必要时，它有责任启动 grace-period。它的代码如图 D.68 所示。第 6 行检查是否有任何 RCU grace-period 任务在本 CPU 上被挂起，如果没有，则在第 8 行退出，告诉顶层状态机继续处于空闲状态。如果有任务需要处理，第 11 行递增 grace-period 阶段计数，第 23 行执行一个内存屏障，以确保在看到需要的响应前，先看到新的计数。第 13、14 行设置所有在线 CPUs 的 rcu_flip_flag。最后，第 15 告知顶层状态机将状态推进到下一个状态。

```
1 static int rcu_try_flip_waitack(void)
2 {
```

```
3 int cpu;
4
5 RCU_TRACE_ME(rcupreempt_trace_try_flip_a1);
6 for_each_cpu_mask(cpu, rcu_cpu_online_map)
7 if (per_cpu(rcu_flip_flag, cpu) != rcu_flip_seen) {
8 RCU_TRACE_ME(rcupreempt_trace_try_flip_ae1);
9 return 0;
10 }
11 smp_mb();
12 RCU_TRACE_ME(rcupreempt_trace_try_flip_a2);
13 return 1;
14 }
```

图 D.69: `rcu_try_flip_waitack()` Implementation

rcu_try_flip_waitack() 函数在图中展示，检查在线 CPU 是否已经应答了 flip 计数 (AKA "increment", but called "flip" because the bottom bit, which rcu_read_lock() uses to index the rcu_flipctr array, does flip)。如果已经应答了，它告知顶层状态机推进到下一个状态。

第 6 行循环遍历所有在线 CPUs，第 7 行检查当前 CPU 是否已经应答最后一个 flip 计数。如果没有，第 9 行告知顶层 grace-period 状态机继续保持当前状态。否则，如果所有在线 CPUs 已经应答了，则第 11 行执行一个内存屏障，以确保最后一个 CPU 应答前，我们不必检查 0。这看起来有点可疑，但是 CPUs 设计者有时就是会做这些奇怪的事情。最后，第 13 行告知顶层 grace-period 状态机推进到到下一个状态。

```
1 static int rcu_try_flip_waitzero(void)
2 {
3 int cpu;
4 int lastidx = !(rcu_ctrlblk.completed & 0x1);
5 int sum = 0;
6
7 RCU_TRACE_ME(rcupreempt_trace_try_flip_z1);
8 for_each_possible_cpu(cpu)
9 sum += per_cpu(rcu_flipctr, cpu)[lastidx];
10 if (sum != 0) {
11 RCU_TRACE_ME(rcupreempt_trace_try_flip_ze1);
12 return 0;
13 }
14 smp_mb();
15 for_each_cpu_mask(cpu, rcu_cpu_online_map)
16 per_cpu(rcu_mb_flag, cpu) = rcu_mb_needed;
17 RCU_TRACE_ME(rcupreempt_trace_try_flip_z2);
18 return 1;
```

```
19 }
```

图 D.70: rcu_try_flip_waitzero() Implementation

rcu_try_flip_waitzero() 函数如图 D.70 所示，检查所有已经存在的 RCU 读临界区是否已经完成，如果是，则告知状态机向前推进其状态。第 8、9 行计算计数值的总和。第 10 行检查结果是否为 0，如果不是，第 12 行告知状态机保留原来的状态。否则，第 14 行执行一个内存屏障，以确保没有 CPU 会在已经退出的 RCU 读临界区之前看到后续的调用结果。这看起来也有点可疑，但是 CPUs 设计者就是这样。第 15 、 16 行设置所有在线 CPUs 的 rcu_mb_flag 变量，第 18 行告知状态机将状态推进到下一个状态。

```
1 static int rcu_try_flip_waitmb(void)
2 {
3 int cpu;
4
5 RCU_TRACE_ME(rcupreempt_trace_try_flip_m1);
6 for_each_cpu_mask(cpu, rcu_cpu_online_map)
7 if (per_cpu(rcu_mb_flag, cpu) != rcu_mb_done) {
8 RCU_TRACE_ME(rcupreempt_trace_try_flip_me1);
9 return 0;
10 }
11 smp_mb();
12 RCU_TRACE_ME(rcupreempt_trace_try_flip_m2);
13 return 1;
14 }
```

图 D.71: `rcu_try_flip_waitmb()` Implementation

rcu_try_flip_waitmb() 函数如图 D.71 所示,它检查所有在线 CPU 是否已经执行了需要的内存屏障，如果是，则告知状态机向前推进。第 6、7 行检查每一个在线 CPU 是否已经执行了必要的内存屏障，如果没有，第 9 行告知状态机不要向前推进。否则，如果所有 CPU 已经执行一个内存屏障，第 11 行执行一个内存屏障，第 13 行告知状态机向前推进。

```
1 static void __rcu_advance_callbacks(struct rcu_data *rdp)
2 {
3 int cpu;
4 int i;
5 int wlc = 0;
6
7 if (rdp->completed != rcu_ctrlblk.completed) {
8 if (rdp->waitlist[GP_STAGES - 1] != NULL) {
9 *rdp->donetail = rdp->waitlist[GP_STAGES - 1];
10 rdp->donetail = rdp->waittail[GP_STAGES - 1];
```

```
11 RCU_TRACE_RDP(rcupreempt_trace_move2done, rdp);
12 }
13 for (i = GP_STAGES - 2; i >= 0; i--) {
14 if (rdp->waitlist[i] != NULL) {
15 rdp->waitlist[i + 1] = rdp->waitlist[i];
16 rdp->waittail[i + 1] = rdp->waittail[i];
17 wlc++;
18 } else {
19 rdp->waitlist[i + 1] = NULL;
20 rdp->waittail[i + 1] =
21 &rdp->waitlist[i + 1];
22 }
23 }
24 if (rdp->nextlist != NULL) {
25 rdp->waitlist[0] = rdp->nextlist;
26 rdp->waittail[0] = rdp->nexttail;
27 wlc++;
28 rdp->nextlist = NULL;
29 rdp->nexttail = &rdp->nextlist;
30 RCU_TRACE_RDP(rcupreempt_trace_move2wait, rdp);
31 } else {
32 rdp->waitlist[0] = NULL;
33 rdp->waittail[0] = &rdp->waitlist[0];
34 }
35 rdp->waitlistcount = wlc;
36 rdp->completed = rcu_ctrlblk.completed;
37 }
38 cpu = raw_smp_processor_id();
39 if (per_cpu(rcu_flip_flag, cpu) == rcu_flipped) {
40 smp_mb();
41 per_cpu(rcu_flip_flag, cpu) = rcu_flip_seen;
42 smp_mb();
43 }
44 }
```

图 D.72: __rcu_advance_callbacks() Implementation

__rcu_advance_callbacks() 函数如图 D.72 所示，推进回调函数，并对 flip 计数器进行应答。第 7 行检查全局 rcu_ctrlblk.completed 计数器自当前 CPU 上次调用本函数以来，是否已经变化。如果没有，回调函数不需要推进 (第 8-37 行)。否则，第 8 – 37 行推进回调函数链表(在 wlc 变量中维护了非空链表数量)。其他情况下，第 38-43 行在必要时应答 flip 计数。

问题 D.58：当第 3-37 行没有执行时，第 38-43 行是如何被执行的?

D.4.2.4 读端原语

本节解释 rcu_read_lock() 和 rcu_read_unlock() 原语，随后讨论这个实现如何处理两个原语没有包含内存屏障这个事实。

D.4.2.4.1 rcu_read_lock()

```
1 void __rcu_read_lock(void)
2 {
3 int idx;
4 struct task_struct *t = current;
5 int nesting;
6
7 nesting = ACCESS_ONCE(t->rcu_read_lock_nesting);
8 if (nesting != 0) {
9 t->rcu_read_lock_nesting = nesting + 1;
10 } else {
11 unsigned long flags;
12
13 local_irq_save(flags);
14 idx = ACCESS_ONCE(rcu_ctrlblk.completed) & 0x1;
15 ACCESS_ONCE(__get_cpu_var(rcu_flipctr)[idx])++;
16 ACCESS_ONCE(t->rcu_read_lock_nesting) = nesting + 1;
17 ACCESS_ONCE(t->rcu_flipctr_idx) = idx;
18 local_irq_restore(flags);
19 }
20 }
```

图 D.73: `__rcu_read_lock()` Implementation

rcu_read_lock()的实现如图 D.73 所示。第 7 行获得任务的 RCU 读临界区嵌套次数。如果第 8 行发现这个计数值为非 0，则我们已经在另外一个 rcu_read_lock()的保护中，这种情况下，第 9 行简单的递增这个计数。

但是，如果这是最外层的 rcu_read_lock()，那么需要做更多的事情。第 13、18 行禁止并恢复中断，以确保相应代码既不被抢占，也不被调度时钟中断打断(它会运行 grace period 状态机)。第 14 行获得 grace period 计数，第 15 行递增本 CPU 的当前计数，第 16 行递增嵌套计数，第 17 行记录旧/新计数器索引，这样 rcu_read_unlock() 可以递减相应的计数。

ACCESS_ONCE() 宏强制编译按顺序访问变量。虽然这不会防止 CPU 重排内存访问，但是它确保本 CPU 上的 NMI 和 SMI 中断处理程序可以按顺序访问。

这是特别重要的：

在对 idx 赋值时缺少 ACCESS_ONCE() ，编译将能够：(a) 去除局部变量 idx (b) 将第 14 行的递增编译为获取-递增-存储操作，对 rcu_ctrlblk.completed 的访问将是分开的获取、存储操作。如果在此期间， rcu_ctrlblk.completed 已经变化，将损坏 rcu_flipctr 的值。

如果对 rcu_read_lock_nesting 的赋值 (第 17 行) 被重排在对 rcu_flipctr 的递增 (第 16 行)前，并且 NMI 在这两个事件之间发生，那么在 NMI 处理函数中的 rcu_read_lock 将错误的认为它已经在 rcu_read_lock()的保护中了。

如果对 rcu_read_lock_nesting 的赋值(第 17 行) 被重排到对 rcu_flipctr_idx 的赋值之后 (第 18 行)，并且在这两个事件之间发生 NMI，那么在 NMI 中的 rcu_read_lock() 将破坏 rcu_flipctr_idx，可能导致相应的 rcu_read_unlock() 递减错误的计数值。继而导致过早的结束 grace period，不确定的延迟一个 grace period，二者也可能同时发生。

不好确定对 nesting 的赋值使用 ACCESS_ONCE 是否必须。也不确定 smp_read_barrier_depends() (第 15 行)是否是必须的：添加它以确保修改 index 与后面的值是按顺序的。

第 13 行到第 19 行需要将中断关闭的原因如下:

假设一个 CPU 装载 rcu_ctrlblk.completed (第 14 行)，然后第二个 CPU 递增这个值，然后第一个 CPU 执行一个调度时钟中断。第一个 CPU 将发现它需要应答 flip 计数，并且它将这样做。这个应答将承诺避免递增旧计数器，但是这个 CPU 将破坏这个承诺。更糟糕的是，这个 CPU 可能在从调度时钟中断返回时立即被抢占，因此最终对这个计数器的递增将在随后的任一时刻发生。任意一种情况都将使 grace-period 检测变得混乱起来。

禁止中断还有一个边际效果是禁止抢占。如果这个代码在获取 rcu_ctrlblk.completed (第 14 行)和递增 rcu_flipctr (第 16 行)之间被抢占，它可能被迁移到其他 CPU。这将导致非原子的递增计数。如果这个 CPU 碰巧在执行 rcu_read_lock() 或者 rcu_read_unlock()，其中一个递增或者递减将丢失，这也使得 grace-period 检测变得混乱起来。在 RISC 机器上，如果抢占发生递增过程中，相同的结果也可能发生 (在获取旧值之后，但是在存储旧计数值前)。

在第 16 行中间发生抢占，并且抢占介于选择当前 CPU 的 rcu_flipctr 数组拷贝及递增元素之间，也会导致类似的错误。其他 CPU 的递增或者递减可能被丢失。

不禁止抢占也可能使先前的 RCU 失败， 先前的 RCU 依赖 rcu_read_lock_nesting 以确定一个任务什么时候处于一个 RCU 读临界区中。例如，一个任务在递增 rcu_flipctr 后，在更新 rcu_read_lock_nesting 前发生抢占，那

么它将延迟 RCU grace periods。

当然，最后三个理由可以通过禁止抢占而不是关中断来解决，但是无论如何，第一个原因需要禁止中断。

D.4.2.4.2 rcu_read_unlock()

```
1 void __rcu_read_unlock(void)
2 {
3 int idx;
4 struct task_struct *t = current;
5 int nesting;
6
7 nesting = ACCESS_ONCE(t->rcu_read_lock_nesting);
8 if (nesting > 1) {
9 t->rcu_read_lock_nesting = nesting - 1;
10 } else {
11 unsigned long flags;
12
13 local_irq_save(flags);
14 idx = ACCESS_ONCE(t->rcu_flipctr_idx);
15 ACCESS_ONCE(t->rcu_read_lock_nesting) = nesting - 1;
16 ACCESS_ONCE(__get_cpu_var(rcu_flipctr)[idx])--;
17 local_irq_restore(flags);
18 }
19 }
```

图 D.74: `__rcu_read_unlock()` Implementation

rcu_read_unlock() 的实现如图 D.74。第 7 行获得 rcu_read_lock_nesting 计数，第 8 行检查是否在一个嵌套 rcu_read_lock() 保护之中。如果是，第 9 行简单的递减计数。

但是，就象 rcu_read_lock()，我们在其他情况下需要做更多的事情。第 13、17 行禁止并恢复中断，以防止在 rcu_read_unlock 过程中调用 grace-period 状态机时，产生调度时钟中断。第 14 行获得 rcu_flipctr_idx，它被相应的 rcu_read_lock() 保存，第 15 行递减 rcu_read_lock_nesting，这样后面的中断和 NMI/SMI 处理函数将更新 rcu_flipctr，第 16 行递减计数(以与相应的 rcu_read_lock()使用相同的索引进行递减, 但是可能是在不同的 CPU 上)。

与 rcu_read_lock()的原因相同，ACCESS_ONCE() 宏 以及中断禁止也是必须的。

问题 D.59: 如果 rcu_read_unlock()中包含 ACCESS_ONCE()的行被编译器

乱序后会出现什么问题？

问题 D.60：如果 rcu_read_unlock()中包含 ACCESS_ONCE()的行被 cpu 乱序后会出现什么问题？

Quick Quiz D.61：如果在 rcu_read_unlock() 中中断没有被关闭将出现什么问题？

D.4.2.4.3 内存屏障方面的考虑

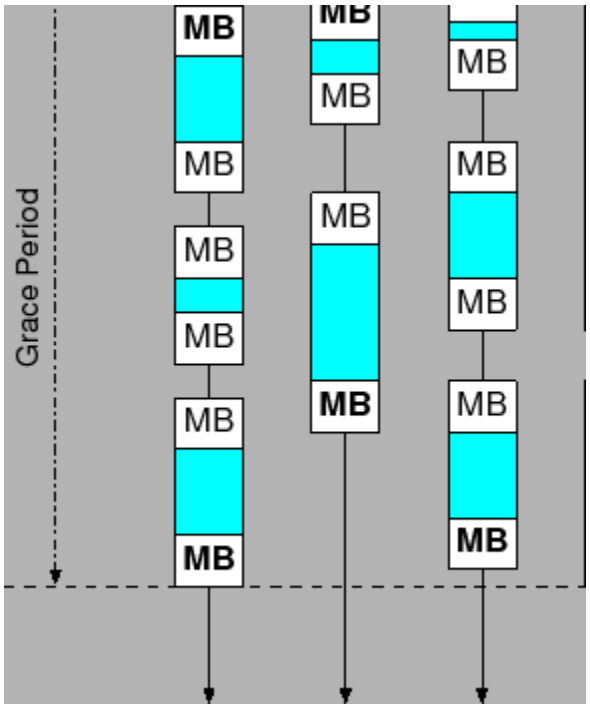


图 D.75: Preemptible RCU with Read-Side Memory Barriers

请注意这两个原语不包含内存屏障，因此不会阻止 CPU 在执行 rcu_read_lock()之前或者在执行 rcu_read_unlock()之后执行临界区代码。rcu_try_flip_waitmb_state 的目的是为了解决可能的乱序。但是仅仅在一个 grace period 开始或者结束时。为了明白为什么这个方法是有用的，考虑图 D.75，它展示在每一个 RCU 开始和结束的地方设置一个内存屏障这种传统的方法的消耗[MSMB06]。

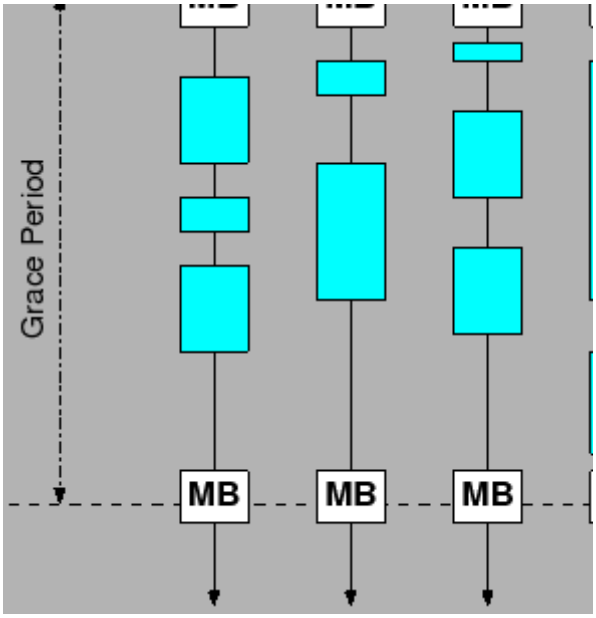


图 D.76: Preemptible RCU with Grace-Period Memory Barriers

"MB"表示内存屏障，并且仅仅边界上的屏障才是必须的，也就是说，在一个特定 CPU 上，每一个 grace period 的第一个和最后一个屏障才是必须的。因此可抢占 RCU 实现将内存屏障与 grace period 相关联，如图 D.76 所示。

LINUX 内核可以在每一个 grace period 执行上百万个 RCU 读临界区，后一种方法可以导致实质上的节省读端时间，这是由于它将所有读临界区的内存屏障分摊到一个 grace period 中。

D.4.3 验证可抢占 RCU

D.4.3.1 测试

可抢占 RCU 算法分别在弱序的 POWER4 和 POWER5 CPUs 上使用 rcutorture 运行了超过 24 小时，并且没有错误。当然，没有办法来证明这个算法是正确的。至多，这说明要么是这两个机器非常幸运，要么可抢占 RCU 中的 BUG 有极端低的发生概率。因此我们需要更多的验证。

这个任务需要采用推理证明的方法，将在下节中描述。

D.4.3.2 推理验证

由于 rcu_read_lock() 和 rcu_read_unlock() 都不包含内存屏障，因此 RCU 读临界区在弱序机器上可能乱序。而且，相对宽松的 RCU 实现允许 CPU 在一个

grace period 开始结束处就内存一致性达成一致。这带来一个问题：一个特定的 RCU 读临界区可能将相关的 grace-period 状态机延长多久？

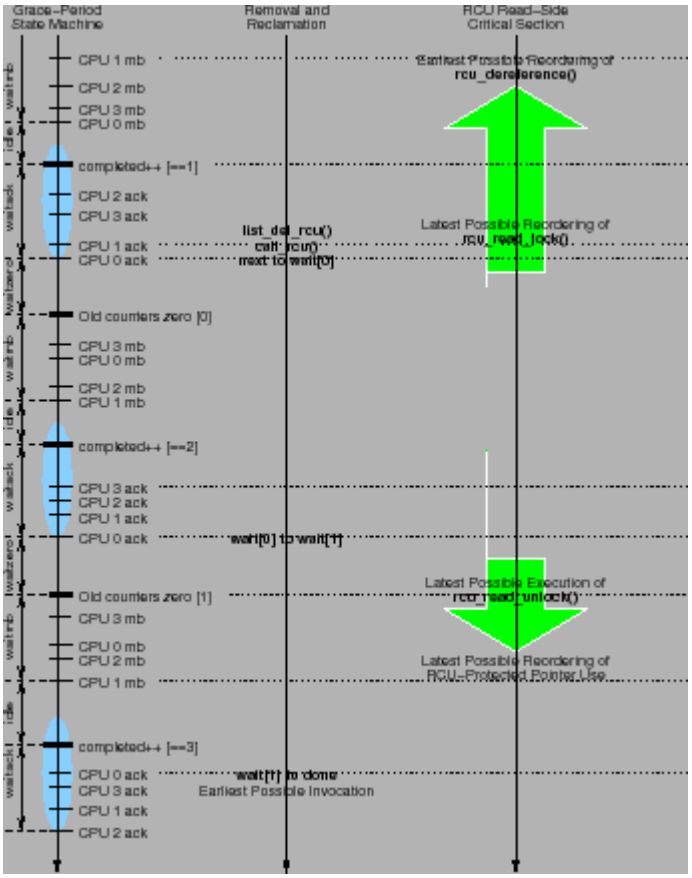


图 D.77: Preemptible RCU Worst-Case Scenario

最坏的情况如图 D.77 所示。在此，CPU 0 执行尽可能短的移动回收操作，而 CPU 1 执行尽可能长的读临界区。由于仅仅在应答 flip 计数器前，回调队列才会被移动，最后，CPU 0 在调度中断应答 flip 计数前执行 list_del_rcu() 和 call_rcu().call_rcu() 将回调放到 CPU 0 的下一个列表中，中断将从下一个列表移动回调到 wait[0] 列表。这个回调将在 CPU 0 的下一个 flip 计数器后的第一个调度时钟中断中再次移动 (从 wait[0] 列表移到 wait[1] 列表)。类似的，回调将在 flip 计数器变成 3 后的第一个调度时钟中断中从 wait[1] 列表移动到完成列表。随后，回调将立即被调用。

同时, CPU 1 执行一个 RCU 读临界区。让我们假设 rcu_read_lock() 在第一个 flip 计数之后(其结果是它的值为 1), 这样 rcu_read_lock() 递增 CPU 1 rcu_flipctr[1] 计数。注意：由于 rcu_read_lock() 不包含任何内存屏障，临界区的内容可能被 CPU 提前执行。但是，这个提早执行过程不能被 CPU 1 在最后一个内存屏障之前执行。如图所示。然而，这也够早的了， rcu_dereference() 可能获得一个被 CPU 0 的 list_del_rcu()删除的项目。

由于 rcu_read_lock() 递增 index-1 计数器，相应的 rcu_read_unlock() 必须在

第一个索引的 "旧计数器变为 0 " 事件之前。但是，由于 rcu_read_unlock() 包含一个内存屏障，相应的 RCU 读临界区的内容 (可能包含一个被 CPU 0 删除项目的引用) 可能被 CPU 1 随后执行。但是，它不能在 CPU 1 的下一个内存屏障之后执行，如图所示。由于 CPU 1 最后的引用早于 CPU 0 最早的回调，两次穿越 grace-period 状态机足够形成一个完整的 grace period，因此下面的做法是安全的：

```
#define GP_STAGES 2
```

问题 D.62：假设在 rcu_read_lock()的禁止中断操作被替换为禁止抢占，将对 GP_STAGES 有什么影响?

问题 D.63：为什么 rcu_dereference() 不能在内存屏障之前？

D.4.3.3 形式验证

对这个算法进行形式验证是十分重要的，但是需要做更多的工作。一个完成这个验证的工具在附录中描述。

问题 D.64：有什么更准确的方式来说明"CPU 0 可能早在 CPU1 的最后一个内存屏障之时看到 CPU1 的递增"?

E.形式验证

并行算法难于编写，甚至难于调试。测试虽然是必要的，但是往往还不够。致命的竞争条件发生的几率可能很小。正确性检查是有意义的。

拥有一个工具来找到所有竞争条件是非常有用的。一些这样的工具已经存在，例如，Promela 语言和它的编译器 Spin，将在本章中描述。E.1 节提供了 Promela 和 Spin 的简要介绍。E.2 节展示了 Promela 和 Spin 的用法，以找到一个非原子性递增的竞争示例。E.3 节使用 Promela 和 Spin 验证一个类似的原子递增示例。E.4 节给出了使用 Promela 和 Spin 的简要介绍。E.5 节展示了一个 spinlock 的 Promela 模型。E.6 节应用 Promela 和 spin 验证一个简单的 RCU 实现。E.7 节应用 Promela 验证一个可抢占 RCU 和 dyntick-idle 之间的竞争条件。E.8 节提供了一个不需要正式验证的简单接口。最后，E.9 节总结了使用形式确认工具以验证并行算法。

E.1 什么是 Promela 和 Spin?

Promela 是一种设计用来帮助验证协议的语言，但是也能用于验证小的并行算法。您用类似于 C 语言的 Promela 来重新编写算法，并进行正确性验证。然后使用 Spin 来将其转化为可以编译运行的 C 程序。最终的程序搜索整个算法状态。

这个全状态的搜索是非常强的，但是也是一个双刃剑。如果您的算法太复杂，或者您的 Promela 实现得比较粗心，内存可能会不足。而且，即使有充足的内存，搜索过程也会消耗太多的内存。因此，请在算法复杂但是小的并行程序中使用这个工具。轻率的试图将它用在中等规模的算法（更不用说整个 LINUX 内核）将非常糟糕。

Promela 和 Spin 可以通过 http://spinroot.com/spin/whatispin.html 下载。

上面的链接也提供到 Gerard Holzmann 关于 [Hol03] Promela 和 Spin 方面的书籍的链接，可以通过下面的链接进行在线搜索:

http://www.spinroot.com/spin/Man/index.html.

本书余下部分描述了如何使用 Promela 来调试并行算法，先从简单例子开始，然后逐渐深入。

E.2 Promela 示例：非原子性递增

```
1 #define NUMPROCS 2
2
3 byte counter = 0;
4 byte progress[NUMPROCS];
5
6 proctype incrementer(byte me)
7 {
8 int temp;
9
10 temp = counter;
11 counter = temp + 1;
12 progress[me] = 1;
13 }
14
15 init {
16 int i = 0;
17 int sum = 0;
18
19 atomic {
20 i = 0;
21 do
22 :: i < NUMPROCS ->
23 progress[i] = 0;
24 run incrementer(i);
25 i++
26 :: i >= NUMPROCS -> break
27 od;
28 }
29 atomic {
30 i = 0;
31 sum = 0;
32 do
33 :: i < NUMPROCS ->
34 sum = sum + progress[i];
35 i++
36 :: i >= NUMPROCS -> break
37 od;
38 assert(sum < NUMPROCS || counter == NUMPROCS)
39 }
40 }
```

图 E.1: Promela Code for Non-Atomic Increment

图 E.1 展示了教科书上非原子性递增产生的竞争条件结果。第 1 行定义了运行的进程个数(我们将改变它以观察状态变化的效果)，第 3 行定义了计数器，第 4 行用于实现第 29-39 行的断言。

第 6-13 行定义一个过程，用来非原子性的递增计数器。参数 me 是进程编号，由初始化代码设置。 由于简单的 Promela 语句都假设为原子的，我们必须将递增代码分成第 10-11 两行。第 12 行的赋值完成处理过程。由于 Spin 系统会搜索整个状态空间，包含可能的状态顺序，因此没有必要使用传统的测试。

第 15-40 行是初始化代码块，它最先被执行。第 19-28 行实际进行初始化，而第 29-39 行执行断言。它们都是原子块，以避免不必要的增加状态空间。因为它们与算法无关。

第 21-27 行的 do-od 结构实现了一个 Promela 循环，可以认为它是一个包含 switch 语句的 C for (;;) 循环。条件块 (前缀是 ::) 并不一定会被扫描。第 22-25 行的 do-od 块初始化递增单元，运行递增函数，最后递增变量 i，第 26 行的第二个 do-od 块在任务启动后就退出循环。

第 29-39 行的原子块也包含一个类似的 do-od 循环，它计算计数值的总和。第 38 行的 assert() 语句检验是否所有的过程都已经完成了。

您可以按如下的方法编译运行程序：

```
spin -a increment.spin              # Translate the model to C
cc -DSAFETY -o pan pan.c            # Compile the model
./pan                               # Run the model
```

```
pan: assertion violated ((sum<2)||(counter==2)) (at depth 20)
pan: wrote increment.spin.trail
(Spin Version 4.2.5 -- 2 April 2005)
Warning: Search not completed
+ Partial Order Reduction
Full statespace search for:
never claim - (none specified)
assertion violations +
cycle checks - (disabled by -DSAFETY)
invalid end states +
State-vector 40 byte, depth reached 22, errors: 1
45 states, stored
13 states, matched
58 transitions (= stored+matched)
51 atomic steps
hash conflicts: 0 (resolved)
```

2.622 memory usage (Mbyte)

图 E.2: Non-Atomic Increment spin Output

这将会产生如图 E.2 的输出。第一行告诉我们断言被违反了。第二行中的跟踪文件描述断言是如何违反的。``Warning" 行重申在我们的模式下，并非全部都扫描了。第二段描述状态搜索的类型。第三节给出状态长度统计：小的模式下有 45 种状态。最后一行显示了使用的内存。

如下命令将跟踪文件变为可读的格式:

spin -t -p increment.spin

```
Starting :init: with pid 0
1: proc 0 (:init:) line 20 "increment.spin" (state 1) [i = 0]
2: proc 0 (:init:) line 22 "increment.spin" (state 2) [((i<2))]
2: proc 0 (:init:) line 23 "increment.spin" (state 3) [progress[i] = 0]
Starting incrementer with pid 1
3: proc 0 (:init:) line 24 "increment.spin" (state 4) [(run incrementer(i))]
3: proc 0 (:init:) line 25 "increment.spin" (state 5) [i = (i+1)]
4: proc 0 (:init:) line 22 "increment.spin" (state 2) [((i<2))]
4: proc 0 (:init:) line 23 "increment.spin" (state 3) [progress[i] = 0]
Starting incrementer with pid 2
5: proc 0 (:init:) line 24 "increment.spin" (state 4) [(run incrementer(i))]
5: proc 0 (:init:) line 25 "increment.spin" (state 5) [i = (i+1)]
6: proc 0 (:init:) line 26 "increment.spin" (state 6) [((i>=2))]
7: proc 0 (:init:) line 21 "increment.spin" (state 10) [break]
8: proc 2 (incrementer) line 10 "increment.spin" (state 1) [temp = counter]
9: proc 1 (incrementer) line 10 "increment.spin" (state 1) [temp = counter]
10: proc 2 (incrementer) line 11 "increment.spin" (state 2) [counter = (temp+1)]
11: proc 2 (incrementer) line 12 "increment.spin" (state 3) [progress[me] = 1]
12: proc 2 terminates
13: proc 1 (incrementer) line 11 "increment.spin" (state 2) [counter = (temp+1)]
14: proc 1 (incrementer) line 12 "increment.spin" (state 3) [progress[me] = 1]
15: proc 1 terminates
16: proc 0 (:init:) line 30 "increment.spin" (state 12) [i = 0]
16: proc 0 (:init:) line 31 "increment.spin" (state 13) [sum = 0]
17: proc 0 (:init:) line 33 "increment.spin" (state 14) [((i<2))]
17: proc 0 (:init:) line 34 "increment.spin" (state 15) [sum = (sum+progress[i])]
17: proc 0 (:init:) line 35 "increment.spin" (state 16) [i = (i+1)]
18: proc 0 (:init:) line 33 "increment.spin" (state 14) [((i<2))]
18: proc 0 (:init:) line 34 "increment.spin" (state 15) [sum = (sum+progress[i])]
18: proc 0 (:init:) line 35 "increment.spin" (state 16) [i = (i+1)]
19: proc 0 (:init:) line 36 "increment.spin" (state 17) [((i>=2))]
20: proc 0 (:init:) line 32 "increment.spin" (state 21) [break]
spin: line 38 "increment.spin", Error: assertion violated
```

```
spin: text of failed assertion: assert(((sum<2)||(counter==2)))
21: proc 0 (:init:) line 38 "increment.spin" (state 22) [assert(((sum<2)||(counter==2)))]
spin: trail ends after 21 steps
#processes: 1
counter = 1
progress[0] = 1
progress[1] = 1
21: proc 0 (:init:) line 40 "increment.spin" (state 24) <valid end state>
3 processes created
```

图 E.3: Non-Atomic Increment Error Trail

E.3 Promela 示例: 原子递增

```
1 proctype incrementer(byte me)
2 {
3 int temp;
4
5 atomic {
6 temp = counter;
7 counter = temp + 1;
8 }
9 progress[me] = 1;
10 }
```

图 E.4: Promela Code for Atomic Increment

```
(Spin Version 4.2.5 -- 2 April 2005)
+ Partial Order Reduction
Full statespace search for:
never claim - (none specified)
assertion violations +
cycle checks - (disabled by -DSAFETY)
invalid end states +
State-vector 40 byte, depth reached 20,
errors: 0
52 states, stored
21 states, matched
73 transitions (= stored+matched)
66 atomic steps
hash conflicts: 0 (resolved)
```

```
2.622 memory usage (Mbyte)
unreached in proctype incrementer
(0 of 5 states)
unreached in proctype :init:
(0 of 24 states)
```

图 E.5: Atomic Increment spin Output

可以象上图那样，简单的将递增函数放到原子块中。也可以简单的将语句行换成 counter = counter + 1，因为 Promela 语句是原子的。两种方法，都会得到无错的跟踪文件。如上图。

E.3.1 组合

表 E.1: Memory Usage of Increment Model

# incrementers	# states	mega bytes
1	11	2.6
2	52	2.6
3	372	2.6
4	3,496	2.7
5	40,221	5.0
6	545,720	40.5
7	8,521,450	652.7

表 E.1 显示了状态数量以及内存消耗(通过重新定义 NUMPROCS):

运行不必要的大的模型如此令人气馁，虽然 652MB 在现代的桌面及笔记本电脑中不算什么限制。

通过这个例子，我们结束使用什么命令来分析 Promela 模型，接下来看看更多详细的例子。

E.4 如何使用 Promela

给定一个源文件 qrcu.spin, 可以使用如下命令:

spin -a qrcu.spin 创建一个文件 pan.c ，该文件搜索所有状态机。

cc -DSAFETY -o pan pan.c 编译生成的状态机。如果仅仅只有断言，-DSAFETY 最优化的生成结果。如果有某些检查，则必须不使用-DSAFETY 进行编译。

-DSAFETY 优化结果是大大提高速度，因此当能够使用它时，就应当使用它。一个不能使-DSAFETY 的例子是通过 DMP 检查活锁。

./pan 真正的搜索状态空间。即使是小的状态机，状态数量也可能达到上千万个。因此需要一个大内存的机器。例如， 有 3 个读者和 2 个写者的 qrcu.spin 需要 2.7GB 内存。

如果您不确定自己是否有足够内存，就在一个窗口运行 top，在另一个窗口运行./pan 。关注 ./pan 窗口，这样可以快速的杀死它。

不要忘记获取输出，特别是您在远程机器上工作的时候。

如果您的模型包含检测，则需要通过-f 命令行参数打开 ``weak fairness'。

spin -t -p qrcu.spin 生成跟踪文件，输出产生错误的步骤。-g 将包含全局变量值的变化情况， -l 将包含局部变量值的变化情况。

E.4.1 Promela 特性

虽然所有计算机语言都是类似的，但是 Promela 仍然会给使用 C, C++, 或者 Java 的人们带来一些惊奇。

在 C 中，``;'' 结束一条语句。在 Promela 中，它将语句进行分隔。幸运的是，最近的 Spin 版本已经变得对此比较宽松了。

Promela 的循环，do 语句. do 语句类似于 if-then-else 语句循环。

在 C 的 switch 语句中，如果没有匹配的情况，整个语句被忽略。在 Promela 中，如果没有匹配的表达式，将得到一个错误。因此，如果错误输出指向无辜的代码行，请检查在 if 或 do 语句中是否遗漏了某个条件。

当在 C 中创建一个压力测试时，通常会运行一些假想的操作。在 Promela 中，通常是运行一个单一的操作，因为 Promela 会搜索出所有可能的结果。有时需要在 Promela 中进行循环操作，例如，如果重复进行多个操作，这样会大大增加状态空间的长度。

在 C 中，最简单的是维护一个循环计数并在适当的时候中止循环。在 Promela 中，循环计数必须被避免（就象避免瘟疫那样），因为它们会导致状态空间迅速增大。从另一个角度来说，在 Promela 中，使用一个无限循环来实现是没有问题的。

在 C 疲劳测试代码中，通常会使用一个每线程控制变量。这易于阅读，对调试测试代码来说也是很有帮助的。在 Promela 中，仅仅在没有其他可选方法时，才使用每线程变量。要明白这一点，考虑一个 5 任务的验证，每一位表示一个任务完成。这会导致 32 个状态。相对来说，一个简单的计数器只有 6 种状态，状态数整整少了 5 倍。5 倍看起来不是一个问题，但是如果要处理超过 150M 的状态呢？这需要 10GB 内存！

无论是 C 还是 Promela 的疲劳测试代码中，其中一个挑战是编写一个好的 assert.

在 Promela 中，分而治之是很有用的。这使得状态空间是可控的。将一个大的模型分成两个大致相等的部分，会导致状态空间减少到它的 sqrt 那么多。例如，100 万个状态组合将减少到一千个状态模型。不仅仅是 Promela 处理得更快，使用的内存更少，两个更小的算法也更易于让人理解。

E.4.2 Promela 编程技巧

Promela 被设计用于分析目的，因此将它用于并行编程不是很好。以下的技巧可以帮助您安全的用好它：

内存乱序。假设您有一对语句 ，它将全局变量 x、y 复制到局部变量 r1、r2 中。这有内存顺序方面的问题（没有锁保护），可以用如下方法：

```
1 if
2 :: 1 -> r1 = x;
3    r2 = y
4 :: 1 -> r2 = y;
5    r1 = x
6 fi
```

两个 if 分支是非确定性的。因为它们都是可用的。因为会搜索整个空间，所有选择最终会覆盖所有情况。

当然，过多的使用这个技巧会导致状态空间快速增长。另外，这需要您预料到可能的乱序情况。

```
1 i = 0;
2 sum = 0;
3 do
4 :: i < N_QRCU_READERS ->
5 sum = sum + (readerstart[i] == 1 &&
```

```
6 readerprogress[i] == 1);
7 i++
8 :: i >= N_QRCU_READERS ->
9 assert(sum == 0);
10 break
11 od
```

图 E.6: Complex Promela Assertion

```
1 atomic {
2 i = 0;
3 sum = 0;
4 do
5 :: i < N_QRCU_READERS ->
6 sum = sum + (readerstart[i] == 1 &&
7 readerprogress[i] == 1);
8 i++
9 :: i >= N_QRCU_READERS ->
10 assert(sum == 0);
11 break
12
```

图 E.7: Atomic Block for Complex Promela Assertion

状态压缩。如果有复杂的断言，使用 atomic 来赋值。最后，它们不是算法的一部分。其中一个例子如上图。

没有理由使用非原子的断言，因为它实际上不是算法的一部分。由于每一个语句都会增加状态数量，我们可以将它放到 atomic 块中，以减少无用的状态数量。

Promela 不提供函数。必须使用 C 预处理宏来代替。但是，避免小心的使用它们。

现在，我们准备见识一下更复杂的例子。

E.5 Promela 示例: 锁

```
1 #define spin_lock(mutex) \
2 do \
3 :: 1 -> atomic { \
4 if \
5 :: mutex == 0 -> \
6 mutex = 1; \
7 break \
8 :: else -> skip \
9 fi \
```

```
10 } \
11 od
12
13 #define spin_unlock(mutex) \
14 mutex = 0
```

图 E.8: Promela Code for Spinlock

由于锁通常是有用的， spin_lock() 和 spin_unlock() 宏在 lock.h 提供，可以被 Promela 模型包含，如图 E.8。spin_lock() 宏包含一个 do-od 死循环 （第 2-11 行）。循环体是一个单一的原子块，包含一个 if-fi 语句. if-fi 结构类似于 do-od 结构，但是它执行一次而不是一个循环。如果在第 5 行没有获得锁，第 6 行获取它，第 7 行则退出循环（也退出原子块）。另一方面，如果锁在第 8 行，锁已经被获取了，则什么也不做，结束 if-fi 以及 原子块，再次进入外层循环，直到锁可用。

spin_unlock() 宏简单的释放锁。

注意：不需要内存屏障，因为 Promela 假设是强序的。在任一指定的 Promela 状态，所有进程都与当前进程拥有一致的状态，以及一致的状态顺序。这与某些计算机系统的“顺序一致性”内存模型类似（如 IPS 和 PA-RISC）。

```
1 #include "lock.h"
2
3 #define N_LOCKERS 3
4
5 bit mutex = 0;
6 bit havelock[N_LOCKERS];
7 int sum;
8
9 proctype locker(byte me)
10 {
11 do
12 :: 1 ->
13 spin_lock(mutex);
14 havelock[me] = 1;
15 havelock[me] = 0;
16 spin_unlock(mutex)
17 od
18 }
19
20 init {
21 int i = 0;
22 int j;
23
24 end: do
```

```
25 :: i < N_LOCKERS ->
26 havelock[i] = 0;
27 run locker(i);
28 i++
29 :: i >= N_LOCKERS ->
30 sum = 0;
31 j = 0;
32 atomic {
33 do
34 :: j < N_LOCKERS ->
35 sum = sum + havelock[j];
36 j = j + 1
37 :: j >= N_LOCKERS ->
38 break
39 od
40 }
41 assert(sum <= 1);
42 break
43 od
44 }
```

图 E.9: Promela Code to Test Spinlocks

这些宏在图 E.9 所示的 Promela 代码中被测试。这段代码与测试递增的代码类似，第三行的 N_LOCKERS 宏定义了测试进程数量。mutex 在第 5 行定义，第 6 行定义的数组用于跟踪锁的持有者，第 7 行被断言代码使用，用来校验仅仅只有一个进程持有锁。

第 9-18 行是 locker 函数，在第 13 行简单的循环申请锁，第 14 行表示它获取了锁，第 15 行表示它释放了锁，第 16 行释放锁。

第 20-44 行是初始化代码。第 26 行初始化当前锁状态的数组。第 27 行启动 locker 线程，第 28 行增加下一个锁编号。一旦所有 locker 线程被创建，do-od 循环开始执行第 29 行，检查断言。第 30、31 行初始化控制变量，第 32-40 行原子的计算数组节点的和，第 41 行是断言，42 行退出循环。

我们可以将以上两个代码片段放到 lock.h 和 lock.spin 来运行这个模型，运行以下命令:

```
spin -a lock.spin
cc -DSAFETY -o pan pan.c
./pan
```

```
(Spin Version 4.2.5 -- 2 April 2005)
+ Partial Order Reduction
```

```
Full statespace search for:
never claim - (none specified)
assertion violations +
cycle checks - (disabled by -DSAFETY)
invalid end states +
State-vector 40 byte, depth reached 357, errors: 0
564 states, stored
929 states, matched
1493 transitions (= stored+matched)
368 atomic steps
hash conflicts: 0 (resolved)
2.622 memory usage (Mbyte)
unreached in proctype locker
line 18, state 20, "-end-"
(1 of 20 states)
unreached in proctype :i
```

图 E.10: Output for Spinlock Test

将会看成到类似于图 E.10 的输出。正如预期的那样，没有断言错误 (``errors: 0")。

问题 E.1：为什么在 locker 中有一条不可到达的语句？最后，这不是一个全状态的搜索吗?

问题 E.2：这个例子有什么 Promela 代码规范问题?

E.6 Promela 示例: QRCU

最后一个例子是在 Oleg Nesterov 的 QRCU [Nes06a,Nes06b]中实际使用的示例。但是经过修改，以提高 synchronize_qrcu() 的速度。

首先，什么是 QRCU?

QRCU 是 SRCU [McK06]的变体，用于读负载更多，并且极低的延迟的情况。如果没有读者，将在一微秒内检测出来。与其他 RCU 实现相比，有更好的延迟。

qrcu_struct 定义一个 QRCU。象 SRCU 一样 (与其他 RCU 变体不一样)，QRCU 不是全局的，它专注于特定的 qrcu_struct。

qrcu_read_lock() 和 qrcu_read_unlock() 定义了 QRCU 读临界区。相应的 qrcu_struct must 必须放到这些原语中，rcu_read_lock() 的返回值必须传递给 rcu_read_unlock()。

例如：

```
idx = qrcu_read_lock(&my_qrcu_struct);
```

```
/* read-side critical section. */
qrcu_read_unlock(&my_qrcu_struct, idx);
```

synchronize_qrcu() 原语将阻塞，直接所有第一个已经存在的 QRCU 读临界区完成，与 SRCU 的 synchronize_srcu()一样。QRCU 的 synchronize_qrcu() 仅仅需要等待特定 qrcu_struct。

例如，synchronize_qrcu(&your_qrcu_struct) 不必等待更早的 QRCU 读临界区。作为对比，synchronize_qrcu(&my_qrcu_struct)有必要等待，因为它共享相同的 qrcu_struct。

一个 QRCU 的 LINUX 内核补丁已经提供 [McK07b]，但是直到 2008 年四月都还没有被包含进入 LINUX 内核。

```
1 #include "lock.h"
2
3 #define N_QRCU_READERS 2
4 #define N_QRCU_UPDATERS 2
5
6 bit idx = 0;
7 byte ctr[2];
8 byte readerprogress[N_QRCU_READERS];
9 bit mutex = 0;
```

图 E.11: QRCU Global Variables

回到 QRCU 的 Promela 代码，全局变量如图 E.11。这个例子使用锁，因此包含了 lock.h。读者和写者的数量可能使用两个#define 语句定义。idx 变量控制哪一个 ctr 数组用于读者。readerprogress 变量允许断言判断所有读者全部完成(因为直到每一个存在的读者已经结束它们的读临界区时，QRCU 才允许完成更新)。。readerprogress 数组元素有如下的值，这些值包含了读者的状态:

0：还没有开始.

1：处于 QRCU 读临界区.

2：完成 QRCU 读临界区.

最后，mutex 变量用于写者操作的串行化.

```
1 proctype qrcu_reader(byte me)
2 {
3 int myidx;
4
5 do
6 :: 1 ->
7 myidx = idx;
8 atomic {
9 if
10 :: ctr[myidx] > 0 ->
```

```
11 ctr[myidx]++;
12 break
13 :: else -> skip
14 fi
15 }
16 od;
17 readerprogress[me] = 1;
18 readerprogress[me] = 2;
19 atomic { ctr[myidx]-- }
20 }
```

图 E.12: QRCU Reader Process

QRCU 读者处理过程是 qrcu_reader()，如图 E.12。do-od 循环位于第 5-16 行，第 6 行的单行``1" 表示这是一个死循环。第 7 行得到全局索引的当前值，如果它的值不是 0(atomic_inc_not_zero())，则在第 8-15 行原子的递增它 (并退出死循环)。第 17 行标记它处于 RCU 读临界区，第 18 行标记它从临界区退出。第 19 行原子的递减相同的计数器。

```
1 #define sum_unordered \
2 atomic { \
3 do \
4 :: 1 -> \
5 sum = ctr[0]; \
6 i = 1; \
7 break \
8 :: 1 -> \
9 sum = ctr[1]; \
10 i = 0; \
11 break \
12 od; \
13 } \
14 sum = sum + ctr[i]
```

图 E.13: QRCU Unordered Summation

如图 E.13 的 C 预处理宏计算计数器的总和，它模拟弱序的实现。第 2-13 行取得其中一个计数器，第 14 行取得另外的值并计算总和。原子块由一个 do-od 语句组成。这个 do-od 语句 (第 3-12 行) 是不同寻常的，它包含两个无条件分支（第 4、8 行），这导致 Promela 随机的选择两个中的一个 (再次提醒, 全状态空间的搜索导致 Promela 最终构造所有可能的选择)。第一个分支得到第 0 个计数器并设置 i 为 1 (因此第 14 行将获取第一个计数器)。而第二个分支正好相反，它获得第一个计数器并将 i 设置为 0 (因此第 14 行将获得第二个计数器)。

问题 E.3：有更好的方法编写 do-od 语句吗?

```
1 proctype qrcu_updater(byte me)
2 {
3 int i;
4 byte readerstart[N_QRCU_READERS];
5 int sum;
6
7 do
8 :: 1 ->
9
10 /* Snapshot reader state. */
11
12 atomic {
13 i = 0;
14 do
15 :: i < N_QRCU_READERS ->
16 readerstart[i] = readerprogress[i];
17 i++
18 :: i >= N_QRCU_READERS ->
19 break
20 od
21 }
22
23 sum_unordered;
24 if
25 :: sum <= 1 -> sum_unordered
26 :: else -> skip
27 fi;
28 if
29 :: sum > 1 ->
30 spin_lock(mutex);
31 atomic { ctr[!idx]++ }
32 idx = !idx;
33 atomic { ctr[!idx]-- }
34 do
35 :: ctr[!idx] > 0 -> skip
36 :: ctr[!idx] == 0 -> break
37 od;
38 spin_unlock(mutex);
39 :: else -> skip
40 fi;
41
42 /* Verify reader progress. */
```

```
43
44 atomic {
45 i = 0;
46 sum = 0;
47 do
48 :: i < N_QRCU_READERS ->
49 sum = sum + (readerstart[i] == 1 &&
50 readerprogress[i] == 1);
51 i++
52 :: i >= N_QRCU_READERS ->
53 assert(sum == 0);
54 break
55 od
56 }
57 od
58 }
```

图 E.14: QRCU Updater Process

使用 sum_unordered 宏，我们现在可以如图 E.14 所示处理写端过程。写端不确定的重复执行，相应的 do-od 循环位于第 7-57 行。第一次循环首先获得全局 readerprogress 数组的快照，存到局部的 readerstart 数组（第 12-21 行）。这个快照将用于第 53 行的断言。第 23 行调用 sum_unordered，然后，第 24-27 行重新调用 sum_unordered。

如果有必要，在第 28-40 行执行慢速代码，第 30、38 行获得、释放写锁，第 31、33 行转换索引，第 34-37 行等待所有的读者完成。

第 44-56 行比较 readerprogress 数组与的 readerstart 数组的当前值，如果任何读者在写者仍然在运行之前开始的话，将产生一个断言。

问题 E.4：为什么第 12-21 行以及第 44-56 行是原子块？

问题 E.5：第 24-27 行的重新计算总和语句真的是必要的吗?

```
1 init {
2 int i;
3
4 atomic {
5 ctr[idx] = 1;
6 ctr[!idx] = 0;
7 i = 0;
8 do
9 :: i < N_QRCU_READERS ->
10 readerprogress[i] = 0;
11 run qrcu_reader(i);
```

```
12 i++
13 :: i >= N_QRCU_READERS -> break
14 od;
15 i = 0;
16 do
17 :: i < N_QRCU_UPDATERS ->
18 run qrcu_updater(i);
19 i++
20 :: i >= N_QRCU_UPDATERS -> break
21 od
22 }
23 }
```

图 E.15: QRCU Initialization Process

其他的代码是初始化块，如图 E.15。这个块简单的在第 5-6 行初始化计数器。在第 7-14 行创建读者线程，在第 15-21 行创建写者线程，为了减少状态空间，这些代码都处于原子块中。

E.6.1 运行 QRCU 示例

要运行 QRCU 示例，将前一节的代码片段组合到单个文件 qrcu.spin 中。并且将 spin_lock() 和 spin_unlock() 的定义放到文件 lock.h 中。然后使用以下的命令构建并运行 QRCU：

```
spin -a qrcu.spin
cc -DSAFETY -o pan pan.c
./pan
```

表 E.2: Memory Usage of QRCU Model

updaters	readers	# states	MB
1	1	376	2.6
1	2	6,177	2.9
1	3	82,127	7.5
2	1	29,399	4.5
2	2	1,071,180	75.4

2	3	33,866,700	2,715.2
3	1	258,605	22.3
3	2	169,533,000	14,979.9

该模型的所有情形如表所示。现在，运行三个读者和三个写者是可以的，但是，这将需要上 T 的内存。因此，该怎么办？下面是一些可能的方法：

看看较少的读者和写者是否足够

手动进行正确性校验

使用更多有用的工具.

分而治之.

后面几节描述这几种方法.

E.6.2 到底需要多少读者和写者?

一个方法是小心的检查 Promela 代码 qrcu_updater()，请注意，全局状态变化仅仅在锁的保护下发生。因此，在同一时刻仅仅一个写者可能修改状态。

这种情况下，读者是明确的，每一个读者仅仅执行一个单一的读临界区然后退出。可能会争论：可用的读者数量是受限的。

E.6.3 可选方法: 正确性校验

一个非正式的校验 [McK07b] 如下：

对于 synchronize_qrcu() 过早退出的情况，在 synchronize_qrcu()执行时，要求至少一个读者正在运行。

在此期间，与读者相应的计数器至少是 1。

synchronize_qrcu() 代码强制其中一个计数器至少为 1。

因此，在任一给定点，任意一个计数器至少为 2，或者所有计数都大于 0。

但是，synchronize_qrcu() 快速代码只能在指定时间读取计数器的值。

There can be at most one reader persisting through such a race condition, as otherwise the sum would be two or greater, which would cause the updater to take the slowpath.

But if the race occurs on the fastpath's first read of the counters, and then again on its second read, there have to have been two counter flips.

Because a given updater flips the counter only once, and because the update-side lock prevents a pair of updaters from concurrently flipping the counters, the only way

that the fastpath code can race with a flip twice is if the first updater completes.

But the first updater will not complete until after all pre-existing readers have completed.

Therefore, if the fastpath races with a counter flip twice in succession, all pre-existing readers must have completed, so that it is safe to take the fastpath.

Of course, not all parallel algorithms have such simple proofs. In such cases, it may be necessary to enlist more capable tools.

E.6.4 可选方法: 更多工具

虽然 Promela 和 Spin 十分有用，还存在更多工具，特别是用于验证硬件。这意味着：如果将您的算法转换为硬件设计 VHDL 语言，它通常用于非常底层的并行算法，就可以将这些工具用于您的代码。但是，这些工具十分昂贵。

虽然商业的多处理器的出现可能导致自由软件减少状态空间，但是现在还没有什么大的帮助。

另外，支持近似的 Spin 搜索特性需要固定数量的内存，但是，在验证并行算法时，我从来不让自己相信这种近似的东西。

另外的方法是分而治之。

E.6.5 可选方法: 分而治之

通常情况下，将大的并行算法分成小的片段是可能的，这样可以分别的证明它们。例如，一个 10-billion-state 模型可能被分成两个 100,000-state 模型。讨论这种方法不会使 Promela 这样的工具变得更简单，但是这会使您的算法更易于理解。

E.7 Promela Parable: dynticks 和可抢占 RCU

在 2008 初，一个 RCU 的可抢占变种被接收到 LINUX 主分支，用来支持实时任务。类似于-rt 补丁中的 RCU 实现 [Mol05]。实时任务需要可抢占 RCU，因为原有的 RCU 实现在 RCU 读临界区中禁止了抢占，这导致过度的延迟。

但是，原有的-rt 实现有一个缺点 (在附录中描述)：每一个周期都需要每一个 CPU 上的工作全部完成，即使 CPU 处于低电源的``dynticks-idle" 状态，这种状态下是不会执行 RCU 读临界区的。dynticks-idle 的想法是：空闲 CPUs 应当完全的关闭电源以节省能量。简短的说，在最近的 LINUX 内核中，可抢占 RCU 可能禁止 energy-conservation 功能。虽然 Josh Triplett 和 Paul McKenney 已经讨

论一些方法以允许 CPUs 在 RCU 周期内保留 low-power 状态(因此保留 LINUX 内核节能的功能)，直到 Steve Rostedt 在-rt 补丁中实现带抢占的 RCU 的新的 dyntick 前，事情还没有达到紧要关头。

这个组合导致一个 Steve 的系统在 boot 时挂起，因此在十月份，Paul 为可抢占 RCU 编写了一个 dynticks-friendly 的修改版本。Steve 编写 rcu_irq_enter() 和 rcu_irq_exit() 接口，这两个接口在 irq_enter() 和 irq_exit()函数中调用。为了允许 RCU 可靠的处理 dynticks，rcu_irq_enter() 和 rcu_irq_exit() 函数是必须的。由于这些适当的修改，Steve 的系统能正常的启动了。但是 Paul 假定我们不能一次性搞定代码，因此继续检查代码。

Paul 从 2007 年十月到 2008 年二月反复审查代码，几乎总是能够找到至少一个 BUG。其中一个例子是： Paul 甚至在还没有真正认识到一个 BUG 是虚幻的 BUG 之前，就编码并测试一个修复。

在二月底的时候，Paul 对此已经有点疲倦了。因此他决定取得 Promela 和 spin [Hol03]的帮助，如附录 D.4 中所描述的。随后将展示七个 Promela 模型，假设有 40GB 的状态空间主内存。

更重要的是，Promela 和 Spin 发现了一个很难捉摸的 BUG!

问题 E.6：Yeah, 太好了！现在, 如果没有 40GB 物理内存的机器，该怎么做???

出现拥有更小的状态空间的简单的、快速的算法就更好了。

E.7. 1 节给出了可抢占 RCU 的 dynticks 接口概要, E.7. 2 节, 和 E.7.3 节列出了相应的教程。

E.7.1 可抢占 RCU 和 dynticks 介绍

每 CPU 的 dynticks_progress_counter 变量是 dynticks 和可抢占 RCU 之间的重要接口。这个变量可拥有偶数值，而不管相应的 CPU 是否处于 dynticks-idle 模式，否则为奇数值。CPU 由于以下三个原因退出 dynticks-idle 模式：

开始运行一个任务,

当进入最外层的嵌套中断处理

进入 NMI 处理.

可抢占的 RCU 的 grace-period 机制采样 dynticks_progress_counter 变量的值，以确定 dynticks-idle CPU 什么时候可以被安全的忽略。

随后三节给出了任务接口、中断/NMI 接口的概要，以及 grace-period 机制如何使用 dynticks_progress_counter 变量。

E.7.1.1 任务接口

当一个特定的 CPU 由于没有更多任务需要运行而进入 dynticks-idle 模式时，它调用 rcu_enter_nohz()：

```
1 static inline void rcu_enter_nohz(void)
2 {
3     mb();
4     __get_cpu_var(dynticks_progress_counter)++;
5     WARN_ON(__get_cpu_var(dynticks_progress_counter) & 0x1);
6 }
```

这个函数简单的递增 dynticks_progress_counter 并检查结果是否为偶数，但是它首先执行一个内存屏障，以确保任何其他 CPU 看见 dynticks_progress_counter 的新值前，将先看到前面的任何 RCU 读临界区已经完成。

类似的，当一个处于 dynticks-idle 模式的 CPU 准备执行一个新的任务时，它调用 rcu_exit_nohz：

```
1 static inline void rcu_exit_nohz(void)
2 {
3     __get_cpu_var(dynticks_progress_counter)++;
4     mb();
5     WARN_ON(!(__get_cpu_var(dynticks_progress_counter) &
6                 0x1));
7 }
```

这个函数再次递增 dynticks_progress_counter，随后执行一个内存屏障，以确保任何其他 CPU 看到随后的 RCU 读临界区的结果前，将看到递增后的 dynticks_progress_counter 值。最后，rcu_exit_nohz() 检查递增后的结果是否是一个奇数值。

rcu_enter_nohz() 和 rcu_exit_nohz 函数处理 CPU 通过执行任务而进入或者退出 dynticks-idle 模式的情形。

E.7.1.2 中断接口

rcu_irq_enter() 和 rcu_irq_exit() 函数处理进入、退出中断/NMI 的情况。当然，嵌套中断也必须正确的计数。可能的嵌套中断由第二个每 CPU 变量处理：rcu_update_flag，在进入一个中断或者 NMI(rcu_irq_enter())时递增它的值，并在退出时递减它的值 (rcu_irq_exit())。另外，已经存在的 in_interrupt() 原语用来区分是处于最外层中断还是处于嵌套中断。

进入中断由下面的 rcu_irq_enter 处理：

```
1 void rcu_irq_enter(void)
```

```
2 {
3     int cpu = smp_processor_id();
4
5     if (per_cpu(rcu_update_flag, cpu))
6         per_cpu(rcu_update_flag, cpu)++;
7     if (!in_interrupt() &&
8         (per_cpu(dynticks_progress_counter,
9                  cpu) & 0x1) == 0) {
10        per_cpu(dynticks_progress_counter, cpu)++;
11        smp_mb();
12        per_cpu(rcu_update_flag, cpu)++;
13    }
14 }
```

第 3 行得到当前 CPU 号，如果 rcu_update_flag 值不为 0，则第 5、6 行 递增嵌套计数。第 7-9 行检查我们是否处于最外层中断，如果是这样，第 10 行递增 dynticks_progress_counter，第 11 行执行一个内存屏障，第 12 行递增 rcu_update_flag。如同 rcu_exit_nohz()，内存屏障确保任何其他 CPU 看到中断处理函数中的 RCU 读临界区（紧随 rcu_irq_enter()）的效果前，前先看到 dynticks_progress_counter 的值。

问题 E.7：为什么不简单的递增 rcu_update_flag，然后仅仅在后面递增 dynticks_progress_counter???

问题 E.8：如果第 7 行发现已经处于最外层中断，我们不总是递增 dynticks_progress_counter?

类似的，中断退出由 rcu_irq_exit()处理：

```
1 void rcu_irq_exit(void)
2 {
3     int cpu = smp_processor_id();
4
5     if (per_cpu(rcu_update_flag, cpu)) {
6         if (--per_cpu(rcu_update_flag, cpu))
7             return;
8         WARN_ON(in_interrupt());
9         smp_mb();
10        per_cpu(dynticks_progress_counter, cpu)++;
11        WARN_ON(per_cpu(dynticks_progress_counter,
12                        cpu) & 0x1);
13    }
14 }
```

第 3 行获得当前 CPU 号。第 5 行检查 rcu_update_flag 是否非 0，如果不是则立即返回。否则，第 6-12 行开始运行。第 6 行递减 rcu_update_flag，如果结

果非 0 则返回。第 8 行检查我们真的退出最外层中断。第 9 行执行一个内存屏障，第 10 行递增 dynticks_progress_counter，第 11 、 12 行检查这个变量现在是偶数。与 rcu_enter_nohz()一样，内存屏障确保任何其他 CPU 在看到 dynticks_progress_counter 递增结果前，将看到之前的 RCU 读临界区的效果。

这两节描述了进入或者退出 dynticks-idle （通过任务或者中断）时，如何维护 dynticks_progress_counter 变量。后面的章节描述这个变量如何被可抢占 RCU 使用。

E.7.1.3 Grace-Period 接口

四个可抢占 RCU grace-period 状态如图 D.63 所示（附录 D.4），仅仅 rcu_try_flip_waitack_state() 和 rcu_try_flip_waitmb_state() 状态需要等待其他 CPUs 响应。

当然，如果特定 CPU 处于 dynticks-idle 状态，我们不应当等待它。因此，在进入这两种状态以前，以前的状态获得每 CPU 的 dynticks_progress_counter 变量快照，将快照放到另一个每 CPU 变量中：cu_dyntick_snapshot。这是通过调用 dyntick_save_progress_counter 实现的，如下：

```
1 static void dyntick_save_progress_counter(int cpu)
2 {
3     per_cpu(rcu_dyntick_snapshot, cpu) =
4         per_cpu(dynticks_progress_counter, cpu);
5 }
```

rcu_try_flip_waitack_state() 状态调用 rcu_try_flip_waitack_needed()，如下：

```
1 static inline int
2 rcu_try_flip_waitack_needed(int cpu)
3 {
4     long curr;
5     long snap;
6
7     curr = per_cpu(dynticks_progress_counter, cpu);
8     snap = per_cpu(rcu_dyntick_snapshot, cpu);
9     smp_mb();
10    if ((curr == snap) && ((curr & 0x1) == 0))
11        return 0;
12    if ((curr - snap) > 2 || (snap & 0x1) == 0)
13        return 0;
14    return 1;
15 }
```

第 7 、 8 行取出 dynticks_progress_counter 的当前值和快照的值。内存屏

障确保 rcu_try_flip_waitzero_state 随后的计数检查。如果自从快照获取以来，已经在 dynticks-idle 状态存在过，则返回 0（意味着没有必须与指定的 CPU 进行通信）。第 10、11 行返回 0。类似的，如果 CPU 最初是 dynticks-idle 状态或者它已经完全经历过一次 dynticks-idle 状态，第 12、13 行也返回 0。如果这些条件都不满足，则第 14 行返回，意味着 CPU 需要得到响应。

作为其中的一部分，rcu_try_flip_waitmb_state 调用 rcu_try_flip_waitmb_needed()，如下：

```
 1 static inline int
 2 rcu_try_flip_waitmb_needed(int cpu)
 3 {
 4     long curr;
 5     long snap;
 6
 7     curr = per_cpu(dynticks_progress_counter, cpu);
 8     snap = per_cpu(rcu_dyntick_snapshot, cpu);
 9     smp_mb();
10     if ((curr == snap) && ((curr & 0x1) == 0))
11         return 0;
12     if (curr != snap)
13         return 0;
14     return 1;
15 }
```

这与 rcu_try_flip_waitack_needed 十分类似，不同之处位于第 12 、 13 行，因为任何进入或者退出 dynticks-idle 状态的事件都执行了 rcu_try_flip_waitmb_state()需要的内存屏障。

现在我们已经看到了所有在 RCU 和 dynticks-idle 之间调用的代码。下一节构建 Promela 模型来验证这些代码。

问题 E.9：您能找出本节中的 BUGs 吗?

E.7.2 验证可抢占 RCU 和 dynticks

这一节一步一步的开发一个用于 dynticks 和 RCU 之间接口的 Promela 模型，每一节都举例说明每一个步骤，先从任务级代码开始，添加断言、中断、以及 NMIs。

E.7.2.1 基本模型

本节将进程级的 dynticks 进入、退出代码以及周期性处理代码修改为

Promela [Hol03]。我们从 2.6.25-rc4 内核的 rcu_exit_nohz() 和 rcu_enter_nohz() 开始，将它们修改成单个 Promela 任务，这个进入、退出 dynticks-idle 模式的模型如下：

```
 1 proctype dyntick_nohz()
 2 {
 3     byte tmp;
 4     byte i = 0;
 5
 6     do
 7     :: i >= MAX_DYNTICK_LOOP_NOHZ -> break;
 8     :: i < MAX_DYNTICK_LOOP_NOHZ ->
 9        tmp = dynticks_progress_counter;
10        atomic {
11           dynticks_progress_counter = tmp + 1;
12           assert((dynticks_progress_counter & 1) == 1);
13        }
14        tmp = dynticks_progress_counter;
15        atomic {
16           dynticks_progress_counter = tmp + 1;
17           assert((dynticks_progress_counter & 1) == 0);
18        }
19        i++;
20     od;
21 }
```

第 6 、 20 行定义一个循环。一旦循环计数器 i 已经超过 MAX_DYNTICK_LOOP_NOHZ 限制，第 7 行就退出循环。第 8 行告诉循环结构执行第 9-19 行。由于第 7 行和第 8 行的条件是相互排斥的，因此 Promela 不会随机选择。第 9 、 11 行模仿 rcu_exit_nohz()的非原子递增 dynticks_progress_counter，而 第 12 行模仿 WARN_ON()。原子结构简单的减少 Promela 状态空间，严格的说，WARN_ON() 不是算法的一部分。同样的，第 14-18 模仿 rcu_enter_nohz 中的递增和 WARN_ON()。最后，第 19 行递增循环计数器。

每一次循环都模仿一次 CPU 退出 dynticks-idle 模式的操作 (如，开始执行一个任务)，然后重新进入 dynticks-idle 模式 (如，任务被阻塞了)。

问题 E.10：为什么 rcu_exit_nohz() 和 rcu_enter_nohz() 中的内存屏障没有被模仿?

问题 E.11：有一个小疑问，不什么模仿 rcu_exit_nohz()后紧跟着模仿 rcu_enter_nohz()? 先模仿进入再模仿退出不是更自然吗?

下一步是模仿 RCU 的周期性处理接口。为此，我们必须模仿 dyntick_save_progress_counter(), rcu_try_flip_waitack_needed(), rcu_try_flip_waitmb_needed()，以及部分模拟 rcu_try_flip_waitack() 和 rcu_try_flip_waitmb()，它们都来自于 2.6.25-rc4 内核。下面的 grace_period() Promela 过程模拟这些功能。

```
1 proctype grace_period()
2 {
3     byte curr;
4     byte snap;
5
6     atomic {
7         printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NOHZ);
8         snap = dynticks_progress_counter;
9     }
10    do
11    :: 1 ->
12        atomic {
13            curr = dynticks_progress_counter;
14            if
15            :: (curr == snap) && ((curr & 1) == 0) ->
16                break;
17            :: (curr - snap) > 2 || (snap & 1) == 0 ->
18                break;
19            :: 1 -> skip;
20            fi;
21        }
22    od;
23    snap = dynticks_progress_counter;
24    do
25    :: 1 ->
26        atomic {
27            curr = dynticks_progress_counter;
28            if
29            :: (curr == snap) && ((curr & 1) == 0) ->
30                break;
31            :: (curr != snap) ->
32                break;
33            :: 1 -> skip;
34            fi;
35        }
36    od;
37 }
```

第 6-9 行输出循环限制并模仿 rcu_try_flip_idle() 中的一行代码，这行代码

调用 dyntick_save_progress_counter()，它获得当前 CPU 的 dynticks_progress_counter 变量的一个快照。这两行原子的执行以减少状态空间。

第 10-22 行模仿 rcu_try_flip_waitack()中的相应代码，这些代码调用 rcu_try_flip_waitack_needed()。这个循环模仿周期状态机等待每一个 CPU 应答的动作，但是仅仅是其中与 dynticks-idle CPUs 交互的部分。

第 23 行模仿 rcu_try_flip_waitzero() 中的一行，这一行调用 dyntick_save_progress_counter()，再一次获取 CPU 的 dynticks_progress_counter 变量的快照。

最后，第 24-36 模仿 rcu_try_flip_waitack() 相关的代码，这些代码调用 rcu_try_flip_waitack_needed()。这个循环模仿周期定时状态机等待每一个 CPU 执行一个内存屏障，它也仅仅是与 dynticks-idle CPUs 交互的一部分。

问题 E.12：稍等! 在 Linux 内核中，dynticks_progress_counter 和 rcu_dyntick_snapshot 都 per-CPU 变量。但是为什么被模拟成一个单一的全局变量?

这个基本的模型 (dyntickRCU-base.spin)，在使用 runspin.sh 脚本运行时，生成 691 个状态，不会生成任何错误。下一节添加一些安全性校验。

E.7.2.2 安全性校验

安全的 RCU 实现必须允许一个周期在任何在周期之前启动的 RCU 读者完成之前结束。这是通过一个 grace_period_state 变量来模仿的，它可以有三个值：

```
1 #define GP_IDLE       0
2 #define GP_WAITING    1
3 #define GP_DONE       2
4 byte grace_period_state = GP_DONE;
```

grace_period() 过程设置这个变量，如下:

```
1 proctype grace_period()
2 {
3     byte curr;
4     byte snap;
5
6     grace_period_state = GP_IDLE;
7     atomic {
8         printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NOHZ);
9         snap = dynticks_progress_counter;
10        grace_period_state = GP_WAITING;
11    }
12    do
```

```
13     :: 1 ->
14        atomic {
15           curr = dynticks_progress_counter;
16           if
17           :: (curr == snap) && ((curr & 1) == 0) ->
18              break;
19           :: (curr - snap) > 2 || (snap & 1) == 0 ->
20              break;
21           :: 1 -> skip;
22           fi;
23        }
24     od;
25     grace_period_state = GP_DONE;
26     grace_period_state = GP_IDLE;
27     atomic {
28        snap = dynticks_progress_counter;
29        grace_period_state = GP_WAITING;
30     }
31     do
32     :: 1 ->
33        atomic {
34           curr = dynticks_progress_counter;
35           if
36           :: (curr == snap) && ((curr & 1) == 0) ->
37              break;
38           :: (curr != snap) ->
39              break;
40           :: 1 -> skip;
41           fi;
42        }
43     od;
44     grace_period_state = GP_DONE;
45 }
```

第 6, 10, 25, 26, 29, 和 44 行更新这个变量，并允许 dyntick_nohz() 过程验证 RCU 安全属性。

问题 E.13：在第 25、26 行，有一个 back-to-back 的变化，我们如何确信第 25 行的变化不会丢失?

dyntick_nohz() Promela 过程实现验证，如下：

```
1 proctype dyntick_nohz()
2 {
3     byte tmp;
4     byte i = 0;
```

```
 5    bit old_gp_idle;
 6
 7    do
 8    :: i >= MAX_DYNTICK_LOOP_NOHZ -> break;
 9    :: i < MAX_DYNTICK_LOOP_NOHZ ->
10       tmp = dynticks_progress_counter;
11       atomic {
12          dynticks_progress_counter = tmp + 1;
13          old_gp_idle = (grace_period_state == GP_IDLE);
14          assert((dynticks_progress_counter & 1) == 1);
15       }
16       atomic {
17          tmp = dynticks_progress_counter;
18          assert(!old_gp_idle ||
19                   grace_period_state != GP_DONE);
20       }
21       atomic {
22          dynticks_progress_counter = tmp + 1;
23          assert((dynticks_progress_counter & 1) == 0);
24       }
25       i++;
26    od;
27 }
```

如果在任务开始执行的时候，grace_period_state 的值是 GP_IDLE，就在第 13 行设置一个新 old_gp_idle 标志，如果在任何执行过程中，grace_period_state 变量已经变为 GP_DONE，则第 18 、 19 行的断言将触发。因为 RCU 读临界区不能越过一个完整的周期。

在运行 runspin.sh 脚本时，最后的模型(dyntickRCU-base-s.spin)产生 964 个状态，并且没有错误。下一节将进行生存期验证。

E.7.2.3 生存期验证

虽然生存期难于验证，但是也有一个简单的技巧。第一步是使 dyntick_nohz() 通过一个 dyntick_nohz_done 变量标示它已经执行完，如下第 27 行所示：

```
1 proctype dyntick_nohz()
2 {
3    byte tmp;
4    byte i = 0;
5    bit old_gp_idle;
6
7    do
8    :: i >= MAX_DYNTICK_LOOP_NOHZ -> break;
```

```
 9     :: i < MAX_DYNTICK_LOOP_NOHZ ->
10        tmp = dynticks_progress_counter;
11        atomic {
12           dynticks_progress_counter = tmp + 1;
13           old_gp_idle = (grace_period_state == GP_IDLE);
14           assert((dynticks_progress_counter & 1) == 1);
15        }
16        atomic {
17           tmp = dynticks_progress_counter;
18           assert(!old_gp_idle ||
19                     grace_period_state != GP_DONE);
20        }
21        atomic {
22           dynticks_progress_counter = tmp + 1;
23           assert((dynticks_progress_counter & 1) == 0);
24        }
25        i++;
26     od;
27     dyntick_nohz_done = 1;
28 }
```

通过这个变量，我们可以添加一个断言到 grace_period() ，检查不必要的阻塞:

```
 1 proctype grace_period()
 2 {
 3     byte curr;
 4     byte snap;
 5     bit shouldexit;
 6
 7     grace_period_state = GP_IDLE;
 8     atomic {
 9        printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NOHZ);
10        shouldexit = 0;
11        snap = dynticks_progress_counter;
12        grace_period_state = GP_WAITING;
13     }
14     do
15     :: 1 ->
16        atomic {
17           assert(!shouldexit);
18           shouldexit = dyntick_nohz_done;
19           curr = dynticks_progress_counter;
20           if
21           :: (curr == snap) && ((curr & 1) == 0) ->
22              break;
```

```
23            :: (curr - snap) > 2 || (snap & 1) == 0 ->
24               break;
25            :: else -> skip;
26            fi;
27         }
28      od;
29      grace_period_state = GP_DONE;
30      grace_period_state = GP_IDLE;
31      atomic {
32         shouldexit = 0;
33         snap = dynticks_progress_counter;
34         grace_period_state = GP_WAITING;
35      }
36      do
37      :: 1 ->
38         atomic {
39            assert(!shouldexit);
40            shouldexit = dyntick_nohz_done;
41            curr = dynticks_progress_counter;
42            if
43            :: (curr == snap) && ((curr & 1) == 0) ->
44               break;
45            :: (curr != snap) ->
46               break;
47            :: else -> skip;
48            fi;
49         }
50      od;
51      grace_period_state = GP_DONE;
52 }
```

我们在第 5 行添加 shouldexit 变量，在第 10 行将其初始化为 0。第 17 行验证它没有被设置。第 18 行将 shouldexit 设置为 dyntick_nohz_done ，dyntick_nohz_done 的值由 dyntick_nohz()维护。如果在 dyntick_nohz()完全执行完后，我们试图执行超过一次 wait-for-counter-flip-acknowledgement 循环，则将触发这个断言。最后，如果 dyntick_nohz()执行完毕，没有更多的状态变化使循环退出，这样导致死循环。

第 32、39 和 40 行以类似第二个循环（内存屏障）那样运行。

但是，运行这个模型 (dyntickRCU-base-sl-busted.spin) 将导致失败，第 23 行检测错误的变量为偶数。这样，spin 输出一个``trail" 文件 (dyntickRCU-base-sl-busted.spin.trail)，该文件记录下导致错误的状态的顺序。使用 spin -t -p -g -l dyntickRCU-base-sl-busted.spin 命令输出这些状态顺序，输出执

行的语句以及变量的值(dyntickRCU-base-sl-busted.spin.trail.txt)。注意输出的行号并不与上面列出的行号一致，因为实际的函数在一个单一的文件中。但是，行号与整个模型文件匹配 (dyntickRCU-base-sl-busted.spin)。

我们看到：dyntick_nohz() 过程在第 34 步完成(搜索 ``34:")，但是 grace_period() 过程没有退出循环。curr 的值是 6 (参见 第 35 步) 而 snap 的值是 5 (参见第 17 步)。因此第 21 行的第一个条件不满足，因为 curr != snap, 第 23 行的第二个条件也不满足，因为 snap 是奇数，而 curr 仅仅比 snap 大于 1。

因此，这两个条件中有一个是不正确的。参考 rcu_try_flip_waitack_needed() 第一个条件的注释块：

If the CPU remained in dynticks mode for the entire time and didn't take any interrupts, NMIs, SMIs, or whatever, then it cannot be in the middle of an rcu_read_lock(), so the next rcu_read_lock() it executes must use the new value of the counter. So we can safely pretend that this CPU already acknowledged the counter.

第一个条件是满足的，因数 if curr == snap 并且 if curr 是偶数，那么相应的 CPU 已经处于 dynticks-idle 模式。因为我们看看第二个条件的注释块：

If the CPU passed through or entered a dynticks idle phase with no active irq handlers, then, as above, we can safely pretend that this CPU already acknowledged the counter.

条件的第一部分是正确的，因为如果 curr 和 snap 相差 2, 那么至少经过一次奇数状态，相应的也经历过一次 dynticks-idle 阶段。但是，条件的第二部分表示开始于 dynticks-idle 模式，但是还没有结束于该模式。因此，我们需要测试 curr 值而不是 snap 值。

正确的 C 代码如下:

```
 1 static inline int
 2 rcu_try_flip_waitack_needed(int cpu)
 3 {
 4     long curr;
 5     long snap;
 6
 7     curr = per_cpu(dynticks_progress_counter, cpu);
 8     snap = per_cpu(rcu_dyntick_snapshot, cpu);
 9     smp_mb();
10     if ((curr == snap) && ((curr & 0x1) == 0))
11         return 0;
12     if ((curr - snap) > 2 || (curr & 0x1) == 0)
13         return 0;
14     return 1;
15 }
```

第 10-13 可以组合并被简化，结果如下。类似的简化也可用于 rcu_try_flip_waitmb_needed。

```
1 static inline int
2 rcu_try_flip_waitack_needed(int cpu)
3 {
4     long curr;
5     long snap;
6
7     curr = per_cpu(dynticks_progress_counter, cpu);
8     snap = per_cpu(rcu_dyntick_snapshot, cpu);
9     smp_mb();
10    if ((curr - snap) >= 2 || (curr & 0x1) == 0)
11        return 0;
12    return 1;
13 }
```

修改后的模型 (dyntickRCU-base-sl.spin) 产生了正确的验证结果，该结果包含 661 个状态并且没有错误。但是，第一个版本的生命期验证是没有意义的，因为生命期验证本身有一个 BUG。

我们已经成功的骓了安全性和生命期条件，但是仅仅针对了进程运行和阻塞。我们也需要处理中断，这个任务在下一节中进行。

E.7.2.4 中断

在 Promela 中，有两种方法模仿中断:

使用 C-preprocessor 技巧，在 dynticks_nohz()过程的每一条语句之间插入中断处理函数，或者在一个独立的过程中模仿中断处理函数。

看起来第二种方法的状态空间更小， 虽然需要以某种方式原子的运行中断处理函数。

幸运的是 Promela 允许原子语句。这个技巧允许我们使用一个中断处理标志，并重新编码 dynticks_nohz()，并原子的检查这个标志，仅仅在标志没有设置的时候才执行 dynticks_nohz()。这可以使用 C 预处理宏实现。

```
1 #define EXECUTE_MAINLINE(label, stmt) \
2 label: skip; \
3       atomic { \
4           if \
5           :: in_dyntick_irq -> goto label; \
6           :: else -> stmt; \
7           fi; \
8       } \
```

可能的使用方法如下：

```
EXECUTE_MAINLINE(stmt1,
                 tmp = dynticks_progress_counter)
```

第二行创建一个特定的标签语句，第 3-8 行是一个原子块，它测试 in_dyntick_irq 变量，如果这个变量被设置 (表示中断正在运行)，就退出分支并跳回标签，否则，第 6 行执行特定的语句。总的效果就是在中断运行时，就停止执行，这正是我们需要的结果。

E.7.2.5 验证中断处理函数

第一步是将 dyntick_nohz() 转换成 EXECUTE_MAINLINE() 形式，如下：

```
 1 proctype dyntick_nohz()
 2 {
 3     byte tmp;
 4     byte i = 0;
 5     bit old_gp_idle;
 6
 7     do
 8     :: i >= MAX_DYNTICK_LOOP_NOHZ -> break;
 9     :: i < MAX_DYNTICK_LOOP_NOHZ ->
10         EXECUTE_MAINLINE(stmt1,
11             tmp = dynticks_progress_counter)
12         EXECUTE_MAINLINE(stmt2,
13             dynticks_progress_counter = tmp + 1;
14             old_gp_idle = (grace_period_state == GP_IDLE);
15             assert((dynticks_progress_counter & 1) == 1))
16         EXECUTE_MAINLINE(stmt3,
17             tmp = dynticks_progress_counter;
18             assert(!old_gp_idle ||
19                     grace_period_state != GP_DONE))
20         EXECUTE_MAINLINE(stmt4,
21             dynticks_progress_counter = tmp + 1;
22             assert((dynticks_progress_counter & 1) == 0))
23         i++;
24     od;
25     dyntick_nohz_done = 1;
26 }
```

请注意：当一组语句象第 11-14 行那样被传递给 EXECUTE_MAINLINE()时，该组内的所有语句都是原子的。

问题 E.14：如果在单个 EXECUTE_MAINLINE()中的语句想要非原子执行，应该怎么做?

问题 E.15: 如果 dynticks_nohz() 过程有 ``if" 或者 ``do" 语句, 这些语句应当在哪里非原子性的执行?

下一步是编写 dyntick_irq() 过程以模仿中断处理函数:

```
1 proctype dyntick_irq()
2 {
3     byte tmp;
4     byte i = 0;
5     bit old_gp_idle;
6
7     do
8     :: i >= MAX_DYNTICK_LOOP_IRQ -> break;
9     :: i < MAX_DYNTICK_LOOP_IRQ ->
10        in_dyntick_irq = 1;
11        if
12        :: rcu_update_flag > 0 ->
13             tmp = rcu_update_flag;
14           rcu_update_flag = tmp + 1;
15        :: else -> skip;
16        fi;
17        if
18        :: !in_interrupt &&
19           (dynticks_progress_counter & 1) == 0 ->
20           tmp = dynticks_progress_counter;
21           dynticks_progress_counter = tmp + 1;
22           tmp = rcu_update_flag;
23           rcu_update_flag = tmp + 1;
24        :: else -> skip;
25        fi;
26        tmp = in_interrupt;
27        in_interrupt = tmp + 1;
28        old_gp_idle = (grace_period_state == GP_IDLE);
29        assert(!old_gp_idle || grace_period_state != GP_DONE);
30        tmp = in_interrupt;
31        in_interrupt = tmp - 1;
32        if
33        :: rcu_update_flag != 0 ->
34           tmp = rcu_update_flag;
35           rcu_update_flag = tmp - 1;
36           if
37           :: rcu_update_flag == 0 ->
38              tmp = dynticks_progress_counter;
39              dynticks_progress_counter = tmp + 1;
40           :: else -> skip;
```

```
41            fi;
42        :: else -> skip;
43        fi;
44        atomic {
45            in_dyntick_irq = 0;
46            i++;
47        }
48    od;
49    dyntick_irq_done = 1;
50 }
```

第 7-48 行的循环模仿多达 MAX_DYNTICK_LOOP_IRQ 个中断，第 8 和 9 行构成循环条件，第 45 行递增控制变量。第 10 行告诉 dyntick_nohz()：中断正在运行，第 45 行告诉 dyntick_nohz() 中断已经处理完毕。第 49 行用来生命期验证。这与 dyntick_nohz()是一致的。

问题 E.16：为什么第 45 和 46 行是原子的执行的(the in_dyntick_irq = 0; and the i++;) ?

第 11-25 行模仿 rcu_irq_enter()，第 26 和 27 行模仿__irq_enter()。第 28 和 29 进行安全性验证，这与 dynticks_nohz()是一致的。第 30 和 31 模仿 __irq_exit()，最后，第 32-43 行模仿 rcu_irq_exit()。

问题 E.17：dynticks_irq()过程不能模仿什么中断属性?

grace_period 过程如下：

```
1 proctype grace_period()
2 {
3     byte curr;
4     byte snap;
5     bit shouldexit;
6
7     grace_period_state = GP_IDLE;
8     atomic {
9         printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NOHZ);
10        printf("MDLI = %d\n", MAX_DYNTICK_LOOP_IRQ);
11        shouldexit = 0;
12        snap = dynticks_progress_counter;
13        grace_period_state = GP_WAITING;
14    }
15    do
16    :: 1 ->
17        atomic {
18            assert(!shouldexit);
19            shouldexit = dyntick_nohz_done && dyntick_irq_done;
```

```
20            curr = dynticks_progress_counter;
21            if
22            :: (curr - snap) >= 2 || (curr & 1) == 0 ->
23               break;
24            :: else -> skip;
25            fi;
26         }
27      od;
28      grace_period_state = GP_DONE;
29      grace_period_state = GP_IDLE;
30      atomic {
31         shouldexit = 0;
32         snap = dynticks_progress_counter;
33         grace_period_state = GP_WAITING;
34      }
35      do
36      :: 1 ->
37         atomic {
38            assert(!shouldexit);
39            shouldexit = dyntick_nohz_done && dyntick_irq_done;
40            curr = dynticks_progress_counter;
41            if
42            :: (curr != snap) || ((curr & 1) == 0) ->
43               break;
44            :: else -> skip;
45            fi;
46         }
47      od;
48      grace_period_state = GP_DONE;
49 }
```

grace_period()的实现非常简单。唯一的变化是在第 10 行增加一个新的 interrupt-count 参数，改变 第 19 和 39 行，添加新的 dyntick_irq_done 变量用于生命期检查，当然在第 22 行和第 42 行也进行了优化。

这个模型 (dyntickRCU-irqnn-ssl.spin) 产生一个正确的验证结果，大约在半 M 的状态空间。但是，这个版本的模型没有处理嵌套中断。下一节将处理这个主题。

E.7.2.6 验证嵌套中断处理

可以将 dyntick_irq()的循环体分割，以模仿嵌套中断，如下：

```
1 proctype dyntick_irq()
2 {
```

```
3   byte tmp;
4   byte i = 0;
5   byte j = 0;
6   bit old_gp_idle;
7   bit outermost;
8
9   do
10  :: i >= MAX_DYNTICK_LOOP_IRQ &&
11      j >= MAX_DYNTICK_LOOP_IRQ -> break;
12  :: i < MAX_DYNTICK_LOOP_IRQ ->
13     atomic {
14        outermost = (in_dyntick_irq == 0);
15        in_dyntick_irq = 1;
16     }
17     if
18     :: rcu_update_flag > 0 ->
19        tmp = rcu_update_flag;
20        rcu_update_flag = tmp + 1;
21     :: else -> skip;
22     fi;
23     if
24     :: !in_interrupt &&
25          (dynticks_progress_counter & 1) == 0 ->
26        tmp = dynticks_progress_counter;
27        dynticks_progress_counter = tmp + 1;
28        tmp = rcu_update_flag;
29        rcu_update_flag = tmp + 1;
30     :: else -> skip;
31     fi;
32     tmp = in_interrupt;
33     in_interrupt = tmp + 1;
34     atomic {
35        if
36        :: outermost ->
37           old_gp_idle = (grace_period_state == GP_IDLE);
38        :: else -> skip;
39        fi;
40     }
41     i++;
42  :: j < i ->
43     atomic {
44        if
45        :: j + 1 == i ->
46           assert(!old_gp_idle ||
```

```
47                      grace_period_state != GP_DONE);
48          :: else -> skip;
49          fi;
50      }
51      tmp = in_interrupt;
52      in_interrupt = tmp - 1;
53      if
54      :: rcu_update_flag != 0 ->
55          tmp = rcu_update_flag;
56          rcu_update_flag = tmp - 1;
57          if
58          :: rcu_update_flag == 0 ->
59              tmp = dynticks_progress_counter;
60              dynticks_progress_counter = tmp + 1;
61          :: else -> skip;
62          fi;
63      :: else -> skip;
64      fi;
65      atomic {
66          j++;
67          in_dyntick_irq = (i != j);
68      }
69  od;
70  dyntick_irq_done = 1;
71 }
```

这比前面的 dynticks_irq()过程更简单。它在第 5 行添加一个计数器变量 j。因此 i 对进入中断处理进行计数，j 对退出中断进行计数。第 7 行的 outermost 变量帮助确定 grace_period_state 变量是否需要进行安全检查。 第 10、11 行的 loop-exit 检查是否进入了特定次数的中断。对 i 的递增被移到第 41 行。第 13-16 设置 outermost 变量，以指示这是否是一个嵌套中断。并且设置 in_dyntick_irq 变量，这个变量被用于 dyntick_nohz() 过程。第 34-40 行获取 grace_period_state 变量的值，但是仅仅在最外层中断才这样做。

第 42 行模仿中断循环退出的条件：只要我们已经离开的次数少于进入的次数，那么离开另一个中断就是合法的。第 43-50 行进行安全检查，但是仅仅在我们从最外层中断退出时。最后，第 65-68 行递增中断退出计数 j，如果是退出最外层中断，则清除 in_dyntick_irq。

这个模型 (dyntickRCU-irq-ssl.spin) 产生一个正确的校验结果，状态空间超过半 M。但是，这个版本的模型没有处理 NMI，将在下一节处理这个问题。

E.7.2.7 验证 NMI 处理

处理 NM 时，我们采用与中断一样的方法，注意 NMIs 没有嵌套。最终的 dyntick_nmi() 过程如下：

```
1 proctype dyntick_nmi()
2 {
3     byte tmp;
4     byte i = 0;
5     bit old_gp_idle;
6
7     do
8     :: i >= MAX_DYNTICK_LOOP_NMI -> break;
9     :: i < MAX_DYNTICK_LOOP_NMI ->
10        in_dyntick_nmi = 1;
11        if
12        :: rcu_update_flag > 0 ->
13           tmp = rcu_update_flag;
14           rcu_update_flag = tmp + 1;
15        :: else -> skip;
16        fi;
17        if
18        :: !in_interrupt &&
19             (dynticks_progress_counter & 1) == 0 ->
20           tmp = dynticks_progress_counter;
21           dynticks_progress_counter = tmp + 1;
22           tmp = rcu_update_flag;
23           rcu_update_flag = tmp + 1;
24        :: else -> skip;
25        fi;
26        tmp = in_interrupt;
27        in_interrupt = tmp + 1;
28        old_gp_idle = (grace_period_state == GP_IDLE);
29        assert(!old_gp_idle || grace_period_state != GP_DONE);
30        tmp = in_interrupt;
31        in_interrupt = tmp - 1;
32        if
33        :: rcu_update_flag != 0 ->
34           tmp = rcu_update_flag;
35           rcu_update_flag = tmp - 1;
36           if
37           :: rcu_update_flag == 0 ->
38              tmp = dynticks_progress_counter;
39              dynticks_progress_counter = tmp + 1;
```

```
40            :: else -> skip;
41            fi;
42        :: else -> skip;
43        fi;
44        atomic {
45            i++;
46            in_dyntick_nmi = 0;
47        }
48    od;
49    dyntick_nmi_done = 1;
50 }
```

当然，实际上 NMIs 需要调整一些东西。例如，EXECUTE_MAINLINE() 宏必须注意到 NMI 处理函数 (in_dyntick_nmi)，这与中断处理函数一样 (in_dyntick_irq) ，是通过检查 dyntick_nmi_done 变量来实现的，如下：

```
1 #define EXECUTE_MAINLINE(label, stmt) \
2 label: skip; \
3       atomic { \
4           if \
5           :: in_dyntick_irq || \
6                in_dyntick_nmi -> goto label; \
7           :: else -> stmt; \
8           fi; \
9       } \
```

我们也要引入一个 EXECUTE_IRQ() 宏，它检查 in_dyntick_nmi 以允许 dyntick_irq() 执行 dyntick_nmi()：

```
1 #define EXECUTE_IRQ(label, stmt) \
2 label: skip; \
3       atomic { \
4           if \
5           :: in_dyntick_nmi -> goto label; \
6           :: else -> stmt; \
7           fi; \
8       } \
```

还有必要将 dyntick_irq() 转换为 EXECUTE_IRQ() ，如下：

```
1 proctype dyntick_irq()
2 {
3    byte tmp;
4    byte i = 0;
5    byte j = 0;
6    bit old_gp_idle;
7    bit outermost;
8
9    do
```

```
10      :: i >= MAX_DYNTICK_LOOP_IRQ &&
11           j >= MAX_DYNTICK_LOOP_IRQ -> break;
12      :: i < MAX_DYNTICK_LOOP_IRQ ->
13           atomic {
14              outermost = (in_dyntick_irq == 0);
15              in_dyntick_irq = 1;
16           }
17 stmt1: skip;
18           atomic {
19              if
20              :: in_dyntick_nmi -> goto stmt1;
21              :: !in_dyntick_nmi && rcu_update_flag ->
22                 goto stmt1_then;
23              :: else -> goto stmt1_else;
24              fi;
25           }
26 stmt1_then: skip;
27           EXECUTE_IRQ(stmt1_1, tmp = rcu_update_flag)
28           EXECUTE_IRQ(stmt1_2, rcu_update_flag = tmp + 1)
29 stmt1_else: skip;
30 stmt2: skip;    atomic {
31              if
32              :: in_dyntick_nmi -> goto stmt2;
33              :: !in_dyntick_nmi &&
34                   !in_interrupt &&
35                   (dynticks_progress_counter & 1) == 0 ->
36                      goto stmt2_then;
37              :: else -> goto stmt2_else;
38              fi;
39           }
40 stmt2_then: skip;
41           EXECUTE_IRQ(stmt2_1, tmp = dynticks_progress_counter)
42           EXECUTE_IRQ(stmt2_2,
43              dynticks_progress_counter = tmp + 1)
44           EXECUTE_IRQ(stmt2_3, tmp = rcu_update_flag)
45           EXECUTE_IRQ(stmt2_4, rcu_update_flag = tmp + 1)
46 stmt2_else: skip;
47           EXECUTE_IRQ(stmt3, tmp = in_interrupt)
48           EXECUTE_IRQ(stmt4, in_interrupt = tmp + 1)
49 stmt5: skip;
50           atomic {
51              if
52              :: in_dyntick_nmi -> goto stmt4;
53              :: !in_dyntick_nmi && outermost ->
```

```
54                old_gp_idle = (grace_period_state == GP_IDLE);
55              :: else -> skip;
56              fi;
57          }
58          i++;
59      :: j < i ->
60  stmt6: skip;
61          atomic {
62              if
63              :: in_dyntick_nmi -> goto stmt6;
64              :: !in_dyntick_nmi && j + 1 == i ->
65                  assert(!old_gp_idle ||
66                              grace_period_state != GP_DONE);
67              :: else -> skip;
68              fi;
69          }
70          EXECUTE_IRQ(stmt7, tmp = in_interrupt);
71          EXECUTE_IRQ(stmt8, in_interrupt = tmp - 1);
72
73  stmt9: skip;
74          atomic {
75              if
76              :: in_dyntick_nmi -> goto stmt9;
77              :: !in_dyntick_nmi && rcu_update_flag != 0 ->
78                  goto stmt9_then;
79              :: else -> goto stmt9_else;
80              fi;
81          }
82  stmt9_then: skip;
83          EXECUTE_IRQ(stmt9_1, tmp = rcu_update_flag)
84          EXECUTE_IRQ(stmt9_2, rcu_update_flag = tmp - 1)
85  stmt9_3: skip;
86          atomic {
87              if
88              :: in_dyntick_nmi -> goto stmt9_3;
89              :: !in_dyntick_nmi && rcu_update_flag == 0 ->
90                  goto stmt9_3_then;
91              :: else -> goto stmt9_3_else;
92              fi;
93          }
94  stmt9_3_then: skip;
95          EXECUTE_IRQ(stmt9_3_1,
96              tmp = dynticks_progress_counter)
97          EXECUTE_IRQ(stmt9_3_2,
```

```
 98            dynticks_progress_counter = tmp + 1)
 99 stmt9_3_else:
100 stmt9_else: skip;
101         atomic {
102             j++;
103             in_dyntick_irq = (i != j);
104         }
105     od;
106     dyntick_irq_done = 1;
107 }
```

注意，我们放开了``if" 语句 (f 如第 17-29 行)。另外，处理局部状态的语句不必在 dyntick_nmi()中执行。

最后，grace_period() 需要进行一些修改：

```
 1 proctype grace_period()
 2 {
 3     byte curr;
 4     byte snap;
 5     bit shouldexit;
 6
 7     grace_period_state = GP_IDLE;
 8     atomic {
 9         printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NOHZ);
10         printf("MDLI = %d\n", MAX_DYNTICK_LOOP_IRQ);
11         printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NMI);
12         shouldexit = 0;
13         snap = dynticks_progress_counter;
14         grace_period_state = GP_WAITING;
15     }
16     do
17     :: 1 ->
18         atomic {
19             assert(!shouldexit);
20             shouldexit = dyntick_nohz_done &&
21                          dyntick_irq_done &&
22                          dyntick_nmi_done;
23             curr = dynticks_progress_counter;
24             if
25             :: (curr - snap) >= 2 || (curr & 1) == 0 ->
26                 break;
27             :: else -> skip;
28             fi;
29         }
30     od;
31     grace_period_state = GP_DONE;
```

```
32    grace_period_state = GP_IDLE;
33    atomic {
34       shouldexit = 0;
35       snap = dynticks_progress_counter;
36       grace_period_state = GP_WAITING;
37    }
38    do
39    :: 1 ->
40       atomic {
41          assert(!shouldexit);
42          shouldexit = dyntick_nohz_done &&
43                       dyntick_irq_done &&
44                       dyntick_nmi_done;
45          curr = dynticks_progress_counter;
46          if
47          :: (curr != snap) || ((curr & 1) == 0) ->
48             break;
49          :: else -> skip;
50          fi;
51       }
52    od;
53    grace_period_state = GP_DONE;
54 }
```

我们在第 11 行添加了 printf()以打印 MAX_DYNTICK_LOOP_NMI 参数，在第 22 行和 44 行对 shouldexit 赋值时，添加了 dyntick_nmi_done。

这个模型 (dyntickRCU-irq-nmi-ssl.spin) 产生一个正确的校验，其状态达到数百 M。

问题 E.18：Paul 总是按照这种痛苦的风格编写代码吗?

E.7.3 回顾

```
static inline void rcu_enter_nohz(void)
{
+ mb();
__get_cpu_var(dynticks_progress_counter)++;
- mb();
}
static inline void rcu_exit_nohz(void)
{
- mb();
__get_cpu_var(dynticks_progress_counter)++;
+ mb();
```

```
}
```

图 E.16: Memory-Barrier Fix Patch

```
- if ((curr - snap) > 2 || (snap & 0x1) == 0)
+ if ((curr - snap) > 2 || (curr & 0x1) == 0)
```

图 E.17: Variable-Name-Typo Fix Patch

前面的努力有一些东西需要回顾：

Promela 和 spin 可以校验中断 /NMI 中断之间相互交互的情况。

代码中的文档可以帮助我们定位 BUGs。 在这种情况下，文档帮助定位了 rcu_enter_nohz() 和 rcu_exit_nohz()中的内存屏障位置错误，如图 E.16 所示的补丁。

及早的验证代码，通常需要在造成破坏性的后果前。这帮助定位了 rcu_try_flip_waitack_needed()中一个很微妙的错误，这个错误是十分难于测试和调试的，补丁如图 E.17。

总是需要验证已经确认过的代码。通常的方法是在插入一个故障的 BUG，并且使用校验代码捕获它。当然，如果代码获取这个 BUG，就需要检查代码本身的 BUG。但是，如果你处于这种情况，一个非常有效的调试技术就是晚上睡个好觉。

使用原子指令可以简化校验。不幸的是，使用 cmpxchg 原子指令将使中断变慢，因此这种情况下不太适用。

需要复杂的校验，通常意味着需要重新设计。实际上本节的校验设计有一个很简单的方法，将在下一节中介绍。

E.8 简单的避免形式校验

可抢占 RCU 的 dynticks 接口的复杂性，主要源自于 irqs 和 NMIs 使用了相同的代码路径和相同的状态变量。这使得我们有一个想法：将中断和 NMIs 的代码路径和变量分开，这已经由分组 RCU [McK08a] 完成了，这是由 Manfred Spraul 间接促成的[Spr08b]。

E.8.1 简单 Dynticks 接口的状态变量

```
1 struct rcu_dynticks {
2 int dynticks_nesting;
3 int dynticks;
4 int dynticks_nmi;
5 };
```

```
6
7 struct rcu_data {
8 ...
9 int dynticks_snap;
10 int dynticks_nmi_snap;
11 ...
```

图 E.18: Variables for Simple Dynticks Interface

图 E.18 展示了新的每 CPU 状态变量。这些值被集中到一个结构中，以允许多个独立的 RCU 实现(如 rcu 和 rcu_bh) 以方便有效的共享 dynticks 状态。在下文中，它们可以被认为是一个独立的每 CPU 变量。

dynticks_nesting, dynticks, 和 dynticks_snap 变量用于 irq 代码，dynticks_nmi 和 dynticks_nmi_snap 变量用于 NMI 代码，虽然 NMI 代码也引用 (但不修改) dynticks_nesting 变量。这些值按如下方法使用：

dynticks_nesting：这个变量对相应的 CPU 应当被 RCU 读临界区监控的次数进行计数。如果 CPU 处于 dynticks-idle 模式，那么它是对中断嵌套级别的计数，否则它大于中断嵌套级别。

Dynticks：如果相应的 CPU 处于 dynticks-idle 模式，并且没有中断处理函数在运行，则这个计数器的值是偶数。否则计数器的值是奇数。换句话说，如果计数器是奇数，那么相应的 CPU 可能处于 RCU 读临界区。

dynticks_nmi：如果相应的 CPU 处于 NMI 处理函数中，并且 NMI 仅仅是在 CPU 处于 dyntick-idle 模式到达时，这个计数器的值为奇数。否则，这个计数器的值是偶数。

dynticks_snap：这是 dynticks 计数器值的快照，但是仅仅是当前 RCU 优雅周期被延长得太久时才能称为快照（也就是说，如果优雅周期不长，则这个值很快就不是快照，而是慢照了）。

dynticks_nmi_snap：这是 dynticks_nmi 计数器的快照。

在一个特定时间间隔内，如果 dynticks 和 dynticks_nmi 都是偶数值，那么相应的 CPU 已经经历过一次静止状态。

问题 E.19：如果在一个中断处理函数完成前，发生了一个 NMI 中断，将发生什么，如果 NMI 持续运行，直到第二个中断开始，又会发生什么?

E.8.2 进入和退出 Dynticks-Idle 模式

```
1 void rcu_enter_nohz(void)
2 {
3 unsigned long flags;
```

```
4 struct rcu_dynticks *rdtp;
5
6 smp_mb();
7 local_irq_save(flags);
8 rdtp = &__get_cpu_var(rcu_dynticks);
9 rdtp->dynticks++;
10 rdtp->dynticks_nesting--;
11 WARN_ON_RATELIMIT(rdtp->dynticks & 0x1, &rcu_rs);
12 local_irq_restore(flags);
13 }
14
15 void rcu_exit_nohz(void)
16 {
17 unsigned long flags;
18 struct rcu_dynticks *rdtp;
19
20 local_irq_save(flags);
21 rdtp = &__get_cpu_var(rcu_dynticks);
22 rdtp->dynticks++;
23 rdtp->dynticks_nesting++;
24 WARN_ON_RATELIMIT(!(rdtp->dynticks & 0x1), &rcu_rs);
25 local_irq_restore(flags);
26 smp_mb();
27 }
```

图 E.19: Entering and Exiting Dynticks-Idle Mode

图 E.19 展示了 rcu_enter_nohz() 和 rcu_exit_nohz()，它进入、退出 dynticks-idle 模式，也可认为是``nohz" 模式。这两个函数在进程上下文调用。

第 6 行确保任何先前的内存访问(也包括在 RCU 读临界区的访问)在进入 dynticks-idle 前，可被其他 CPU 看见。 第 7 、 12 行禁止并重新打开中断。第 8 行获得当前 CPU 的 rcu_dynticks 结构指针，第 9 行递增当前 CPU 的 dynticks 计数。现在，这个计数值应当是偶数。表示我们正在从进程上下文进入 dynticks-idle 模式。最后，第 10 行递减 dynticks_nesting，现在它应当是 0。

rcu_exit_nohz() 函数非常简单，仅仅是递增 dynticks_nesting 而不是递减它，同时检查 dynticks。

E.8.3 从 Dynticks-Idle 模式进入 NMIs

```
1 void rcu_nmi_enter(void)
2 {
3 struct rcu_dynticks *rdtp;
```

```
4
5 rdtp = &__get_cpu_var(rcu_dynticks);
6 if (rdtp->dynticks & 0x1)
7 return;
8 rdtp->dynticks_nmi++;
9 WARN_ON_RATELIMIT(!(rdtp->dynticks_nmi & 0x1),
10 &rcu_rs);
11 smp_mb();
12 }
13
14 void rcu_nmi_exit(void)
15 {
16 struct rcu_dynticks *rdtp;
17
18 rdtp = &__get_cpu_var(rcu_dynticks);
19 if (rdtp->dynticks & 0x1)
20 return;
21 smp_mb();
22 rdtp->dynticks_nmi++;
23 WARN_ON_RATELIMIT(rdtp->dynticks_nmi & 0x1, &rcu_rs);
24 }
```

图 E.20: NMIs From Dynticks-Idle Mode

图 E.20 显示了 rcu_nmi_enter() 和 rcu_nmi_exit() 函数，表示从 dynticks-idle 模式进入、退出 NMI。但是，如果 NMI 在中断处理期间到达，那么 RCU 就已经注意到当前 CPU 的 RCU 读临界区了。因此 rcu_nmi_enter 的第 6 、 7 行和 rcu_nmi_exit 的第 19 、 20 行在 dynticks 为奇数的时候就返回。否则，这两个函数递增 dynticks_nmi，两个函数都在递增和可能的 RCU 读临界区之间执行内存屏障，分别如第 11、21 行所示。

E.8.4 Interrupts From Dynticks-Idle Mode

```
1 void rcu_irq_enter(void)
2 {
3 struct rcu_dynticks *rdtp;
4
5 rdtp = &__get_cpu_var(rcu_dynticks);
6 if (rdtp->dynticks_nesting++)
7 return;
8 rdtp->dynticks++;
9 WARN_ON_RATELIMIT(!(rdtp->dynticks & 0x1), &rcu_rs);
10 smp_mb();
```

```
11 }
12
13 void rcu_irq_exit(void)
14 {
15 struct rcu_dynticks *rdtp;
16
17 rdtp = &__get_cpu_var(rcu_dynticks);
18 if (--rdtp->dynticks_nesting)
19 return;
20 smp_mb();
21 rdtp->dynticks++;
22 WARN_ON_RATELIMIT(rdtp->dynticks & 0x1, &rcu_rs);
23 if (__get_cpu_var(rcu_data).nxtlist ||
24 __get_cpu_var(rcu_bh_data).nxtlist)
25 set_need_resched();
26 }
```

图 E.21: Interrupts From Dynticks-Idle Mode

图 E.21 显示了 rcu_irq_enter() 和 rcu_irq_exit()，这两个函数通知 RCU 子系统进入和退出中断。 rcu_irq_enter() 的第 6 行递增 dynticks_nesting，如果这个变量为非 0 值，第 7 行直接返回。否则，第 8 行递增 dynticks，这样它会变成奇数值，这与本 CPU 现在能够执行 RCU 读临界区是一致的。第 10 行执行一个内存屏障，以确保对 dynticks 的递增在任何后续的 RCU 读临界区可以执行前，变得可见。

rcu_irq_exit 的第 18 行递减 dynticks_nesting，如果结果非 0，第 19 行直接返回。否则，第 20 行执行一个内存屏障，以确保对 dynticks 的递增在任何之前的读临界区之后可见。第 22 行验证 dynticks 为偶数。第 23-25 行检查是否在 irq 处理之前有 RCU 回调函数在排队， 如果是这样，就通过一个 reschedule IPI 强制这个 CPU 退出 dynticks-idle 模式。

E.8.5 检查 Dynticks 静止状态

```
1 static int
2 dyntick_save_progress_counter(struct rcu_data *rdp)
3 {
4 int ret;
5 int snap;
6 int snap_nmi;
7
8 snap = rdp->dynticks->dynticks;
9 snap_nmi = rdp->dynticks->dynticks_nmi;
```

```
10 smp_mb();
11 rdp->dynticks_snap = snap;
12 rdp->dynticks_nmi_snap = snap_nmi;
13 ret = ((snap & 0x1) == 0) && ((snap_nmi & 0x1) == 0);
14 if (ret)
15 rdp->dynticks_fqs++;
16 return ret;
17 }
```

图 E.22: Saving Dyntick Progress Counters

图 E.22 显示了 dyntick_save_progress_counter()，它获取特定 CPU 的 dynticks 和 dynticks_nmi counters 的快照。第 8 、9 行将这两个变量的快照保存到局部变量中。第 10 行执行一个内存屏障，该屏障与图 E.19，E.20 和 E.21 中的内存屏障配对。第 11、12 行记录快照，以后的 rcu_implicit_dynticks_qs 会使用这两个变量，第 13 行检查 CPU 是否处于 dynticks-idle 模式，并且没有中断和 NMI 正在处理 (也就是说, 两个快照值都是偶数值)，这表示处于一个静止状态。如果是这样，第 14 、15 行记录下这个事件，第 16 行返回 true。

```
1 static int
2 rcu_implicit_dynticks_qs(struct rcu_data *rdp)
3 {
4 long curr;
5 long curr_nmi;
6 long snap;
7 long snap_nmi;
8
9 curr = rdp->dynticks->dynticks;
10 snap = rdp->dynticks_snap;
11 curr_nmi = rdp->dynticks->dynticks_nmi;
12 snap_nmi = rdp->dynticks_nmi_snap;
13 smp_mb();
14 if ((curr != snap || (curr & 0x1) == 0) &&
15 (curr_nmi != snap_nmi || (curr_nmi & 0x1) == 0)) {
16 rdp->dynticks_fqs++;
17 return 1;
18 }
19 return rcu_implicit_offline_qs(rdp);
20 }
```

图 E.23: Checking Dyntick Progress Counters

图 E.23 显示 dyntick_save_progress_counter，它被调用以确定 CPU 是否已经进入 dyntick-idle 模式。第 9 和 11 行获取一个新的 dynticks 和 dynticks_nmi

的快照，而第 10、12 行行重新获取早先保存由 dynticks_save_progress_counter() 保存的快照。第 13 行执行一个内存屏障，这个内存屏障与图 E.19, E.20 和 E.21 中的内存屏障对应。第 14、15 行检查 CPU 当前是否也处于一个静止状态 (curr 和 curr_nmi 为偶数值) 或者自从调用 dynticks_save_progress_counter()以来已经经历过一个静止状态(dynticks 和 dynticks_nmi 的值已经改变)。如果这些检查已经能够确保 CPU 经历过一个 dyntick-idle 静止状态，那么第 16 行记录下这个事实。第 19 行检查可能导致在 RCU 中等待一个离线 CPU 的竞争条件。

问题 E.20：这仍然相当复杂。为什么不用一个 cpumask_t 位集来表示每一个 CPU 是否处于 dyntick-idle 模式，当进入中断或者 NMI 时清除相应的位，在退出时设置它?

E.8.6 讨论

一个小小的改变导致 RCU 的 dynticks 接口有了实质性的简化。关键的变化是减少在中断和 NMI 上下文的共享。在这个简化的接口中，唯一的共享是在 NMI 上下文引用 irq 中的变量 (dynticks 变量)。这种类型的共享是有利的，因为 NMI 函数从不会修改这个变量，因此它的值在 NMI 处理函数的生命周期中是不变的。与 E.7 节对比一下吧，在这里，NMI 可能在 irq 函数执行的时候，修改共享的状态。

校验是一个非常棒的事情，但是简单化更好。

E.9 概要

Promela 是一个用来校验小型并行算法的非常强大的工具。它是并行内核黑客工具箱中非常有用的一款，但是不应当是唯一的一款。QRCU 的经历就是这样一种情况：通过 Promela 校验，正确性证明，以及运行 rcutorture，我现在对 QRCU 算法和实现有相当的信心。但是仅仅通过其中一种方法，就不能确定有这种信心了!

然而，如果您的办法太复杂，您就会发现太依赖于校验工具，需要认真的考虑重新进行设计。例如，E.7 节中复杂的可抢占 RCU 的 dynticks 接口实现，实际上有一个更简单的可选实现，如 E.8 节所描述。其他也是相同的，更简单的实现比验证一个复杂实现更好!

F. 问题答案

G. 术语表

H. 感谢