arXiv:2302.00748v1 [cs.DC] 1 Feb 2023

系统范围崩溃下具有常数 RMR 复杂度的可恢复互斥锁

Prasad Jayanti* Siddhartha V. Jayanti† Anup Joshi‡

2023年1月17日

摘要

我们为系统范围崩溃模型设计了两种可恢复互斥（RME）锁。第一种算法每个进程仅需 $O(1)$ 空间，并在 CC 模型中实现 $O(1)$ 最坏情况 RMR 复杂度。第二种算法在第一种算法的基础上改进，实现了（同样的）每个进程 $O(1)$ 空间，并在 $both$ CC 和 DSM 模型中实现 $O(1)$ 最坏情况 RMR 复杂度。此外，这两种算法都允许动态创建的、名称任意的线程加入协议并访问这些锁。据我们所知，在仅假设具备标准硬件支持的情况下，只有这两种 RME 锁能够实现最坏情况 $O(1)$ RMR 复杂度。鉴于 Chan 和 Woelfel 针对个体崩溃模型中的 RME 给出的 $\Omega(\log\ n/\log\log\ n)$ 最坏情况 RMR 下界，我们的结果表明，在 CC 和 DSM 模型中，系统范围崩溃模型与个体崩溃模型在最坏情况 RMR 复杂度方面存在差异。

1 引言

随着非易失性存储器（*NVM*）的出现及其广泛商用，即使发生由断电 [29, 30, 31] 引起的系统范围崩溃，它也能保留自身状态，由此开启了可恢复（又称持久化）算法的时代。这类算法使崩溃并因系统范围断电而丢失进程寄存器内容的进程，能够在重启后顺利恢复，并从崩溃时刻
[17, 2, 12, 13, 16, 14, 15, 16, 19, 20, 21, 22, 5, 6]继续计算。鉴于互斥（*mutex*）锁在并发计算 [11], 中的核心地位，可恢复互斥（RME）锁受到了广泛关注 [16, 14, 15, 16, 19, 20, 21, 22, 5, 6]。RME锁研究的主要目标，是设计在崩溃情况下，对“缓存一致性”（CC）和“分布式共享内存”（DSM）系统都具有较低“RMR”复杂度的算法。本文设计了能够容忍系统范围崩溃、适用于CC和DSM系统的RME锁。我们的锁最坏情况下的RMR复杂度仅为常数，允许动态创建的线程访问，并且每个访问线程只需常数空间。我们的算法仅使用读、写、比较并交换（CAS）和取值并存储（FAS）指令，这些指令在现代机器上都普遍受支持。

在本节余下部分，我们将回顾相关背景，阐述我们的贡献，并将其与当前最先进的成果进行比较。

1.1 故障模型

在实践中，崩溃通常由电源故障引起，导致系统中的所有进程同时崩溃 [17]。针对这类系统范围崩溃的可恢复算法，已在系统领域和理论领域得到广泛研究 [17, 15, 12, 13, 27, 9, 7, 8, 24]。许多理论文献，尤其是与可恢复互斥相关的研究，也探讨了个体崩溃模型 [14, 16, 19, 20, 21, 22, 5, 6, 4, 3] ，在该模型中，个别进程可以崩溃并重新启动，而其他进程的运行不受影响。针对个体崩溃模型设计的算法也适用于系统范围崩溃，因为一系列个体崩溃可以模拟一次系统范围崩溃。然而，直接针对系统范围崩溃设计的算法可以利用

*达特茅斯学院；prasad.jayanti@dartmouth.edu
†谷歌研究院和麻省理工学院；sjayanti@google.com，siddhartha@csail.mit.edu
‡雅虎；anup.s.joshi.gr@dartmouth.edu

同时发生故障这一结构，从而有望实现更好的效率保证。本文的主要结果——一种在系统范围崩溃情况下仍能容错的可恢复互斥算法，其最坏情况下的 RMR 数为常数——体现了这种效率提升；相比之下，Chan 和 Woelfel 已证明，在个体崩溃模型中，可恢复互斥的最坏情况 RMR 下界为 $\Omega(\log\ n/\log\log\ n)$；其中 n 表示参与该算法的进程数。

1.2 RME锁：问题陈述

标准互斥锁 ℓ 由每个进程的两个方法组成——ℓ.尝试$_p$() 和 ℓ.退出$_p$()——而一个可恢复互斥（RME）锁还额外包含一个名为 ℓ.恢复$_p$() 的方法。最初，每个进程都处于剩余区 且处于正确状态。当进程处于正确状态并位于剩余区时，可以调用并执行 p ℓ.尝试$_p$()。当 ℓ.尝试$_p$() 返回时， p 被称为处于临界区（*CS*）。进程从 CS 中 p 接下来调用并执行 ℓ.退出$_p$()。当此方法完成后， p 回到剩余区并处于正确状态。

在任意时刻，进程都可能执行一个正常步骤，或者可能发生系统级崩溃。在进程执行正常步骤时， p p 执行其程序计数器 PC_p所指向的指令。当发生系统级崩溃时，对于每个进程， p p 进入崩溃状态，PC_p 被设置为 p的剩余区，并且 p不在 NVM 中的其他所有寄存器和局部变量都被设置为任意值。当 p 随后重新启动时，必须调用 ℓ.恢复$_p$()。此方法完成后，它返回 IN REM 或 IN CS，并且 p 被认为从崩溃 状态恢复到正确 状态。ℓ.恢复$_p$() 返回 IN REM 时，会使 p 进入剩余区；返回 IN CS 时，则使 p 进入 CS。

RME锁的两个基本性质是：

- 互斥：任何时刻，至多只有一个进程处于临界区。
- 临界区重入（CSR）[16]：如果进程 p 在临界区内崩溃，那么在 p 重新进入临界区之前，其他进程都不能进入临界区。换言之，如果 p 在临界区内崩溃，那么后续执行的ℓ.恢复$_p$()一旦返回，就必须使 p 重新处于临界区内。

CSR性质是必要的，因为 p 在崩溃时可能正在临界区内操作某个共享数据结构，而崩溃可能会使该数据结构处于不一致状态。CSR性质确保 p 有机会先于其他任何进程进入临界区并完成清理。（按照可恢复互斥领域文献中的惯例，我们假设临界区是幂等的。）

RME锁还必须确保在公平 执行中取得进展——也就是说，没有进程会永远停留在临界区内，也没有进程会永久停止执行步骤，除非它处于状态正确的 remainder 中。我们的算法满足以下活性性质：

- 无饥饿性：在每次公平运行中，如果进程 p 执行 ℓ.尝试$_p$()，且此次尝试方法执行期间没有发生崩溃，那么该方法最终会完成，使进程 p 进入临界区。

另外两个重要性质是 [21]：

- 有界恢复：如果进程 p 执行 ℓ.恢复$_p$() 且期间没有崩溃，那么 p 会在自身执行常数步后从该方法返回。
- 有界退出：如果进程 p 执行 ℓ.退出$_p$() 且期间没有崩溃，那么 p 会在自身执行常数步后从该方法返回。

有界恢复保证，如果 p 在临界区内崩溃，那么重启后，它能够重新进入临界区，而不会受到其他进程的阻碍 [21]。有界退出则规定，当进程只想放弃锁时，不应受到任何阻碍。

1.3 动态加入与自适应空间

以往的可恢复互斥锁是为固定集合中的 N 个进程设计的，这些进程的标签为 1、.....、 N，并且需要预先分配长度为 $O(N)$ 或 $O(N^2)$ 的共享数组用于通信。我们设计的可恢复互斥锁支持动态加入，也就是说，任意命名、按需创建的进程（又称线程）都可以访问

我们的锁。因此，访问这些锁的进程数量没有预定义上限。为实现这一特性，我们将协议中的所有通信结构设计为基于指针的，从而无需使用预分配定长数组。

我们的空间使用量具有自适应性，也就是说，空间会随实际使用我们锁的进程数量 n 增长（不同于空间取决于锁所针对的预设最大进程数量 N）。

1.4 RMR 复杂度

互斥算法的复杂度通常在两种多处理器模型中进行研究：缓存一致性（CC）和分布式共享内存（DSM）。

在 CC 模型中，每个进程都有一个缓存。进程 p 对共享变量 X 执行读操作时，如果缓存中尚无副本，就会将 X 的副本从共享内存取到 p 的缓存中。任何进程对 X 执行的任何非读操作，都会使所有缓存中的 X 副本失效。若进程 p 对 X 执行的操作不是读操作，或者 X 的副本尚未存在于 p 的缓存中，则该操作计为一次远程内存引用（RMR）。进程崩溃时，其缓存内容会丢失。

在 DSM 模型中，共享内存在各进程之间分区。每个共享变量 X 恰好位于其中一个分区中。只有当 X 不在进程 p 所属的共享内存分区中时，进程 p 对 X 执行的任何操作才计为一次 RMR。

进程 p 的一次通行 始于它离开 remainder（处于正确状态或崩溃状态）之时，止于 p 随后返回 remainder（处于正确状态或崩溃状态）的最早时刻。

RME 算法的通行段复杂度 或*RMR* 复杂度，是进程在一次通行中产生的最坏情况下的远程内存引用次数。

1.5 我们的贡献

我们设计了两种能够容忍系统范围崩溃的 RME 算法。第一种算法在 CC系统上的最坏情况 RMR 复杂度为 $O(1)$。第二种算法在第一种算法的基础上进行了改进，在*CC* 和 DSM系统上均具有 $O(1)$ 的最坏情况 RMR 复杂度。两种算法每个进程只需 $O(1)$ 的空间，支持动态加入，并满足第 1.2 节所述的所有性质——互斥、CSR、无饥饿性、有界恢复和有界退出。两种算法仅使用标准的取值并存储（*FAS*）和比较并交换（*CAS*）指令进行同步。[1]

1.6 与先进水平的比较

此前有四项工作探讨了实现常数 RMR 复杂度的可恢复互斥锁 [14, 19, 15, 5]的可能性。其中两项工作需要硬件支持相应指令，以原子方式操作共享内存中两个互不相关的字：取并存并存（FASAS）指令和双字比较并交换（DCAS）指令，从而在单进程崩溃模型[14, 19]下实现 $O(1)$ 的最坏情况 RMR 复杂度。据我们所知，目前没有任何系统支持这些指令。

Chan 和 Woelfel 针对独立崩溃模型设计了一种可恢复互斥算法，其 RMR 复杂度为摊销$O(1)$，且需要无限空间 [5]。同两位作者给出的下界表明，在单进程崩溃模型 [6], 下无法实现$O(1)$ 最坏情况 RMR 复杂度；同时，这也证实了 Golab 和 Hendler [14] 以及 Jayanti 等人 [20] 提出的最坏情况 $\Theta(\log n/\log\log n)$ RMR 算法，是该计算模型下针对 CC 和 DSM 的最优算法，其中 n 表示访问该锁的进程数。

Golab 和 Hendler（GH）为系统范围崩溃模型设计了一对可恢复互斥算法，该模型假设“进程会从环境中获得有关故障发生的额外信息”，以实现 $O(1)$ 最坏情况 RMR [15]。具体而言，GH 算法假设进程可以

[1]操作 $r \leftarrow \mathbf{FAS}(X, v)$ 的语义如下：如果指令执行前 X 的值为 u，则该操作会将 u 写入 CPU 寄存器 r，并将 X 的值更新为 v。操作 $r \leftarrow \mathbf{CAS}(X, u, v)$ 的语义如下：如果 X 的值为 u，则该操作会将 X 的值更改为 v，并将 $true$ 写入 CPU 寄存器 r；否则，X 保持不变，且该操作会将 $false$ 写入 r。

访问一个共享故障计数器，该计数器必须由带外环境机制在每次崩溃事件后递增。与我们的算法一样，他们的 CC 算法可以实现为允许动态加入，且每个进程仅使用常数空间。然而，他们的 DSM 算法不允许动态加入，需要预先知道可能访问该锁的最大进程数 N，并且每个进程使用 $O(N)$ 空间，即总共使用 $O(N^2)$ 空间。我们的 CC 和 DSM 算法都允许动态加入，且每个进程仅使用 $O(1)$ 空间，即总共使用 $O(n)$ 空间，其中 n 是运行中实际访问该锁的进程数（因此，n 可能远小于 N）。

GH 算法不满足有界恢复和有界退出性质，但满足更强版本的无饥饿性：该性质指出，在包含无限多个超级通行阶段的无限运行中，每个执行尝试区段的进程最终都会进入临界区。

据我们所知，在不依赖标准硬件支持以外任何条件的情况下，我们的算法是唯一能够实现最坏情况 $O(1)$ RMR 复杂度的可恢复互斥锁。此外，它们也是首批同时实现有界恢复和有界退出性质以及 $O(1)$ RMR 复杂度的算法，也是首批在 DSM模型中实现每个进程 $O(1)$ 空间的算法。此外，鉴于 Chan 和 Woelfel 针对个体崩溃模型中的可恢复互斥给出的 $\Omega(\log n/\log\log n)$ 最坏情况 RMR 下界，我们的结果表明，在 CC 和 DSM模型中，系统范围崩溃模型与个体崩溃模型在最坏情况 RMR 复杂度方面存在差异（不假设存在故障检测器）。

1.7 RM 相关研究 E

Golab 和 Ramaraju 将 RME 问题形式化，并通过改造传统互斥算法设计了若干算法 [16]。Ramaraju [28], Jayanti 和 Joshi [21], 以及 Jayanti、Jayanti 和 Joshi [19]设计了支持先来先服务属性的 RME 算法 [25]。Golab 和 Hendler [14]以及 Jayanti、Jayanti 和 Joshi [20] 提出了 RMR 复杂度为次对数的 RME 算法。Dhoked 和 Mittal 提出了一种 RMR 复杂度为次对数且具有自适应性的锁 [10]。Chan 和 Woelfel [6]近期给出的下界与 Jayanti、Jayanti 和 Joshi [20]以及 Golab 和 Hendler [14] 的上界相匹配，从而确定了个体崩溃模型下 RME 的最坏情况 RMR 复杂度：对于 CC 和 DSM 模型，均为 $\Theta(\log n/\log\log n)$。有趣的是，Chan 和 Woelfel 先前的工作表明，在空间无限的情况下，可以在摊还 意义上实现 $O(1)$ 的 RMR 复杂度。也有人设计了可中止的 RME 锁：Jayanti 和 Joshi 首先提出了这类锁，使用比较并交换，且 RMR 复杂度为对数级 [22], ；随后 Katzan 和 Morrison 使用比较并交换和取值并加，实现了 RMR 复杂度为次对数的锁 [23]。所有这些结果都适用于个体崩溃故障模型。如上所述，Golab 和 Hendler [15] 提出了一种算法，在假设存在故障检测器的情况下，可在系统范围崩溃模型中实现 $O(1)$ 的 RMR 复杂度。

1.8 组织结构

在第 2 节中，我们提出了一种 RME锁算法，该算法仅针对 CC模型具有 $O(1)$ 的 RMR 复杂度。在第 4 节中，我们对该算法进行了调整，使其在 CC 和 DSM 上都达到 $O(1)$ 的 RMR 复杂度。最后，我们在第 5 节中进行总结。

2 面向 CC 的 RME锁

本节中，我们提出了一种用于在系统范围崩溃模型下实现 RME锁的算法。该算法在 CC 机器上具有 $O(1)$ 的 RMR 复杂度，如图1所示。我们将在第 2.1 节中解释其设计背后的思想，并在第 2.2 节中逐行进行说明。随后，我们将在第 3 节中给出一个基于不变式的正确性证明。

2.1 高层次思想

该算法使用以下持久性（NVM）变量。其中一些是共享变量，其余带有下标 p 的变量则是进程 p 的局部变量。

- SEQ 和 S_p: SEQ 存储一个单调递增的序列号，而局部变量 S_p 保存进程 p 认为的当前序列号。如果 p 崩溃后重新启动，它会将 SEQ 增加到 $S_p + 1$。

- LOCK[0], LOCK[1], LOCK[2]: 这是标准（不可恢复）互斥锁的三个实例。由于我们的 RME 锁的属性取决于这些底层"基础锁"的属性，因此我们将其实例化为 Lee 的队列锁 [26], ，相关内容及其属性已在附录 A 的图 6 中介绍。为与其他进程竞争以获得临界区的所有权， p 使用编号为S_p%3 的锁。

- STOP[0], STOP[1], STOP[2]: 这些是布尔变量，变量 STOP[S_p%3] 表示LOCK[S_p%3] 是否处于良好状态。也就是说，如果进程正在使用锁 LOCK[S_p%3] 时发生崩溃，且该锁之后未被重置，那么该锁就不处于良好状态，因此不应使用它来获取临界区的访问权。

- CSOWNER：该变量存储正在已实现的可恢复锁$\mathcal{L}$ 临界区内的进程名称（即实际的临界区，而非三个基础锁中任何一个的临界区）。如果没有进程处于 $\mathcal{L}$的临界区内，其值为 $\perp$ 。

- 活动状态$_p$：这个布尔型局部变量在 p 执行算法期间保持$true$ 。因此，如果 p 在执行算法时崩溃，并因此跳转到剩余区，Active$_p$ 的值为 $true$。另一方面，如果 p 正常完成算法（例如，没有发生崩溃），那么当 p到达剩余区时，Active$_p$ 的值为 $false$ 。

为了竞争获取可恢复锁 $\mathcal{L}$，每个进程 p 都会将 SEQ 中的序列读入 S_p ，并尝试获取基础锁 LOCK[S_p%3] （通过执行 LOCK[S_p%3].尝试$_p$()）。例如，如果序列号为 10 时，进程 p 和 q 竞争获取 $\mathcal{L}$ ，它们会尝试获取 LOCK[10%3] （或LOCK[1]）。如果没有发生崩溃，那么每个进程都会获取 LOCK[1], 进入 $\mathcal{L}$的临界区，然后释放LOCK[1]。因此， p 和 q 依次进入 $\mathcal{L}$ 的临界区，然后返回剩余区。

对于更复杂的情况，假设系统在 p 和 q 正在争用 LOCK[1]时发生崩溃。崩溃会抹除这些进程的 CPU 寄存器, 从而使 LOCK[1] 今后无法使用。当其中一个进程（例如 p）随后重新启动并进入 $\mathcal{L}$的 recover$_p$() 时，它会根据 Active$_p$ 中的 $true$ 值推断出自己一定是在执行该算法时发生了崩溃。因此， p 通过写入 11 使 SEQ 前进，然后继续获取 LOCK[11%3], ；如有必要，则获取 LOCK[2]。然而，假设第三个进程 r在 p 将 SEQ 改为 11 之前执行了该算法。由于 r 没有先前的上下文，当它在SEQ中读取到 10 时，只会在 S_r 中写入 10，并争用 LOCK[10%3] （即 LOCK[1]）；后者可能已经卡住，但 r 并不知道这一点。如果 p 现在采取步骤，它会在 SEQ 中写入 11, 并继续争用 LOCK[11%3]（即 LOCK[2]）。因此，此时有进程在两把不同的锁处等待，即 LOCK[1]和 LOCK[2]。为防止 r 永远卡在 LOCK[10%3], 处，我们要求它在 LOCK[10%3]的 try$_r$() 过程中等待时，同时检查 STOP[10%3] 是否变为 $true$。一旦注意到 STOP[10%3]发生变化， r 就可以迅速切换到 LOCK[11%3]。

是否存在进程可能正在等待三把锁的情形？答案是否定的。为说明这一点，假设当 SEQ 的值为 v时发生系统级崩溃。过去未曾在 SEQ 中看到过值 v 的进程（例如前述情形中的进程 r ），以及在SEQ 中看到过 v 但正常退出算法的进程，都会继续前往 LOCK[v%3], ；而那些曾在 SEQ 中看到过 v 且遭遇崩溃的进程，则会在 SEQ 中写入 $v + 1$ ，并等待在 LOCK[$(v + 1)$%3]。因此，不可能有任何进程聚集到第三把锁上。特别是，在算法使用的三把基础锁中，我们可以确定没有进程会等待在 LOCK[$(v - 1)$%3]。由于这把锁可能因先前的崩溃而陷入卡死状态，现在正是重置它并使其为将来使用做好准备的好时机；因此，随着 SEQ 从 v 变为 $v + 1$， p 会重置 LOCK[$(v - 1)$%3] 。三把基础锁已足够，并且在 SEQ 从 v 变为 $v + 1$ 时应重置 LOCK[$(v - 1)$%3] ，这一观察是我们算法中的关键见解。

共享变量（存储在非易失性存储器中）$\text{SEQ} \in \mathbb{N}$，初始值为 1。
$\text{LOCK}[0 \cdots 2]$ 是一个基础互斥锁数组，按.....实现

如图 6 所示。

$\text{STOP}[0 \cdots 2]$ **是一个布尔值数组，每个元素初始为** $false$。
CSOWNER **存储一个进程标识符或** $\perp$**，初始为** $\perp$。**进程** p **的局部持久变量（存储在非易失性存储器中）Active**$_p$ **是一个布尔值，初始为** $false$。
$\text{S}_p \in \mathbb{N}$**，初始为 1。 1.** `Remainder Section`**procedure try**$_p$**() 2. Active**$_p \leftarrow true$**3** $\text{S}_p \leftarrow \text{SEQ}$ **。. LOCK[**S_p**%3].try**$_p$**() ‖ 等待直到** $\text{STOP}[\text{S}_p\text{\%3}] = true$ **4. 如果** $\text{SEQ} \neq \text{S}_p$**：跳转到第9行 6. 等待直到** $\text{CSOWNER} = \perp$ **‖ 等待直到** $\text{STOP}[\text{S}_p\text{\%3}] = true$**跳转到第9行 7. 如果比较并交换(**CSOWNER**,** $\perp$**,** p**)：返回处于临界区 8. 如果** $\text{SEQ} \neq \text{S}_p$**：9** $\text{S}_p \leftarrow \text{S}_p + 1$ **。. LOCK[**S_p**%3].try**$_p$**() 11. 等待直到 10** $\text{CSOWNER} = \perp$**12. 如果比较并交换(**CSOWNER**,** $\perp$**,** p**)：返回处于临界区 13. 等待直到** $\text{CSOWNER} = \perp$**14.** $\text{CSOWNER} \leftarrow p$**15. 返回处于临界区 16.** `Critical Section`**procedure exit**$_p$**() 17. 如果** $\text{S}_p = \text{SEQ}$**：18. LOCK[**S_p**%3].exit**$_p$**() 19.** $\text{CSOWNER} \leftarrow \perp$**20. Active**$_p \leftarrow false$ `procedure recover`$_p$**() 21. 如果 Active**$_p \wedge \text{SEQ} = \text{S}_p$**：22. LOCK[(**$\text{S}_p - 1$**)%3].reset**$_p$**() 23. STOP[(**$\text{S}_p - 1$**)%3]** $\leftarrow false$**24. SEQ** $\leftarrow \text{S}_p + 1$ **25. STOP[**S_p**%3]** $\leftarrow true$**26. 如果** $\text{CSOWNER} = p$**：返回处于临界区 27. Active**$_p \leftarrow false$**28. 返回处于剩余区**

图1： 适用于 CC 机器上任意数量、任意名称进程的 RME 锁算法 $\mathcal{L}$ 。此处展示进程 p 的代码。

2.2 逐行说明

本节将非正式地介绍图1所示算法的工作方式。我们首先说明进程 p 在没有发生崩溃时如何执行该算法，然后解释发生崩溃时该算法如何工作。

无崩溃时的尝试。

进程 p 通过调用 try$_p$() 过程，开始尝试从剩余区进入临界区。在**第2行**， p 将 Active$_p$ 设为 $true$ ，以此记录自己已开始尝试。随后， p 在**第3行**将当前序列号 SEQ 读入其局部变量 S_p ，并在**第 4 行**尝试获取对应的锁 $\text{LOCK}[\text{S}_p\text{\%3}]$ 。执行 $\text{LOCK}[\text{S}_p\text{\%3}]$.try$_p$() 时， p 还会同时监视对应停止标志的值 $\text{STOP}[\text{S}_p\text{\%3}]$, , 方法是在 try 方法和**等待直到** 之间交替执行各步。若 S_p%3 确实是当前锁，那么 p最终会成功尝试获取该锁。

否则，如果另一个进程 q 曾在序列号为 S_p的锁中处于活动状态时崩溃，之后又重新启动，那么它会递增 SEQ，并将标志位 STOP[S_p%3]设为 true。当 p 完成第 4 行时，可能是因为它成功获得了锁，也可能是因为停止标志被置位。因此，它会在第 5 行检查序列号 SEQ是否已递增；如果已递增，就放弃 LOCK[S_p%3]，并继续在第9行更新其序列号，然后尝试获取新安装的锁（稍后将对此进行解释）。否则，如果 S_p 仍是当前序列号（即仍等于 SEQ），那么 p 会在第6行通过对 CSOWNER进行自旋等待，直至临界区空闲。再次强调，p 知道序列号可能会更新，因此也会同时监视 STOP[S_p%3]；如果停止标志被置位，就像之前一样继续执行第9行。如果p 最终发现临界区已空出（即在第6行 CSOWNER $=\perp$），它会在第7行通过使用比较并交换（CAS）将自己的名称写入 CSOWNER 字段，尝试取得临界区的所有权。如果 p 的比较并交换操作成功，就进入临界区。否则，如下文所述，p 在第6行读到临界区为空后，仍可能有两种原因导致比较并交换失败。第一种情况是，p 使用的锁一定是较旧的锁；在 p 将自身入队之前，该锁发生了崩溃，随后安装了一个新锁，使另一个进程得以进入临界区（而这些情况 p都未察觉）。第二种情况是，p 自身已在最新的锁中入队，但它输给了另一个进程 q；该进程已在较旧的锁中入队（与上述第一种情况中的 p 一样）。为确定上述哪种情况导致了这次比较并交换失败，p 会在第8行将 SEQ 与S_p 进行比较。如果二者相同，p 便知道自己处于正确的锁中，因此前往第13行等待轮到自己占用临界区。否则，p 会转而尝试最新的活动状态基础锁，因此前往第9行。

从上述描述中可以看出，p 有三种不同的方式可以到达**第 9 行**：从第 5 行、第 6 行或第 8 行。在所有情况下，p都知道，在执行第 3 行之后，序列号增加了。由于p 知道，在两次崩溃之间，序列号最多只能增加一次（这是我们的算法维持的不变量），因此它将 S_p 增加 1（第**9**行），使其等于 SEQ的当前值。这一次，当 p 在第 10 行尝试获取锁 LOCK[S_p%3] 时，可以保证它最终会成功（如果没有系统级崩溃），因此无需同时监控停止标志。在第 **11** p 行，它等待以确保临界区为空。它需要确保这一点，原因有二：要么某个进程在崩溃前获取了临界区，并且仍在其中执行；要么某个进程在当前锁替换旧锁之前，通过旧锁到达第**7** 行，并成功执行了**比较并交换**。p 越过**11**行后，如果临界区仍为空，就会尝试通过**比较并交换** 将自己的名称写入 CSOWNER。如果 p 成功执行了**比较并交换**，就会进入临界区。否则，p失败的唯一原因就是，如上所述，另一个进程必定已在第**7** 行成功执行了**比较并交换**。p会在第**13** 行再次等待，以确保临界区为空。p 到达**14** 行后，就能确定自己是通过加入最新的活动锁到达该行的，并且不会再有其他竞争者。因此，它会将自己的名称写入 CSOWNER，然后进入临界区（第**15**行）。

在 exit$_p$() 过程中，p 首先检查它排队并从中进入临界区的基础锁是否仍是活动锁（第 **17** 行）。如果它确实仍是活动锁，p 就会在第 **18** 行执行该锁的 exit$_p$() 过程。否则，p 便知道自己已不在所用锁的队列中，因此不会执行该锁的 exit$_p$() 方法。在第 **19** p 行，**19** p 通过向 CSOWNER 写入 $\perp$，告知其他进程自己正在放弃临界区。最后，在第 **20** p 行，**20** p 将 Active$_p$ 设为 $false$，以完成此次尝试。

崩溃恢复。

现在我们讨论当 p 在执行一次尝试时崩溃会发生什么。我们的算法背后的思路是保留一个活动的基础锁，而 SEQ中的数值决定三个实例中哪一个处于活动状态。因此，在任何配置中，LOCK[SEQ%3] 都是活动的基础锁。这一思路指导进程判断崩溃后是否需要设置一把新锁。因此，在第 **21** 行，p 会判断自己在崩溃前是否执行过任何有意义的步骤；如果执行过，则判断自己是否可能尝试加入当前活动的基础锁队列。如果 Active$_p = false$，p 就确定自己在崩溃前没有执行任何有意义的步骤，因此会继续执行该过程的第 **26** 行（从那里它确定能找到 CSOWNER $\neq p$，继而返回 remainder，完成此次尝试）。如果 Active$_p = true$ 但 $S_p \neq$ SEQ，那么 p 就知道自己不再需要设置新锁，因为要么新锁已经设置好，要么当前活动锁并非 p 加入的那把锁。因此，p 会继续执行第 **26** 行。如果 Active$_p = true$且 $S_p =$ SEQ，那么 p 可能在第 **4** 行或第 **10** 行将自己加入了 LOCK[SEQ%3]。因此，假设是 p 自己破坏了 LOCK[SEQ%3], 的队列结构，它就会在第**22-25** 行设置一把新锁。我们的算法维持这样一个不变量：LOCK[(SEQ + 1)%3] 始终已初始化，也就是说，LOCK 的下一个实例始终准备就绪、可供使用。因此，在按顺序安装下一把锁之前，p 会先准备好

顺序上再下一把锁，以维持该不变量。为此，在第 22 行， p 首先重置锁 $\text{LOCK}[(\text{S}_p - 1)\%3]$, ，它是顺序上再下一把锁之后即将使用的锁（也就是说，编号为 $(\text{SEQ} + 2)\%3$ 或 $(\text{SEQ} - 1)\%3$ 的锁，是顺序上再下一把锁之后即将使用的锁）。接着， p 在第 23 行重置与锁 $\text{LOCK}[(\text{S}_p - 1)\%3]$ 关联的 STOP标志位。随后， p 发出信号，表示不应再使用 $\text{LOCK}[\text{S}_p\%3]$ ，因此它在第 24 行将 SEQ 加 1。然后，它在第 25 行将 $\text{STOP}[\text{S}_p\%3]$ 设为 $true$ ，这样一来，在上次崩溃之后到现在这段时间内加入锁 $\text{LOCK}[\text{S}_p\%3]$ 队列的任何进程，都知道自己应转到顺序中的下一把锁，即 $\text{LOCK}[\text{SEQ}\%3]$。我们的算法维持的一个不变量是，在两次连续崩溃之间， SEQ 的值最多递增一次。以下示例说明这是如何实现的。假设 p、 q和 r 通过在第 3 行读取 SEQ ，并将其记录到各自的变量 S_p、 S_q和 S_r中，从而将值 10 作为目标值。假设之后发生崩溃，且 p 和 q 立即重启并积极执行步骤。 p 和 q 都读取 $\text{SEQ} = \text{S}_p$ 和 $\text{SEQ} = \text{S}_q$，因而先后执行第 22–23 行。之后， q 首先执行第 24 行，首次将 SEQ 设为 11。此后，在下一次崩溃之前， p 至多会执行一次第 24 行；这一步不会改变 SEQ 的值，因为 p 会尝试将其设为 11，而 SEQ 已经持有该值。此外，在 q执行这一步之后，无论 r 何时开始执行步骤，它总会找到 $\text{SEQ} = 11$ ，而这与 S_r 在崩溃前持有的值不同。因此， r 永远不会改变 SEQ的值。于是，执行第 25 行后， p 便完成修复并继续执行第 26 行。

在第 **26** p 行，检查是否在崩溃前获得了进入临界区的权限，方法是检查 $\text{CS}_{\text{OWNER}} = p$。如果是， p 就通过从 recover$_p$() 返回 IN CS，进入临界区。否则， p 确定已经修复了崩溃可能导致的所有问题，并且在当前尝试中不能再进入临界区。因此，它将 Active$_p$ 标志位设为 $false$ （第 **27** 行），以表示尝试结束，并返回 IN REM（第 **28** 行），回到 remainder。

如上所述，第 **26** 行可能会使 p 进入临界区，从而满足 CSR 属性。因此，在 exit$_p$() 过程 p 会检查自己是正常进入临界区，还是如上所述因 CSR 属性而进入临界区。如果 p发现第 **17** 行的 $\text{SEQ} = \text{S}_p$ ，就会推断自己是正常进入临界区的，因此会在第**18**行执行 $\text{LOCK}[\text{S}_p\%3]$ 的 exit$_p$() 过程。否则， p 知道自己已不在所用锁的队列中，因此不会执行 exit$_p$()。在第 **19** p 行，通知其他进程自己正在放弃临界区，方法是将 $\perp$ 写入 CS_{OWNER}。最后，在第 **20** p 行将 Active$_p$ 设为 $false$ ，以完成此次尝试。

3 算法正确性

正确性证明基于算法的归纳不变式，我们在图 2 中给出了该不变式。为便于分析，我们为每个进程 p 引入一个抽象变量 $status_p$ （仅用于证明）。$status_p$ 反映了协议的哪个区段是 p 最近一次崩溃时所在的区段（如果发生过崩溃）。具体而言，如果 p 在尝试区段、临界区或退出区段中崩溃，则 $status_p$ 分别被设置为从尝试区恢复、从临界区恢复或从退出状态恢复。随后，当 p 执行完 recover 方法时，它会再次达到 $good$ 状态。当处于 $good$ 状态时，如果 p 调用 recover 方法，则 $status_p$ 会被设置为从余数部分恢复。

在附录 B 中，我们通过归纳法证明该不变量，并利用它证明 CC 机器的主要结果。附录 B 由若干小节组成:

- 在 B.2 节中，我们证明了若干引理，以说明三个基础锁均被正确使用，即遵循使用模式。
- 在 B.3 节中，我们使用该不变量证明算法的性质，即互斥、无饥饿性、CSR、有界恢复和有界退出。
- 在 B.4 节中，我们论证该算法的 RMR 复杂度为 $O(1)$。
- 在附录 C 中，我们给出该不变量的完整归纳证明。

为了说明我们如何利用不变量证明这些性质，下面我们重现互斥和无饥饿性的证明。当然，这些证明依赖于附录中的其他引理，文中也对这些引理作了适当引用。最后，我们以主要定理的陈述结束本节，该定理总结了关于缓存一致性算法的结果。

引理 15 (　　互斥）。在任何 ru 的任何配置中，至多有一个进程处于临界区 n.

证明。假设存在矛盾：在同一配置中，有两个进程 p 和 q 处于临界区。因此，在同一配置中，$PC_p = \mathbf{16}$ 和 $PC_q = \mathbf{16}$。根据条件 9，我们还得到在同一配置中有 $\text{CSOWNER} = p$ 和 $\text{CSOWNER} = q$。因此，CSOWNER 在同一配置中有两个不同的值，矛盾。

□

引理 17（无饥饿性）。在任何运行的任意配置中，处于临界区的进程至多有一个。在每个公平运行中，如果进程 p 执行尝试方法，且在此次尝试方法执行期间没有发生崩溃，那么该方法最终会完成，使 p 进入临界区。

证明。由于运行中只有有限个崩溃步骤，为便于论证，取一个运行，并选定该运行中的最早时刻 τ，使得截至 τ所有崩溃都已发生。我们需要证明，如果进程 p 在时刻 $t > \tau$调用 try$_p$()，那么它会在某个时刻 $t' > t$处于临界区。检查算法可知，try$_p$() 方法返回时，会使 p 处于临界区，因为每条返回语句返回的值都是 IN CS。因此，我们需要证明进程 p 不会永远卡在第 4、5–6、10、11 或 13 行，这就能确保 p 确实会从 try$_p$() 返回。根据引理 B.5、B.6 和 B.7，我们知道 p 最终会分别越过第 5–6、11 和 13 行。当 $PC_p = \mathbf{4}$时，根据条件 11，$\text{S}_p \in \{\text{SEQ} - 1, \text{SEQ}\}$。如果 $\text{S}_p = \text{SEQ} - 1$，根据条件 12，$\text{STOP}[\text{S}_p\%3] = true \vee (\exists q, PC_q = \mathbf{25} \wedge \text{S}_q = \text{S}_p)$。无论哪种情况，$\text{STOP}[\text{S}_p\%3] = true$ 最终都会发生，而 p 会在第 4 行的等待循环中注意到这一点并越过该行。因此，我们假设每当 p 执行第 4 行或第 10 行时，$\text{S}_p = \text{SEQ}$，因为根据条件 11，这是唯一的另一种可能性。接下来，我们将论证LOCK[SEQ%3] 满足无饥饿性，从而可推出 p 会越过第 4 行或第 10 行。检查算法可知，任何在第 22 行执行重置()的进程，都是为了 $\text{LOCK}[\text{SEQ} - 1]$或 $\text{LOCK}[\text{SEQ} - 2]$，因为根据条件 11 $\text{S}_p \in \{\text{SEQ} - 1, \text{SEQ}\}$，对于任何满足 $PC_p = \mathbf{22}$的进程 p。由此可知，自最近一次崩溃以来，没有进程会执行 LOCK[SEQ%3].重置()，满足基础锁无饥饿性的第一个条件。显然，基础锁无饥饿性的第二个条件也满足。引理 B.5、B.6 和 B.7 确保基础锁无饥饿性的最后一个条件同样满足。因此，我们知道 LOCK[SEQ%3] 满足无饥饿性。由此，进程 p 根据基础锁的无饥饿性会越过第 4 行或第 10 行。于是结论得证。

□

定理 3.1. 图*1*中的算法可在 *CC* 机器上解决可恢复互斥问题，适用于任意数量、名称任意的进程。它满足互斥、无饥饿性、临界区重入、有界恢复和有界退出。该算法的空间复杂度为每个进程 $O(1)$，在 *CC* 机器上的 *RMR* 复杂度为每次通行 $O(1)$ 。

4 改进适用于 DSM 的算法

刚才介绍的 CC 算法在 DSM 模型中的 RMR 复杂度无界。本节将对其进行改进，得到一种在 DSM 和 CC 模型中 RMR 复杂度均为 O(1) 的新算法。改进后的算法与 CC 算法极为相似，因此其正确性可由 CC 算法的正确性推出。下文将介绍如何将 CC 算法转换为高效 DSM 算法的主要思路（第 4.1 节），随后给出技术细节（第 4.2 节和第 4.3 节）。

4.1 DSM 算法的核心思想

要使我们的 RME 算法在 DSM 模型中达到 O(1) RMR 复杂度，我们需要在 DSM 模型中具有 O(1) RMR 复杂度的基础互斥锁——$\text{LOCK}[i]$ 用于 $i \in [0, 2]$。我们通过使用图 7 中 Lee 的第二个算法，而不是他的第一个算法，来实现这一点。除了这一简单改动之外，我们还必须解决一个问题：算法中进程进行忙等待的每一行都可能产生无界数量的 RMR。接下来，我们将说明该算法中有三个忙等待来源，并分别解释如何将每种来源产生的 RMR 降至 O(1)。

- 较早算法第 6、11 和 13 行中对 CSOWNER 的忙等待，在 DSM 机器上会产生无界的远程内存引用，因为变量 CSOWNER 必须位于这样一个内存分区中：

定义：
• $\forall i \in [0,2]$, Lock[i].State = （Lock[i].TrySet, Lock[i].CSSet, Lock[i].ExitSet）。
• $\forall i \in [0,2]$, Lock[i].Set = Lock[i].TrySet ∪ Lock[i].CSSet ∪ Lock[i].ExitSet.
条件：

1. $1 \leq S_p \leq Seq$2. Lock[(Seq + 1)%3].State = (ϕ, ϕ, ϕ) ∧ ($PC_p \in \{$**23**, **24**$\}$ ⇒ Lock[(S_p − 1)%3].State = (ϕ, ϕ, ϕ)) 3. Stop[Seq%3] = $false$ ∧ Stop[(Seq + 1)%3] = $false$ ∧ (PC_p = **24** ⇒ Stop[(S_p − 1)%3] = $false$) 4. ((活动状态$_p = false \vee S_p <$ Seq ∨ $PC_p \in \{$**3**, **19**, **20**, **26**, **27**$\}$) ⇒ $p \notin$ Lock[Seq%3].Set) ∧ ($S_p <$ Seq − 1 ⇒ $p \notin$ Lock[S_p%3].Set) 5. $\forall i \in [0,2]$, p 至多属于 Lock[i].TrySet、 Lock[i].CSSet或 Lock[i].ExitSet中的一个。 6. $PC_p \in \{$**4**, **10**$\}$ ⇒ $p \in$ Lock[S_p%3].TrySet7. ((PC_p = **5** ∧ Stop[S_p%3] = $false$) ∨ $PC_p \in \{$**6-8**, **11-15**$\}$ ∨($PC_p \in \{$**16**, **17**$\}$ ∧ S_p = Seq)) ⇒ $p \in$ Lock[S_p%3].CSSet8. PC_p = **18** ⇒ $p \in$ Lock[S_p%3].ExitSet9. ((活动状态$_p = false \vee PC_p \in \{$**2-13**, **20**, **27**$\}$) ⇒ CSowner ≠ p) ∧ (PC_p = **14** ⇒ CSowner =⊥)∧ (($PC_p \in \{$**15-19**$\}$ ∨ $status_p$ = *recover-from-cs*) ⇒ CSowner = p) 10. ($PC_p \in \{$**3-20**, **22-25**$\}$ ⇒活动状态$_p = true$)∧ ((($PC_p \in \{$**1**, **21-27**$\}$ ∧ $status_p \in \{$*good*, *recover-from-rem*$\}$) ∨ $PC_p \in \{$**2**, **28**$\}$) ⇒活动状态$_p = false$) 11. ($PC_p \in \{$**4-8**, **18**, **22-24**$\}$ ⇒ $S_p \in \{$Seq − 1, Seq$\}$) ∧ ($PC_p \in \{$**9**, **25**$\}$ ⇒ S_p = Seq − 1)∧ ($PC_p \in$ [**10**, **15**] ⇒ S_p = Seq) ∧ ((PC_p = **26** ∧ CSowner = p) ⇒ $S_p <$ Seq) 12. 该条件表示，如果 Seq 已递增，那么 Stop[(Seq − 1)%3] 已经处于 $true$ 状态，或者某个进程正准备将其设为 $true$。($PC_p \in \{$**4-6**$\}$ ∧ S_p = Seq − 1) ⇒(Stop[S_p%3] = $true$ ∨ (∃q, PC_q = **25** ∧ S_q = S_p)) 13. 该条件有助于证明：一旦 Seq 递增，在下一次崩溃之前就不会再次递增。论证基于这样一个事实：在运行过程中，一旦某个进程进入某些只能在Seq 递增后出现的状态，就不会有其他进程在下一次崩溃之前再次递增 Seq 。(($PC_p \in \{$**4-7**, **18**, **22-25**$\}$ ∧ S_p = Seq − 1) ∨(PC_p = **7** ∧ CSowner ≠⊥) ∨ $PC_p \in \{$**8-15**$\}$)⇒ ∀q, (¬($PC_q \in \{$**1**, **21**$\}$ ∧活动状态$_q = true$ ∧ S_q = Seq) ∧($PC_q \in \{$**22-25**$\}$ ⇒ S_q = Seq − 1)) 14. 该条件基本上意味着，从三个基础锁中至多可以产生两个活动队列。此外，一旦来自较早队列的某个进程 q 越过第 7 行，该队列在接下来的运行过程中直到下一次崩溃前都会冻结。(∀q, ($PC_p \in \{$**7**, **8**, **13**$\}$ ∧ S_p = Seq ∧ CSowner = q) ⇒ (S_q = Seq − 1 ∧ $q \in$ Lock[S_q%3].CSSet∧ ∀r, ($q \neq r \wedge S_r = S_q$)⇒ ($PC_r \in \{$**1-4**, **9**, **20**, **21-28**$\}$ ∨(PC_r = **5** ⇒ Stop[S_r%3] = $true$)))∧ ((($PC_p \in \{$**8**, **13**$\}$ ∧ S_p = Seq ∧ CSowner =⊥) ∨ $PC_p \in \{$**14**, **15**$\}$)⇒ (∀r, ($S_r \neq$ Seq − 1 ∨ $PC_r \in \{$**1-4**, **9**, **20**, **21-28**$\}$ ∨(PC_r = **5** ⇒ Stop[S_r%3] = $true$)) ∧ ∃q, $q \in$ Lock[(Seq − 1)%3].CSSet))

图 2： 图1所示的可恢复互斥算法的不变量。该算法在每次运行的每种配置中都满足上述所有条件的合取。

除一个进程外，对其他所有进程都是远程的。为克服这一困难，我们引入一种称为可捕获对象的新对象，它针对算法在 CSowner上执行的每种操作都提供一个方法，并实现这些方法，使其只产生 O(1) 个远程内存引用。更具体地说，在我们的 DSM 算法中，CC算法的共享变量CSowner 被一个可捕获对象 CSowner 替代；而先前算法中的步骤**“等待直到** CSowner =⊥”（第 6、11 和 13 行）、“CSowner ←⊥”（第 19 行）、“读取CSowner”（第 26 行）、“CSowner ← p”（第14行）以及“CAS(CSowner, ⊥, p)”（第 7 和 12 行）则分别由常数RMR方法 CSowner.wait$_p$()、CSowner.release$_p$()、CSowner.read$_p$()、CSowner.write$_p$() 和 CSowner.capture$_p$() 替代，如图3所示。

• 在第 4 行和第6行均出现的语句**“等待直到** Stop[S_p%3] = $true$”，是 DSM模型中远程内存引用无界增长的一个来源。尽管第 4 行的等待语句与第6行的等待语句完全相同，但二者之间存在显著差异。在第6行，最多只有一个进程会在 Stop[S_p%3] 上忙等待——即 Lock[S_p%3] 的获胜者。相比之下，许多进程可以

共享变量（存储于非易失性存储器中） $\text{SEQ} \in \mathbb{N}$，初始值为 1。$\text{LOCK}[0 \cdots 2]$ 是一个基本互斥锁数组，

$\text{STOP}[0 \cdots 2]$ 是一个布尔信号对象数组，如图5所实现，每个对象初始为$\text{STOP}[0 \cdots 2]$。

CSOWNER是一个可捕获对象，如图 如图 7 所实现。

如图5所实现，每个对象初始为 $false$。

4所实现，初始为 $\perp$。

进程 p 的局部持久变量（存储在 NV M)

活动状态$_p$ 是一个布尔值，初始值为 $false$。$\text{S}_p \in \mathbb{N}$，**初始值为 1。1.** `Remainder Section`**procedure try$_p$() 2. 活动状态$_p \leftarrow true$3** $\text{S}_p \leftarrow \text{SEQ}$ 。**.** $\text{LOCK}[\text{S}_p\%3]$.`try`$_p$**()** **4** **5** $\text{SEQ} \neq \text{S}_p$ **9** 。**如果： 则转到** **6** CSOWNER.`wait`$_p$ S_p 3 **第 行。（%）‖** $\text{STOP}[\text{S}_p\%3]$.`wait`$_p$**() 转到第9行 7. 如果** CSOWNER.`capture`$_p$**() 则返回处于临界区 8. 如果** $\text{SEQ} \neq \text{S}_p$ **则9** $\text{S}_p \leftarrow \text{S}_p + 1$ 。**..** $\text{LOCK}[$**(**$\text{S}_p - 1$**)%** 3].`exit`$_p$**() 10.** **9 1** $\text{LOCK}[\text{S}_p\%3]$.`try`$_p$**() 11.** CSOWNER.`wait`$_p$**(**$\text{S}_p$ **%3) 12. 如果** CSOWNER.`capture`$_p$**() 则返回处于临界区 13.** CSOWNER.`wait`$_p$**(**S_p**%3) 14.** CSOWNER.`write`$_p$**()** **15. 返回处于临界区 16.** `Critical Section` `procedure exit`$_p$**()17** $x_p \leftarrow \text{SEQ}$ 。**..如果** $\text{S}_p = x_p$ **则** **18** $\text{LOCK}[\text{S}_p\%3]$.`exit`$_p$**() 18.1.如果** $\text{S}_p = x_p - 1$ **则** $\text{LOCK}[\text{S}_p$ **%3].**`exit`$_p$**() 19.** CSOWNER.`release`$_p$**() 20. 活动状态** $_p \leftarrow false$`procedure recover`$_p$**() 21. 如果 活动状态** - $_p \wedge \text{SEQ} = \text{S}_p$**: 22.** $\text{LOCK}[$**(**$\text{S}_p - 1$**)%3].**`reset`$_p$**() 23.** $\text{STOP}[$**(** $\text{S}_p - 1$**)%3].**`reset`$_p$**()24** $\text{SEQ} \leftarrow \text{S}_p + 1$ 。**..** $\text{STOP}[\text{S}_p$ **%3].**`set`$_p$**() 25.1.如果** **25** $\text{S}_p = \text{SEQ} - 1$**: 则** $\text{LOCK}[\text{S}_p$**%** 3].`exit`$_p$**() 26. 如果** CSOWNER.`read`$_p$**()** $= p$ **则返回处于临界区 27. 活动状态**$_p \leftarrow false$**28. 返回处于剩余区**

-

-

图3： 适用于任意数量、名称任意的进程（针对 DSM 机器）的 RME锁算法 $\mathcal{L}$ 。展示进程 p 的代码。

在第 4 行进行忙等待。我们设计了不同的解决方案，将这两行产生的无界远程内存引用降至 $O(1)$。

为处理第 6 行，我们实现了一个新对象，称为布尔信号对象。它为算法对 $\text{STOP}[i]$ 执行的每种操作都提供一个方法，并实现这些方法，使其每次仅产生 $O(1)$ 个 RMR。更具体地说，在我们的 DSM 算法中，缓存一致性算法的共享变量 $\text{STOP}[i]$ 被布尔信号对象 Stop$[i]$ 取代；早期算法中的步骤**“等待直到** $\text{STOP}[i] = true$”（第 6 行）、“$\text{STOP}[i] \leftarrow true$”（第 25 行）和“$\text{STOP}[i] \leftarrow false$ ”（第 23 行）则分别被常数RMR方法 Stop$[i]$.wait$_p$()、Stop$[i]$.set$_p$() 和 Stop$[i]$.reset$_p$() 取代。对比图 1 和图 3 的第 6、23 和 25 行即可看出这一点。

- 在缓存一致性算法的第 4 行， p 会在并行执行对 $\text{LOCK}[S_p\%3]$ 的 Try 方法时，反复读取 $\text{STOP}[S_p\%3]$变量。之所以反复读取，是因为如果某个进程 q 安装$\text{LOCK}[(S_p+1)\%3]$，而此时 p 正在执行 $\text{LOCK}[S_p\%3]$, 的 Try 方法，算法就需要 p 检测到这一变化，并迁移到新安装的锁。如果事件按以下顺序发生，就会出现这种情况：当 q 正在执行 $\text{LOCK}[S_p\%3]$ 的 try 方法时发生崩溃；此后进程重新启动时， p 开始执行算法，从 SEQ读取 S_p，并且（由于不知道之前发生过崩溃）执行基础锁 $\text{LOCK}[S_p\%3]$ 的 try 方法（第 4 行）；随后 q重启，执行 recover 方法，并将 SEQ 递增至 S_p+1。

  在上述场景中，由于已安装较新的基础锁，即 $\text{LOCK}[(S_p+1)\%3]$, , p 以及其他任何卡在 $\text{LOCK}[S_p\%3]$ 尝试区段中的此类进程，都应释放自身并迁移到 $\text{LOCK}[(S_p+1)\%3]$的尝试区段。为了满足这一需求，同时只耗费 $O(1)$ 个远程内存引用，我们利用 Lee 锁的“释放性质”。该性质保证，只要所有进程——包括那些在使用 $\text{LOCK}[S_p\%3]$时崩溃的进程——最终都执行该锁的退出区段，就不会有进程卡在 $\text{LOCK}[S_p\%3]$ 中（附录 A 中的定理A.2陈述了释放性质，而该定理前的段落则对这一性质作了非正式说明）。这一洞见消除了第 4 行的“等待直到”语句（比较图1和图3中的第 4 行），并增加了三行（第 9.1、18.1 和 25.1 行）：进程在安装新锁或迁移到新锁时，会对旧锁执行退出方法。

以上思路催生了图3中的 DSM算法。该算法增加了三行，在这些行中调用 exit$_p$，如上所述；其他每一行都对应于前面 CC算法中编号相同的行。

在接下来的两个小节中，我们将介绍可捕获对象和布尔信号对象的规范与实现，从而完整描述我们的 DSM算法。

4.2 可捕获对象：规范与实现

A可捕获对象 $\mathcal{C}$ 的规范如下：$\mathcal{C}$.状态 是一个进程标识符或 $\perp$，而 $\mathcal{C}$ 支持以下操作。

- $\mathcal{C}$.read$_p$()：返回 $\mathcal{C}$.状态。
- $\mathcal{C}$.write$_p$()：将 $\mathcal{C}$.状态 改为 p。
- $\mathcal{C}$.release$_p$()：将 $\mathcal{C}$.状态 改为 $\perp$。
- $\mathcal{C}$.capture$_p$()：如果 $\mathcal{C}$.状态 $=\perp$，则将 $\mathcal{C}$.状态 改为 p 并返回 $true$；否则，返回 $false$，并保持 $\mathcal{C}$.状态不变。

- $\mathcal{C}$.wait$_p$($i\in\{0,1,2\}$)：此方法仅当在其执行期间的某个时刻 $\mathcal{C}$.状态 $=\perp$ 时才返回。因此，它等价于语句“等待直到 $\mathcal{C}$.状态 $=\perp$”（对于 wait 操作而言，参数 i 与操作语义无关，但出于实现上的便利，我们引入了该参数，具体解释见下文。）

在图3的算法中，有三行代码会使进程等待 CSOWNER （第 6、11、13 行）。在这些行中，对于任意 $i\in\{0,1,2\}$，任一时刻至多只有一个进程 p 执行 CSOWNER.wait$_p$(i)。因此，就我们的目的而言，只需实现一个可捕获对象，使其在每个 i 至多支持一个等待进程（以及任意数量可能执行其他操作的进程）。

图4中的简单算法能够高效地实现此类对象，具体如下列定理所述。

定理4.1. 对于图4中的算法，若对于每个 $i\in\{0,1,2\}$，没有两个进程并发执行 $\mathcal{C}$.wait(i)，则：

- $\mathcal{C}$ 是严格线性化的 [1], , 即，(1) $\mathcal{C}$ 是线性化的；(2) 如果操作完成之前发生崩溃，那么该操作要么在崩溃之前生效，要么永远不会生效。
- read$_p$()、write$_p$()、release$_p$() 和 capture$_p$() 操作均为无等待操作。

共享变量（存储于非易失性存储器中）：X 保存进程标识符或 $\perp$；初始化为 $\perp$。GO_p 保存一对值（$\text{GO}_p.seq$, $\text{GO}_p.flag$），其中 $\text{GO}_p.seq$ 是整数，$\text{GO}_p.flag$ 是布尔值； GO_p 初始化为任意值。GO_p 位于进程 p 的共享内存分区中。$W[0 \cdots 2]$ 是一个指针数组。每个指针的初始值均为 $\perp$。`procedure` $\mathcal{C}$.`read`$_p$() 1. 返回 X`procedure` $\mathcal{C}$.`write`$_p$() 2. $X \leftarrow p$`procedure` $\mathcal{C}$.`capture`$_p$() 3. 返回 CAS($X, \perp, p$)`procedure` $\mathcal{C}$.`wait`$_p$($i_p \in \{0,1,2\}$) 4. $x_p \leftarrow \text{GO}_p.seq$5. $\text{GO}_p \leftarrow (x_p + 1, false)$ 6. $W[i_p] \leftarrow$ &GO_p7. 如果 $X = \perp$，则返回 8. 等待直到 $\text{GO}_p.flag$~~`procedure` $\mathcal{C}$.`release`$_p$~~() 9. $X \leftarrow \perp$10. 对于 $i_p \in \{0,1,2\}$11. $ptr_p \leftarrow W[i_p]$12. 如果 $ptr_p \neq \perp$，则 13. $(x_p, b_p) \leftarrow *ptr_p$14. 如果 $\neg b_p \wedge (X = \perp)$，则 15. CAS($*ptr_p$, $(x_p, false)$, $(x_p, true)$)

图4： 可捕获对象的严格线性化实现 $\mathcal{C}$，该对象支持每个 $i \in \{0,1,2\}$ 有一个进程等待。进程 p 的代码。

- 假设进程 p 调用 $\mathcal{C}$.wait$_p$(i_p) 后没有发生崩溃，并且假设从某个任意时间点开始，$\mathcal{C}$.状态 $=\perp$ 始终成立。那么，在公平运行中，p 最终会从$\mathcal{C}$.wait$_p$(i_p)返回。

- 在 DSM 和缓存一致性机器中，进程调用并完成这五种操作中的任意一种，都会产生 $O(1)$ 个远程内存引用。

证明概要：该算法维护关键不变量 $\mathcal{C}$.状态 $= X$，并分别在第 1、2、3 和 9 行对读、写、捕获和释放操作进行线性化。只有当等待操作从 wait 方法返回时，我们才对该操作进行线性化；我们注意到，如果等待操作返回，那么在该等待操作执行期间必定存在某个时刻满足 $X = \perp$，我们就在任意一个这样的时刻对该操作进行线性化。因此，引理的第（1）部分得证。其余部分通过检查代码即可轻松得出。■

4.3 信号对象：规范与实现

布尔信号对象 *boolean signal object* $\mathcal{S}$ 的规范如下： $\mathcal{S}$.状态 为布尔值，且 $\mathcal{S}$ 支持以下三种操作。

- $\mathcal{S}$.集合$_p$()：将 $\mathcal{S}$.状态 更改为 $true$。
- $\mathcal{S}$.重置$_p$()：将 $\mathcal{S}$.状态 更改为 $false$
- $\mathcal{S}$.wait$_p$()：只有当方法执行期间的某个时刻 $\mathcal{S}$.状态 = 为真时，此方法才会返回。因此，它等同于语句“等待直到 $\mathcal{S}$.状态 =为真”

在图3的算法中，对于任意一个 $i \in \{0, 1, 2\}$，任意时刻最多只有一个进程 p 在 $\text{STOP}[i]$ （第6行）等待。因此，就我们的目的而言，只需实现一个仅支持一个等待进程（以及任意数量的设置和重置进程）的布尔信号对象即可。图5中的简单算法可以高效地实现这种对象，概述如下：

共享变量（存储在非易失性存储器中）： X 是布尔值；初始化为 $false$。GO_p 保存一个布尔值；初始值任意。 GO_p 位于 p的内存分区中。W 是一个指针；初始化为 $\perp$。

`procedure` $\mathcal{S}$.`wait`$_p$()**1.** $\text{GO}_p \leftarrow false$**2.** $W \leftarrow \&\text{GO}_p$**3. 如果** X**则返回 4. 等待直到** GO_p

`procedure` $\mathcal{S}$.`set`$_p$()**5.** $X \leftarrow true$**6.** $ptr_p \leftarrow W$**7. 如果** $ptr_p \neq \perp$ **则 8.** $*ptr_p \leftarrow true$

`procedure` $\mathcal{S}$.`reset`$_p$()**9.** $X \leftarrow false$

图5： 支持单个等待进程的布尔信号对象 $\mathcal{S}$ 的严格线性化实现。进程 p的代码。

定理4.2。F　　对于图5中的算法，如果没有两个进程同时执行 $\mathcal{S}$.wait()，那么：

- $\mathcal{S}$ 是严格线性化的。
- 操作set$_p$()和reset$_p$()是无等待的。
- 假设进程 p 调用 $\mathcal{S}$.wait$_p$() 后没有发生崩溃，并且 $\mathcal{S}$.状态 $=true$从某个任意时刻开始一直成立。那么，在公平运行中， p 最终会从$\mathcal{S}$.wait$_p$() 返回。
- 在 DSM 和 CC 机器中，进程调用并完成这三种操作中的任意一种，都会产生 $O(1)$ 个 RMR。

证明概要：该算法保持了关键不变量 $\mathcal{S}$.状态 $= X$，并分别在线 5 和线 9 处将集合和重置操作线性化。仅当等待操作从 wait 方法返回时，我们才将其线性化；我们注意到，若等待操作返回，则在该等待操作的执行过程中存在某个时刻，此时 $X = true$，我们便在任一这样的时刻将该操作线性化。因此，引理的第（1）部分成立。其余部分通过检查代码即可轻易得出。■

4.4 最终结果

下面的定理总结了本文的结果。

定理4.3。 图3中的算法在 DSM 和 CC 机器上为任意数量、名称任意的进程解决了可恢复互斥问题。该算法满足互斥、有界退出、有界临界区重入和无饥饿性。该算法的空间复杂度为每个进程 $O(1)$，并且该算法在 DSM 和 CC 机器上的 RMR 复杂度为每次通行 $O(1)$。

证明：由于图3中的算法与图1中的算法具有相同的不变量，因此这些性质的证明也相同。□

5 结论

针对系统范围崩溃，我们设计了首个适用于 CC 和 DSM 模型的 RME 算法，其最坏情况下的 RMR 复杂度为 $O(1)$。我们的算法允许动态创建的线程即时访问，并且每个线程仅需 $O(1)$ 空间。

对于 RME 问题，在个体崩溃模型 下，无论是 CC 还是 DSM 机器（使用实际可行的原语），其最坏情况 RMR 复杂度都已由 Jayanti 等人以及 Golab 和 Hendler [20, 14] 给出的上界，以及 Chan 和 Woelfel[6]给出的匹配下界确定为 $\Theta(\log n/\log\log n)$。本文将系统范围崩溃模型 下的最坏情况 RMR 复杂度确定为 $\Theta(1)$，从而将我们的直觉形式化：相比系统级崩溃，个体崩溃可能需要付出更大代价来应对。

最后，我们提出两个开放问题：

1. 本文中的算法使用无界序列号。在现代 64 位架构上，这并非实际问题，因为即使每秒发生一百万次系统级崩溃，计数器也需要 50 万年才会溢出。不过，从理论角度而言，研究能否去掉这类变量仍然很有意思。

2. 我们的算法不满足先到先服务（FCFS）性质，但满足以下较弱的性质：如果一次无限运行中只有有限次崩溃，那么在该运行的某个无限后缀中，先到先服务（FCFS）性质将成立。在不牺牲其他指标的前提下设计一种先到先服务算法，是一个令人向往的开放问题。

参考文献

[1] Aguilera, M. K., and Frølund, S. 严格线性化与中止的力量。技术报告 HPL-2003-241，惠普实验室，2003年。[2] Attiya, H., Ben-Baruch, O., and Hendler, D. 嵌套安全的可恢复线性化：非易失性内存的模块化构造。载于 PODC（2018），C. Newport 和 I. Keidar 编，ACM，第 7–16 页。

[3] Attiya, H., Ben-Baruch, O., and Hendler, D. 嵌套安全的可恢复线性化：非易失性内存的模块化构造。载于《2018年 ACM 分布式计算原理研讨会论文集》（美国纽约州纽约市，2018年），PODC '18，计算机协会，第 7–16 页。[4] Ben-David, N., Blelloch, G. E., Friedman, M., and Wei, Y. 故障持久内存上的无延迟并发。载于《第31届 ACM 算法与体系结构中的并行性研讨会》（美国纽约州纽约市，2019年），SPAA '19，计算机协会，第 253–264 页。[5] Chan, D. Y. C., and Woelfel, P. 基于标准原语实现的常数摊销 RMR 复杂度可恢复互斥。载于 PODC '20：ACM 分布式计算原理研讨会，线上会议，意大利，2020年8月3日至7日（2020年），Y. Emek 和 C. Cachin 编，ACM，第 181–190 页。[6] Chan, D. Y. C., and Woelfel, P. 可恢复互斥 RMR 复杂度的紧下界。第 533–543 页。[7] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., andSwanson, S. Nv-heaps：利用新一代非易失性内存，让持久对象快速且安全。计算机协会。[8] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., andSwanson, S. Nv-heaps：利用新一代非易失性内存，让持久对象快速且安全。《SIGARCH 计算机体系结构新闻》39卷1期（2011年3月），第 105–118 页。

[9] Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B. C., Burger, D., and Coetzee, D. 通过字节寻址持久内存改善 I/O。载于《第22届 ACM 操作系统原理研讨会论文集（2009）》，SOSP 2009，大天空，蒙大拿州，美国，2009年10月11日至14日（2009年），J. N. Matthews 和 T. E. Anderson 编，ACM，第 133–146 页。[10] Dhoked, S., and Mittal, N. 可恢复互斥的自适应方法。载于 PODC '20：ACM 分布式计算原理研讨会，线上会议，意大利，2020年8月3日至7日（2020年），Y. Emek 和 C. Cachin 编，ACM，第 1–10 页。[11] Dijkstra, E. W. 并发程序控制问题的解答。《ACM 通讯》8卷9期（1965年9月），第 569 页起。[12] Friedman, M., Herlihy, M., Marathe, V. J., and Petrank, E. 面向非易失性内存的持久化无锁队列。载于《第23届 ACM SIGPLAN 并行编程原理与实践研讨会论文集》，PPoPP 2018，维也纳，奥地利，2018年2月24日至28日（2018年），A. Krall 和 T. R. Gross 编，ACM，第 28–40 页。[13] Friedman, M., Petrank, E., and Ramalhete, P. Mirror：让无锁数据结构实现持久化。计算机协会，美国纽约州纽约市，2021年，第 1218–1232 页。[14] Golab, W., and Hendler, D. 次对数时间的可恢复互斥。载于《ACM 分布式计算原理研讨会论文集》（美国纽约州纽约市，2017年），PODC '17，ACM，第 211–220 页。[15] Golab, W., and Hendler, D. 系统级故障下的可恢复互斥。载于《2018年 ACM 分布式计算原理研讨会论文集》（美国纽约州纽约市，2018年），PODC '18，ACM，第 17–26 页。[16] Golab, W., and Ramaraju, A. 可恢复互斥： [扩展摘要]。载于《2016年 ACM 分布式计算原理研讨会论文集》（美国纽约州纽约市，2016年），PODC '16，ACM，第 65–74 页。[17] Izraelevitz, J., Mendes, H., and Scott, M. L. 全系统崩溃故障模型下持久内存对象的线性化。载于《分布式计算——第30届国际研讨会，DISC 2016，法国巴黎，2016年9月27日至29日，论文集》（2016年），C. Gavoille 和 D. Ilcinkas 编，《计算机科学讲义》第 9888 卷，Springer，第 313–327 页。[18] Jayanti, P., and Jayanti, S. 面向缓存一致性和 DSM 的确定性常数摊销 RMR 可中止互斥锁。《ACM 并行计算汇刊》8卷4期（2021年），22:1–22:26。[19] Jayanti, P., Jayanti, S., and Joshi, A. 仅使用 FASAS 实现的最优可恢复互斥。载于《第6届国际网络系统会议》（2018年），NETYS 2018，Springer，Cham。[20] Jayanti, P., Jayanti, S., and Joshi, A. 一种在缓存一致性和 DSM 上均具有次对数 RMR 的可恢复互斥锁算法。载于《2019年 ACM 分布式计算原理研讨会论文集》（美国纽约州纽约市，2019年），PODC '19，计算机协会，第 177–186 页。[21] Jayanti, P., and Joshi, A. 具有无等待恢复的可恢复 FCFS 互斥。载于第31届国际分布式计算研讨会（2017年），DISC 2017，第 30:1–30:15 页。[22] Jayanti, P., and Joshi, A. 具有可中止性的可恢复互斥。载于《网络系统》（Cham，2019年），M. F. Atig 和 A. A. Schwarzmann 编，施普林格国际出版公司，第 217–232 页。[23] Katzan, D., and Morrison, A. 具有次对数 RMR 复杂度的可恢复、可中止和自适应互斥。载于第24届国际分布式系统原理会议，OPODIS 2020，2020年12月14日至16日，法国斯特拉斯堡（线上会议）（2020年），Q. Bramas、R. Oshman 和 P. Romano 编，LIPIcs 第 184 卷，Schloss Dagstuhl - Leibniz-Zentrum f¨ur Informatik，第 15:1–15:16 页。

[24] Kolli, A., Pelley, S., Saidi, A., Chen, P. M., and Wenisch, T. F. 面向持久化内存的高性能事务。载于《编程语言与操作系统架构支持》第21届国际会议论文集（美国纽约州纽约市，2016年），ASPLOS '16，计算机协会，第399–411页。[25] Lamport, L. Dijkstra并发编程问题的新解。《ACM 通讯》17卷8期（1974年8月），453–455页。[26] Lee, H. 采用取值与存储对象的DSM模型上的本地自旋互斥算法。硕士论文，多伦多大学，2003年。[27] Li, N., and Golab, W. 简短公告：可恢复共享对象的可检测顺序规范。载于2021年ACM分布式计算原理研讨会论文集（美国纽约州纽约市，2021年），PODC'21，计算机协会，第557–560页。[28] Ramaraju, A. RGLock：适用于非易失性主存系统的可恢复互斥。硕士论文，滑铁卢大学，2015年。[29] Raoux, S., Burr, G. W., Breitwisch, M. J., Rettner, C. T., Chen, Y.-C., Shelby, R. M.,Salinga, M., Krebs, D., Chen, S.-H., Lung, H.-L., et al. 相变随机存取存储器：一种可扩展技术。IBM研究与开发杂志 52卷4/5期（2008年），465页。[30] Strukov, D. B., Snider, G. S., Stewart, D. R., and Williams, R. S. 寻得失落的忆阻器。《自然》453卷7191期（2008年），80页。[31] Tehrani, S., Slaughter, J. M., Deherrera, M., Engel, B. N., Rizzo, N. D., Salter, J., Durlam, M., Dave, R. W., Janesky, J., Butcher, B., et al. 采用磁性隧道结的磁阻随机存取存储器。《IEEE会议录》91卷5期（2003年），703–714页。

采用原子重置的 Lee 标准（不可恢复）互斥锁

我们介绍 Lee 提出的两种广为人知的标准（不可恢复）队列锁，并为它们配备原子重置方法。第一种锁见图 6，在 CC 模型下的 RMR 复杂度为 $O(1)$，并用于第 2 节 RME 锁的三个基础锁。第二种锁见图 7，在 CC 和 DSM 模型下的 RMR 复杂度均为 $O(1)$，并用于第 4 节 RME 锁的三个基础锁。

我们用四个集合来表示锁的状态 ℓ：ℓ.TrySet、ℓ.CSSet、ℓ.ExitSet和 ℓ.CrashSet。前三个集合分别包含处于 ℓ的尝试区段、临界区和退出区段的进程。发生系统级崩溃时，这些集合中的进程会被移至 ℓ.CrashSet （另外三个集合则变为空集）。因此， ℓ.CrashSet 包含崩溃发生时正在使用该锁的进程。对 ℓ执行原子重置时，四个集合都会被设为 $\emptyset$，表示 ℓ 再次处于可供使用的状态，就像一把全新的互斥锁；此外，重置可以随时执行。

该使用模式陈述了一个显而易见的事实：进程 p 只有在不属于这四个集合中的任何一个时，才可以进入 ℓ.TrySet ；当 p 离开 ℓ.TrySet （即完成尝试区段后），它会进入 ℓ.CSSet；当 p 离开 ℓ.CSSet 时，它会进入 ℓ.ExitSet；当 p 离开 ℓ.ExitSet 时，它便不属于任何集合。此外， p 只有在分别属于 ℓ.TrySet、ℓ.CSSet 或 ℓ.ExitSet 时，才能执行尝试区段、临界区或退出区段中的一步。

Lee 锁的性质总结如下。

定理 A.1。 假设进程遵循上述使用模式，则图 6 和图 7 中的 Lee 锁满足以下性质：

- 互斥：$|\ell.\text{CSSet}| \leq 1$
- 无饥饿性：在每次公平运行中，如果某个进程 p 执行 ℓ.尝试$_p$()，并且在该次尝试方法执行的整个期间 ℓ.CrashSet 为空，那么该方法最终会完成，使 p 进入临界区。

- 有界退出：如果某个进程 p 执行 ℓ.退出$_p$()且未发生崩溃，那么该进程会在其自身执行常数步后 p 从该方法返回。。
- 原子重置： ℓ.Reset$_p$() 是原子的，并会返回到其初始状态 ℓ 。
- RMR 和空间复杂度：图 6 中的锁在 CC模型中的 RMR 复杂度为 $O(1)$，图 7 中的锁在 CC 和 DSM 模型中的 RMR 复杂度均为 $O(1)$，且这两种锁对每个进程的空间复杂度均为 $O(1)$。

Lee算法还具有一个额外性质 [18], ，说明如下。假设发生系统级崩溃时，一组进程 A正在使用某个锁 ℓ 。随后进程重新启动时，假设一组进程 B （与 A不相交）调用尝试方法。由于进程 B 会排在那些因先前崩溃而丢失状态的进程 A 后面，因此它们可能永远困在尝试区段中。然而，如果 A中的所有进程在重启后都执行退出区段，那么 B 中的某个进程（非正式地说，是 B中排在队列最前面的进程）将完成尝试区段并返回；此外，如果 B中的每个进程在从尝试区段返回后都执行退出区段，那么 A中的所有进程都会依次从尝试区段返回。崩溃的进程 A在 ℓ.CrashSet中执行退出区段，违反了使用模式。因此，互斥可能会遭到破坏，但在为 DSM模型设计可恢复互斥算法时， B 中没有进程会陷入停滞这一特性很有用。以下定理总结了迄今为止所说明的性质。

定理A.2 （释放性质）。 设 ℓ 是图 6 或 7 中的 Lee 锁。假设 ℓ.CrashSet $=A \neq \emptyset$ 和 ℓ.TrySet $\neq \emptyset$。如果 A 中的每个进程都执行退出区段，并且 ℓ.TrySet 中的每个进程在离开 ℓ.TrySet 时（如果离开的话）都执行退出区段，那么 ℓ.TrySet 中的每个进程最终都会离开ℓ.TrySet。

共享变量（存储在非易失性存储器中）： Tail 是指向某个位置的指针，该位置包含一个布尔值或 $\perp$； Tail 初始化为 $\perp$。**进程 p的持久变量（存储在非易失性存储器中）：** 节点$_p$[0], 节点$_p$[1] 是指向布尔值的指针，初始值任意。标记$_p \in \{0,1\}$，初始值任意。

```
procedure ℓ.try_p()10. 标记_p = 1 –标记p11.  *节点_p[标记_p] ← false12.  prev_p ← FAS(Tail,节点_p[标记_p])13. 如果 prev_p ≠ ⊥ ，则等待直到 *prev_p = true procedure ℓ.exit_p()14.  *节点_p[标记_p] ← true procedure ℓ.reset_p()15.  Tail ← ⊥
```

图 6： 适用于缓存一致性机器的 Lee 锁 [26], ，其中添加了一行用于重置。此处展示的是进程 p 的代码。

B 算法正确性

我们首先在第 B.1 节介绍算法满足的不变量，然后利用该不变量在第 B.3 节证明算法的正确性。

B.1 算法的不变量

该算法的不变量见图 2。

共享变量（存储于非易失性存储器中）： Tail 是一个指向某个位置的指针，该位置包含一个指针或 ⊥； Tail 被初始化为 ⊥。**进程本地的持久变量 p（存储于非易失性存储器中）：** 节点$_p$[0]、节点$_p$[1] 是指向内存字的指针，其初始值任意。标记$_p \in \{0,1\}$的初始值任意。

```
    procedure ℓ.try_p()
1.  Face_p = 1 − 标记_p
2.  *节点_p[标记_p] ← ⊥
3.  prev_p ← FAS(Tail、节点_p[标记_p])
4.  如果 prev_p ≠ ⊥ 那么
5.      Go_p ← false
6.      如果 FAS(*prev_p,&Go_p) = token 则返回
7.      等待，直到 Go_p

    procedure ℓ.exit_p()
8.  ptr_p ← FAS(*节点_p[标记_p], token)
9.  如果 ptr_p ≠ ⊥ 则 *ptr_p ← true

    procedure ℓ.reset_p()
10. Tail ← ⊥
```

图 7： Lee 的缓存一致性和 DSM 机器队列锁 [26], , 其中添加了一行用于重置。显示的是进程 p 的代码。

B.2 基础锁所需的证明

在本节中，我们证明一些引理，以证明我们的 RME 算法会按照锁所要求的方式调用基础锁。这确保基础锁的行为符合预期。引理 B.1–B.2 论证了主算法按照基础锁所预期的转换规则运行。引理 B.5–B.7 对论证基础锁的无饥饿性前置条件成立非常重要。也就是说，没有进程会永远停留在临界区（CS）中：如果任意进程 q 在任意时刻位于 ℓ.CSSet $t' > t$，则 q 在 t'之后位于 ℓ.ExitSet 。为说明这一点，我们注意到，从算法可以看出，一个进程在整个运行过程中只能在特定位置循环。当一个进程 p 位于第 **4** 行或第 **10** 行时，我们知道 $p \in$ Lock[S$_p$%3].TrySet。然而，当 $PC_p \in [\mathbf{5}, \mathbf{6}]$时，可能出现 $p \in$ Lock[S$_p$%3].CSSet；或者，当$PC_p \in \{\mathbf{11}, \mathbf{13}\}$时，当然会出现 $p \in$ Lock[S$_p$%3].CSSet。当一个进程正在重置锁 ℓ时，没有其他进程会执行该锁的 ℓ.try() 或ℓ.exit() 方法中的一步。更准确地说，对于每个锁 ℓ 和每个进程 q，如果对于某个进程 p而言， PC_p指向 ℓ.reset$_p$()中的一步，那么 PC_q 不会指向 ℓ.try$_q$()或 ℓ.exit$_q$()中的一步。这正是引理 B.8 所论证的内容。下面的引理 B.8 基本上意味着，一旦某个进程 p 完成 ℓ.reset$_p$()的执行，对于任意基础锁 ℓ和任意进程 q，无论 $q = p$ 与否，都会回到执行 ℓ.try$_q$()、临界区（CS）和 ℓ.exit$_q$()方法的模式，从调用 ℓ.try$_q$()开始。这是因为 ℓ.reset$_p$()会重置该锁，将 ℓ.TrySet、 ℓ.CSSet和 ℓ.ExitSet 设为空集合；并且根据引理 B.1–B.2，我们知道主算法遵守基础锁所要求的转换。

引理 B.1. 进程 p 只有在 p 不属于三个集合中的任何一个时，才可以调用 Lock[Seq%3].$try_p()$ 。

证明。 进程 p可以在第 **4** 行或第 **10** 行调用 Lock[Seq%3].尝试$_p$()。根据条件**4**，当 $PC_p = \mathbf{3}$、 $p \notin$ Lock[Seq%3].Set 时，我们便得到了所述结论。当 $PC_p = \mathbf{9}$ 时，根据条件**11**， $S_p =$ Seq $- 1$；因此再次根据条件**4**，我们得到了所述结论。 □

引理 B. **2.** 进程 p 只有在 p 属于 Lock[Seq%3].CSSe 时，才可以调用 Lock[Seq%3].$exit_p()$ T.

证明。 进程 p 可能会在第 **18** 行调用 Lock[Seq%3].退出$_p$()。只有当 $S_p =$ Seq 在第 **17** 行执行一步时，才可能发生这种情况。在这种情况下，根据条件 **7**， $p \in$ Lock[S$_p$%3].CSSet，这意味着当 p 调用 Lock[Seq%3].退出$_p$() 时，该论断成立。 □

引理 B.3。 存在一个常数 c，使得如果进程 p 处于临界区（即，当$PC_p = \mathbf{16}$时）发生崩溃，那么在执行了 c 个没有崩溃步骤介入的步骤之前， p 会重新进入临界区。

证明。 如果进程 p 处于临界区时发生崩溃，那么在崩溃后的紧接配置中，状态$_p$ = 从临界区恢复 且 $PC_p = \mathbf{1}$。此外，在崩溃后的配置中， CSOWNER $= p$ 仍然成立，因为根据条件 9，它在崩溃前就具有该值，而且崩溃不会改变CSOWNER的值。根据控制流的描述，我们知道 $PC_p = \mathbf{21}$ 最终会发生。我们注意到，一旦$PC_p = \mathbf{21}$， PC_p就会在其自身执行常数个正常步骤后变为 **26**；这一点可通过检查算法得知，因为从第 **21** 行到第 **26**行的执行路径中没有循环。

现在我们通过反证法论证， CSOWNER 从 p在临界区中崩溃后立即形成的配置开始，直到 p 执行第 26 行之前的配置为止，始终保留值 p 。因此，反设 CSOWNER 在 p于临界区中崩溃之后、但在 p 执行第 26 行之前，从 p 变成了其他值。设进程 r 是本次运行中最早将 CSOWNER 的值从 p 改为其他值的进程。 CSOWNER 的值只会在 try() 和 exit() 方法中改变；由于 p 要么位于第 1 行，要么位于 recover$_p$() 方法中，因此 $p \neq r$。注意，如果 r 因第 7 行或第 12 行的 CAS 操作而改变了它，那么在 r执行这一步之前的配置中， CSOWNER $=\perp$ ，因此 r 就不是最早将 CSOWNER 的值从 p 改为其他值的进程。类似地，如果 r 要执行第 14 行，根据条件 9，当 $PC_r = \mathbf{14}$时， CSOWNER $=\perp$ ，因此 r 就不是最早将 CSOWNER 的值从 p 改为其他值的进程。因此，通过检查该算法可知， CSOWNER只能因 r执行第 19 行而变为 $\perp$ 。设 C 为 $PC_r = \mathbf{19}$时的配置。我们知道，在 C CSOWNER $= p$中。由于在 C中 $PC_r = \mathbf{19}$ ，根据条件 9 可知，在 C中 CSOWNER $= r$。因此， CSOWNER 在 C中有两个不同的值，这就产生了矛盾。由此可知， CSOWNER 从 p在临界区中崩溃后立即形成的配置开始，直到 p 执行第 26 行之前的配置为止，始终保留值 p 。

由此可知，如果进程 p 处于临界区（即 $PC_p = \mathbf{16}$时）发生崩溃，那么 p 会在执行完 c 个没有崩溃步骤介入的步骤之前重新进入临界区 □

引理 B.4. 假设存在一次公平运行，在该运行中， CSOWNER $= p$ 在时间 t， CSOWNER $=\perp$ 在时间 $t' > t$ 。

证明。 假设存在一次公平运行，其中 CSOWNER $= p$ 在时间 t。根据条件 9，Active$_p = true$ 和$PC_p \in \{$**15-19**， **21-26**$\}$，或者结合条件 9 和 10， $PC_p = \mathbf{1}$ 且 状态$_p \notin \{$正常， 从余数部分恢复$\}$.
如果 $PC_p = \mathbf{1}$ 且 状态$_p \notin \{$正常， 从余数部分恢复$\}$，根据控制流的描述，我们知道 $PC_p = \mathbf{21}$ 最终会发生。检查算法可知，在以下任一情况下， p 最终都会执行第 **26** 行：Active$_p = true$ 且 $PC_p \in \{$**21-26**$\}$，或者 $PC_p = \mathbf{1}$ 且 状态$_p \notin \{$正常，从余数部分恢复$\}$。根据与引理 B.3 证明中类似的论证可知， CSOWNER会保留值 p ，直至其执行第 **26** 行并重新进入临界区，且 $PC_p = \mathbf{16}$ 最终会发生。一旦 $PC_p \in \{$**15-19**$\}$ ，不难看出， p 会在之后的某个时刻于第 **19** 行将 CSOWNER 设为 $\perp$ ，从而满足该断言。 □

引理 B.5. 如果在时间 t， p 位于第 **5–6** 行，那么在某个时间 $t' > t$， p 位于不同的行。

证明。 假设 p 位于第 **5-6** 行。根据条件 11， $\mathrm{S}_p \in \{\mathrm{SEQ} - 1, \mathrm{SEQ}\}$。我们按以下方式讨论各种情况：

情况 1 假设 $\mathrm{S}_p = \mathrm{SEQ} - 1$。根据条件 12， $\mathrm{STOP}[\mathrm{S}_p\%3] = true \vee (\exists q, PC_q = \mathbf{25} \wedge \mathrm{S}_q = \mathrm{S}_p)$ 如果 $\mathrm{STOP}[\mathrm{S}_p\%3] = true$， p 会在下次执行第 **5** 行时注意到这一点，并跳出循环。否则，我们有 $\exists q, PC_q = \mathbf{25} \wedge \mathrm{S}_q = \mathrm{S}_p$。在第 **25** q 行将 $\mathrm{STOP}[\mathrm{S}_p\%3]$ 设为 $true$， p 会在下次执行第 **5** 行时注意到这一点，因此跳出循环。

情况 2 假设 $\mathrm{S}_p = \mathrm{SEQ}$。为便于论证，我们假设在当前通行中，$PC_p = \mathbf{5}$ 首次出现之后以及 $\mathrm{SEQ} = \mathrm{S}_p$ 此后永远运行期间都没有发生崩溃；否则，由于上述某种情况， p 会位于不同的行，结论显然成立。注意，根据条件 3，$\mathrm{STOP}[\mathrm{SEQ}\%3]$ 将永远保持为 $false$ ，这会阻止 p 跳出第 **5** 行的循环。

如果 CSOWNER $=\perp$ 和 p 注意到它下一次执行第 **6** 行时， p 会跳出循环。否则，假设 CSOWNER $= q$。根据条件 9 第一部分的逆否命题可知 $q \neq p$。根据引理 B，4 CSOWNER 稍后会变为 $\perp$ 。如果 p 注意到在第 **6** 行 CSOWNER $=\perp$，则结论成立。否则，我们如下论证： CSOWNER 在如上所述变为 $\perp$ 之后，最多只能有一次取值为 $r \neq \perp$ ，随后当 p 最终注意到它时，又会变为 $\perp$

因此，假设 CSOWNER 在如上所述变为 $\perp$ 后立即变为值 $r \neq \perp$，但在 p 于第 6 行注意到它之前。注意，r 的这个值只能由 r 自身在第 7、12 或 14 行写入 CSOWNER。因此，$r \neq p$；否则 p 已经跳出循环，结论成立。首先注意，CSOWNER不可能在第 12 或 14 行发生变化。这是因为，根据条件 7 和条件 11，在第 12 行以及 14 $S_r = SEQ$ 和 $r \in LOCK[S_r\%3].CSSET$。由于$PC_p \in \{6\}$且 $S_p = SEQ$（这意味着根据条件 3，$STOP[S_p\%3] = false$），根据条件 7，我们已经有$p \in LOCK[S_p\%3].CSSET$。因此，$LOCK[SEQ\%3]$的互斥性质意味着 r 不能与 $S_r = SEQ$同时处于 $LOCK[S_r\%3].CSSET$ 中。由此可知，值 r 只能在第 7 行写入 CSOWNER。根据条件 11，并使用上述相同的论证，可知 $S_r = SEQ-1$；而根据条件 7，当 r 在第 7 行将自己的名字写入CSOWNER时，$r \in LOCK[S_r\%3].CSSET$。一旦 r 在第 7 行将值 r 通过 CAS 写入 CSOWNER，它就会进入临界区，完成临界区操作并调用 exit$_r$()。在第 17 r 行，$S_r = SEQ-1 \neq SEQ$，因此它永远不会在第 18 行调用 $LOCK[S_r\%3]$.exit$_r$()。因此，r 将永远留在 $LOCK[S_r\%3].CSSET$ 中；根据基础锁 $LOCK[S_r\%3]$, 的互斥性质，之后没有其他进程会执行第 7 行。因此，一旦 r 在第 19 行将 $\perp$ 写入 CSOWNER，就没有其他进程会在第 7、12 或 14 行将自己的名字写入 CSOWNER。因此，p 在第 6 行注意到 CSOWNER $=\perp$，并跳出循环。

我们注意到，根据上述两种情况，该断言成立。 □

引理 B.6. 如果在时间 t，p 位于**第11行**，那么在某个时间 $t' > t$，p 位于另一行。

证明。我们首先注意到，根据条件 11，当 $PC_p = 11$ 时，$S_p = SEQ$。该断言可由上述引理 B.5 证明中情形 2 所给出的相同论证得出。 □

引理 B.7. 如果在时间 t，p 位于**第13行**，那么在某个时间 $t' > t$，p 位于另一行。

证明。假设在时间 t，p 位于第 13 行。我们注意到，根据条件 11，当 $PC_p = 13$ 时，$S_p = SEQ$。如果 CSOWNER $= q \neq \perp$，根据条件 9，$q \neq p$。由引理 B.4 可知，CSOWNER $=\perp$ 会在稍后的时间t'。因此，假设 CSOWNER $=\perp$，根据条件 14，我们得到 $\forall r$, ($S_r \neq SEQ - 1 \vee PC_r \in \{1, 4, 9, 20, 21, 28\} \vee (PC_r = 5 \Rightarrow STOP[S_r\%3] = true)$). 这意味着一旦 CSOWNER $=\perp$，就不存在满足 $S_r = SEQ - 1$ 且$PC_r = 7$的进程 r。由于 $PC_p = 13$, 根据条件 7，$p \in LOCK[S_p\%3].CSSET$。由于 $S_p = SEQ$，再次根据条件 7，不存在满足 $S_q = SEQ$ 且 PC_q 为 7、12 或 14 的进程 q。因此，一旦 p 位于第 13 行，只要 p 没有注意到 CSOWNER为 $\perp$，CSOWNER就不会从 $\perp$ 改变为其他值。由此可知，p 会从第 13 行的循环中退出并转到第 14 行。 □

引理 B.8. 对于每把锁 ℓ 和每个进程 q，如果对于某个进程 p，PC_p 在时间 t指向 $\ell.reset_p()$ 中的某一步，那么在 t，PC_q 不会指向 $\ell.try_q()$ 或 $\ell.exit_q()$ 中的某一步。

证明。假设在时间 t，$PC_p = 22$ 与 $S_p = SEQ$。反设在 t 存在一个进程 q，使得 $PC_q = 4$与 $S_q = SEQ - 1$。由条件 13 可知，对于每个进程 r，若 $PC_r \in \{22, 25\}$，则 $S_r = SEQ - 1$，这与我们最初的假设 $PC_p = 22$与 $S_p = SEQ$矛盾。因此，不存在进程 q 使得在 t，$PC_q = 4$ 与 $S_q = SEQ - 1$。用同样的论证可得，不存在进程 q使得 $PC_q = 18$与 $S_q = SEQ - 1$。由条件 11 可知，若对于某个进程q，$PC_q = 10$，则 $S_q = SEQ$。因此，当 p 在时间 t指向 $LOCK[(SEQ - 1)\%3]$.重置$_p$() 中的某一步时，不存在进程 q指向 $LOCK[(SEQ - 1)\%3]$.尝试$_q$() 或 $LOCK[(SEQ - 1)\%3]$.退出$_q$() 中的某一步。

假设 $PC_p = 22$在时间 t与 $S_p = SEQ-1$。根据条件 11，对于任意进程 q，如果 $PC_q \in \{4, 10, 18\}$，则 $S_q \in \{SEQ - 1, SEQ\}$。因此，当 p 指向 $LOCK[(SEQ - 2)\%3]$.重置$_p$()（或等价地，指向 $LOCK[(SEQ + 1)\%3]$.重置$_p$()）中的某一步时，在时间 t，不存在进程 q 指向 $LOCK[(SEQ - 2)\%3]$.尝试$_q$() 或 $LOCK[(SEQ - 2)\%3]$.退出$_q$() 中的某一步。

因此，结论由上述内容得出。 □

B.3 性质证明

引理 B.9（M　互斥）。在任意运行的任何配置中，至多有一个进程处于临界区 .

证明。反设有两个进程 p 和 q 在同一配置中处于临界区。因此， $PC_p = \mathbf{16}$ 和 $PC_q = \mathbf{16}$ 在同一配置中。根据条件 9，我们还得到 $\text{CSOWNER} = p$ 以及 $\text{CSOWNER} = q$ 在同一配置中。因此， CSOWNER 在同一配置中有两个不同的值，这就产生了矛盾。

□

引理 B.10（有界退出）。存在一个整数 b，使得在任意运行中，如果任一进程 p 调用并执行 $exit_p()$ 且未发生崩溃，则该方法至多经过 b 步 p即可完成。

证明。我们注意到，基础算法满足有界退出性质。通过检查算法可知，图1中的 exit$_p$() 过程包含有界数量的步骤，且没有循环。因此命题成立。

□

引理 B.11（无饥饿性）。在每个公平运行中，如果一个进程 p 执行尝试方法，且在此次尝试方法执行期间没有发生崩溃，则该方法最终会完成，使 p进入临界区。

证明。由于运行中只有有限个崩溃步骤，为便于论证，取一个运行，并从该运行中选取最早的配置，使得所有崩溃都已发生。我们需要证明：如果进程 p 在时间 t处于 try$_p$() 中，那么它在另一时间 $t' > t$处于临界区。从检查算法可知，当 try$_p$() 方法返回时，它会使 p 处于临界区，因为每条返回语句都返回值 IN CS。因此，我们需要证明进程 p 不会永远卡在第 4、5–6、10、11 或 13 行，这将确保 p 确实会从 try$_p$() 返回。根据引理 B.5、B.6 和 B.7，我们知道 p 最终会分别通过第 5–6、11 和 13 行。当 $PC_p = 4$时，根据条件 11，$\text{S}_p \in \{\text{SEQ} - 1, \text{SEQ}\}$。如果 $\text{S}_p = \text{SEQ} - 1$，根据条件 12， $\text{STOP}[\text{S}_p\%3] = true \vee (\exists q,$ $PC_q = \mathbf{25} \wedge \text{S}_q = \text{S}_p)$。在这两种情况下， $\text{STOP}[\text{S}_p\%3] = true$ 最终都会发生，而 p 会在第 4 行的等待循环中注意到这一点，并通过该行。因此，我们假设每当 p 执行第 4 行或第 10 行时， $\text{S}_p = \text{SEQ}$，因为根据条件 11，这是唯一的其他可能性。接下来，我们将论证 $\text{LOCK}[\text{SEQ}\%3]$ 的无饥饿性满足，从而意味着 p 会通过第 4 行或第 10 行。通过检查算法可知，任何会在第 22 行执行 reset() 的进程，都会因 $\text{LOCK}[\text{SEQ} - 1]$或 $\text{LOCK}[\text{SEQ} - 2]$而执行该操作，因为根据条件 11 $\text{S}_p \in \{\text{SEQ} - 1, \text{SEQ}\}$，对于任意进程 p，都有 $PC_p = \mathbf{22}$。由此可知，自最近一次崩溃以来，没有进程会执行 $\text{LOCK}[\text{SEQ}\%3]$.reset()，从而满足基础锁的无饥饿性的第一个条件。显然，基础锁的无饥饿性的第二个条件也得到满足。引理 B.5、B.6 和 B.7 确保基础锁的无饥饿性的最后一个条件也得到满足。因此，我们知道 $\text{LOCK}[\text{SEQ}\%3]$ 满足无饥饿性。于是，根据基础锁的无饥饿性，一个进程 p 会通过第 4 行或第 10 行。由此可知命题成立。

□

引理 B.12 (临界区重入)。在任何一次运行中，如果进程 p 在临界区内崩溃，那么在 p 随后重新进入临界区之前，其他进程都不会进入临界区。

证明。该结论可由引理 B.3 和 B.9 直接得出。 □

引理 B.13 (有界恢复至临界区/退出)。存在一个整数 b，使得在任何一次运行中，如果任一进程 p 在不发生崩溃且 状态$_p \in$ {从临界区恢复、从退出状态恢复}的情况下执行 $recover_p()$，则该方法至多在 p的 b 步内完成。

证明。检查 recover$_p$() 方法可知，当进程 p 在不发生崩溃的情况下执行 recover$_p$() 时，该方法会在常数步数内完成。 □

引理 B.14 (快速恢复至剩余区)。存在一个常数 b（与 $|\mathcal{P}|$无关），使得在任何一次运行中，如果任一进程 p 在不发生崩溃且 状态$_p \in$ {正常、从余数部分恢复}的情况下执行 $recover_p()$，则该方法至多在 p的 b 步内完成。

证明。 检查 recover$_p$() 方法可知，当进程 p 执行 recover$_p$() 且未发生崩溃时，该方法会在常数步数内完成。

□

引理 B.15 (有界恢复至剩余区)。 存在一个整数 b，使得在任何一次运行中$recover_p()$，由状态为状态$_p =$ 从尝试区恢复的进程 p 执行并返回处于剩余区时， p 必须在至多 b 步内完成对 $recover_p()$ 的该次执行。

证明。 检查 recover$_p$() 方法可知，当进程 p 执行 recover$_p$() 且未发生崩溃时，该方法会在常数步内完成。 □

B.4 RMR 复杂度

我们知道，基础锁在执行过程 try$_p$()、exit$_p$() 和 reset$_p$() 时会产生 $O(1)$ 的 RMR 复杂度。因此，行 **22**、**4**、**10** 和 **18** 的执行需要 $O(1)$ 个 RMR。检查算法可知，图1中算法的其余各行在 CC 机器上会产生 $O(1)$ 个 RMR。因此，进程执行图1中的算法时，每次通行会产生 $O(1)$ 个 RMR。

B.5 不变量证明

不变量的证明见附录 C

B.6 缓存一致性算法主定理

下面的定理总结了缓存一致性算法的结果。

定理 B.16. 图*1*中的算法可在 *CC* 机器上为任意且未知数量的进程解决 *RME* 问题。它满足互斥、有界退出、有界临界区重入和无饥饿性。该算法在 *CC* 机器上的 *RMR* 复杂度为 $O(1)$ 每次通行。

C 不变式证明

引理 C.1. 图*1*中的算法满足图 *2* 中所述的不变量（即所有条件的合取），也就是说，在该算法的每次运行中，不变量在每个配置中都成立。

证明. 我们用归纳法证明该引理。具体而言，我们证明 (i) 基础情形： 不变量在初始配置中成立；以及 (ii) 归纳步骤： 若不变量在配置 C 中成立，且某个进程执行一步，使配置从 C 变为 C'，则不变量在 C'中成立。

在初始配置中，我们有 $\text{SEQ} = 1$，每个 $\text{LOCK}[0\ldots2]$ 都处于其初始状态，$\text{STOP}[0\ldots2]$ 全都处于 $false$， $\text{CSOWNER} = \perp$， $\forall p$，活动状态$_p = false$，以及 $\text{S}_p = 1$。注意，由于锁处于初始状态，因此 $\forall i \in [0,2]$， $\text{LOCK}[i]$.状态 $= (\phi,\ \phi,\ \phi)$。由于所有进程都处于 remainder 区，$\forall p$， $PC_p = \mathbf{21}$。由此可知，所有条件在初始配置中都显然成立。因此，基础情形得证。

对于归纳步骤，设 C 是任意运行中的任意配置。归纳假设指出，条件 1 至 14 在 C 中均成立。设 (C, σ, C') 是 $\mathcal{P}$ 中某个进程执行的任意一步（正常步或崩溃步）。现在，我们通过论证条件 1 至 14 在 C' 中均成立，来完成归纳步骤。

证明中使用以下记号：(i) IH 表示归纳假设；对于所有 $i \in [1,14]$，IH:i 表示 IH 中指出不变量的条件 i 在 C 中成立的部分；(ii) 若 D 是任意配置，且 x 是任意变量，则 $D.x$ 表示 x 在 D 中的值。

1. 证明条件 1 在 C' 中成立
   SEQ 仅在某个进程执行第**24**行时才会改变。进程 p 执行该行时，根据归纳假设**11**，$\text{S}_p \in \{\text{SEQ} - 1, \text{SEQ}\}$。若 $\text{S}_p = \text{SEQ}$，则 SEQ 增加 1；否则保持不变。因此，$C'.\text{SEQ}$ 始终大于或等于 1。

   同样， S_p 会在第**3** 行或第**9**行发生变化。当 S_p 在第**3**行发生变化时，它取 SEQ的值，因此我们有 $C'.\text{S}_p \le C'.\text{SEQ}$。当 S_p在第**9**行发生变化时，根据归纳假设**11**， $\text{S}_p = \text{SEQ} - 1$。因此，根据这一步$\text{S}_p = \text{SEQ}$ ，从而有 $C'.\text{S}_p \le C'.\text{SEQ}$。
2. 证明 C' 中条件 2 成立我们将如下分别证明条件 2 的每个合取项。

（a）。 $C'.\text{Lock}[(\text{Seq}+1)\ \%3].\text{State} = (\phi,\phi,\phi)$。

每当调用基础锁的 try$_p$() 或 exit$_p$() 时，调用位置都是第 4、10 或 18 行。根据归纳假设11，$\text{S}_p \in \{\text{Seq}-1, \text{Seq}\}$ 在这些行处，因此这些行不会影响 $\text{Lock}[(\text{Seq}+1)\%3].\text{TrySet}$、 $\text{Lock}[(\text{Seq}+1)\%3].\text{CSSet}$ 或 $\text{Lock}[(\text{Seq}+1)\%3].\text{ExitSet}$。只有在第 24 行的 $\text{S}_p = \text{Seq}$ 时， Seq 才会改变。根据归纳假设 IH:2，我们知道当$PC_p = \mathbf{24}$时， $\text{Lock}[(\text{S}_p-1)\%3].\text{State} = (\phi,\phi,\phi)$。由于这一步会使 Seq 增加 1，因此可得 $(C'.\text{S}_p - 1)\%3 = ((C'.\text{Seq} - 1) - 1)\%3 = (C'.\text{Seq} - 2)\%3 = (C'.\text{Seq}+1)\%3$。因此，在第 24 行执行 p 这一步且 $\text{S}_p = \text{Seq}$ 时， $\text{Lock}[(C'.\text{Seq}+1)\%3].\text{State} = (\phi,\phi,\phi)$。

(b). $C'.PC_p \in \{\mathbf{23}, \mathbf{24}\} \Rightarrow \text{Lock}[(C'.\text{S}_p - 1)\%3].\text{State} = (\phi,\phi,\phi)$.

为证明这一蕴含，假设 $C'.PC_p \in \{\mathbf{23}, \mathbf{24}\}$。

若 $C.PC_p = \mathbf{22}$，则 p 执行 $\text{Lock}[(\text{S}_p-1)\%3]$. 重置$_p$()，这会将 $\text{Lock}[(\text{S}_p-1)\%3].\text{State}$ 设置为 (ϕ, ϕ, ϕ)。

若 $C.PC_p \in \{\mathbf{23}, \mathbf{24}\}$，根据归纳假设 IH:2 $\text{Lock}[(C.\text{S}_p - 1)\%3].\text{State} = (\phi,\phi,\phi)$，从 C 到 C' 的这一步是由某个 $q \neq p$ 执行的。 q 只能通过执行第 4、10 或 18 行之一来修改锁的 TrySet、 CSSet 或 ExitSet 。根据归纳假设 11， $\text{S}_p \in \{\text{Seq}-1, \text{Seq}\}$ 在 $PC_q \in \{\mathbf{4}, \mathbf{18}\}$ 时成立，而 $\text{S}_p = \text{Seq}$ 在 $PC_q = \mathbf{10}$ 时成立。若 q 执行了一步，其中 $PC_q \in \{\mathbf{4}, \mathbf{18}\}$ 且 $\text{S}_q = \text{Seq} - 1$，那么根据归纳假设13， $C.\text{S}_p = \text{Seq} - 1$。我们注意到， q 执行的这一步不会影响 Seq 或 S_p。因此， q 执行的这一步不会影响 $\text{Lock}[(C.\text{S}_p - 1)\%3].\text{TrySet}$、 $\text{Lock}[(C.\text{S}_p - 1)\%3].\text{CSSet}$ 或 $\text{Lock}[(C.\text{S}_p - 1)\%3].\text{ExitSet}$。因此，根据归纳假设 IH:2， $\text{Lock}[(C'.\text{S}_p - 1)\%3].\text{State} = (\phi,\phi,\phi)$。

如果 q执行了一步，涉及 $PC_q \in \{\mathbf{4}, \mathbf{10}, \mathbf{18}\}$和 $\text{S}_q = \text{Seq}$，那么根据归纳假设11，我们知道 $C.\text{S}_p \in \{\text{Seq}-1, \text{Seq}\}$。由归纳假设 IH:2 可得 $C.\text{Lock}[(\text{S}_p - 1)\%3].\text{State} = (\phi,\ \phi,\ \phi)$；并且，由于 q 执行的这一步不会影响 $\text{Lock}[(\text{Seq}-1)\%3]$或 $\text{Lock}[(\text{Seq}-2)\%3]$, 中任一方的 TrySet、 CSSet或 ExitSet ，因此我们有 $C'.\text{Lock}[(\text{S}_p - 1)\%3].\text{State} = (\phi,\ \phi,\ \phi)$。

3. 证明 C'中条件 3 成立

我们分别证明条件 3 的各个合取项，如下所示。

(a). $C'.\text{Stop}[\text{Seq}\%3] = false$。

只有在行 **23** 和 **25**，才会更改 Stop 中的单元格。行 **23** 将 $\text{Stop}[(\text{S}_p - 1)\%3]$ 设为 $false$，因此无需讨论这种情况。如果 $PC_p = \mathbf{25}$，根据归纳假设11， $\text{S}_p = \text{Seq} - 1$。由此可知，执行行 **25** 不会修改 $\text{Stop}[\text{Seq}\%3]$。

唯一会改变 Seq 的位置是在 $\text{S}_p = \text{Seq}$ 时的第 **24** 行。根据归纳假设 IH:3，我们知道$\text{Stop}[(\text{Seq}+1)\%3] = false$。由于这一步会将 Seq 增加 1，因此可得 $(C'.\text{Seq})\%3 = (C.\text{Seq}+1)\%3$。因此，根据 $\text{S}_p = \text{Seq}$ 时第 **24** 行的 p 这一步， $\text{Stop}[C'.\text{Seq}\%3] = false$。

(b). $C'.\text{Stop}[(\text{Seq}+1)\%3] = false$。

只有在第 **23** 行和第 **25** 行， Stop 中的单元格才会改变。第 **23** 行将 $\text{Stop}[(\text{S}_p - 1)\%3]$ 设置为 $false$，因此无需讨论这种情况。如果 $PC_p = \mathbf{25}$，根据归纳假设11， $\text{S}_p = \text{Seq} - 1$。由此可知，执行第 **25** 行不会修改 $\text{Stop}[(\text{Seq}+1)\%3]$。

唯一会改变 Seq 的地方是在 $\text{S}_p = \text{Seq}$ 时的第 **24** 行。根据归纳假设:3，我们知道当$PC_p = \mathbf{24}$时， $\text{Stop}[(\text{S}_p - 1)\%3] = false$。由于这一步会将 Seq 增加 1，因此可得 $(C'.\text{S}_p - 1)\%3 = ((C'.\text{Seq} - 1) - 1)\%3 = (C'.\text{Seq} - 2)\%3 = (C'.\text{Seq}+1)\%3$。因此，根据 p 在 $\text{S}_p = \text{Seq}$ 时执行的第 **24** 行中的步骤， $\text{Stop}[(C'.\text{Seq}+1)\%3] = false$。

(c). $C'.PC_p = \mathbf{24} \Rightarrow \text{Stop}[(C'.\text{S}_p - 1)\%3] = false$。

为证明这个蕴含，假设 $C'.PC_p = \mathbf{24}$。

在 $C.PC_p = \mathbf{23}$ 的情况下， p 会在这一步中将 $\text{Stop}[(\text{S}_p - 1)\%3]$ 设置为 $false$。

对于情形 $C.PC_p = \mathbf{24}$，根据归纳假设 IH:3 $\text{Stop}[(C.\text{S}_p - 1)\%3] = false$，从 C到 C'的步是由

某个 $q \neq p$。 q 只能在第 **25** 行将单元格从 STOP 设为 $true$；具体而言， q 在第 **25** 行将 $\text{STOP}[\text{S}_q\%3]$ 设为$false$。当 $PC_q = \textbf{25}$时，根据归纳假设**11**，有 $\text{S}_q = \text{SEQ}-1$。由归纳假设**13**可知 $\text{S}_p = \text{SEQ}-1$。

因此 $\text{STOP}[(C.\text{S}_p - 1)\%3] = \text{STOP}[(C'.\text{S}_p - 1)\%3] = false$。

4。证明 C' 中条件**4**成立。我们分别证明如下条件**4**的各个合取项。

(a). (C'.活动状态$_p = false \vee C'.\text{S}_p < C'.\text{SEQ} \vee C'.PC_p \in \{\textbf{3}, \textbf{19}, \textbf{20}, \textbf{26}, \textbf{27}\}$)
$\Rightarrow p \notin \text{LOCK}[C'.\text{SEQ}\%3].\text{SET}$。

We 将蕴含分成以下子部分，并论证每个部分的正确性。

i. C'.活动状态$_p = false \Rightarrow p \notin \text{LOCK}[C'.\text{SEQ}\%3].\text{SET}$。为证明这个蕴含，假设 C'.活动状态$_p = false$。

在 C.活动状态$_p = false$ 的情况下，根据归纳假设 IH:4 $p \notin \text{LOCK}[C.\text{SEQ}\%3].\text{SET}$ ，从 C 到 C' 的这一步是由某个 $q \neq p$ 执行的。唯一可能影响该条件的一步是 q 将 SEQ 的值更改；这可能发生在第 **24** 行，而且该行只会将 SEQ 加 **1**。我们注意到，归纳假设 IH:2 意味着 $\text{LOCK}[(C.\text{SEQ}+1)\%3].\text{STATE} = (\phi,\ \phi,\ \phi)$，而 q 将 SEQ 加 **1** 意味着 $\text{LOCK}[(C'.\text{SEQ})\%3].\text{STATE} = (\phi,\ \phi,\ \phi)$，这意味着 $p \notin \text{LOCK}[C'.\text{SEQ}\%3].\text{SET}$。

在 C.活动状态$_p = true$的情况下，只有 p 执行行 **20** 或 **27** ，才能将 Active$_p$变为 $false$。因此，$C.PC_p \in \{\textbf{20}, \textbf{27}\}$。根据归纳假设 IH:4，我们有 $p \notin \text{LOCK}[C.\text{SEQ}\%3].\text{SET}$，这蕴含 $p \notin \text{LOCK}[C'.\text{SEQ}\%3].\text{SET}$。

ii. $C'.\text{S}_p < C'.\text{SEQ} \Rightarrow p \notin \text{LOCK}[C'.\text{SEQ}\%3].\text{SET}$。
为证明这一蕴含，假设 $C'.\text{S}_p < C'.\text{SEQ}$。

在 $C.\text{S}_p < C.\text{SEQ}$的情况下，根据归纳假设 IH:4 $p \notin \text{LOCK}[C.\text{SEQ}\%3].\text{SET}$ ，从 C 到 C' 的一步是由某个 $q \neq p$执行的。此情况与上文情况 i 的论证类似。

根据归纳假设 IH:1，我们知道 $C.\text{S}_p \leq C.\text{SEQ}$。因此，另一种情况是 $C.\text{S}_p = C.\text{SEQ}$。检查算法可知 $C'.\text{SEQ} = C.\text{SEQ} + 1$，因为 S_p 从未递减。因此，这一步是进程 q 执行行 **24** （可能与 p相同）。根据归纳假设 IH:2，我们有 $C.\text{LOCK}[(\text{SEQ}+1)\%3].\text{STATE} = (\phi,\ \phi,\ \phi)$，这蕴含$p \notin \text{LOCK}[C'.\text{SEQ}\%3].\text{SET}$。

iii. $C'.PC_p \in \{\textbf{3}, \textbf{19}, \textbf{20}, \textbf{26}, \textbf{27}\} \Rightarrow p \notin \text{LOCK}[C'.\text{SEQ}\%3].\text{SET}$。
为证明这一蕴含，假设 $C'.PC_p \in \{\textbf{3}, \textbf{19}, \textbf{20}, \textbf{26}, \textbf{27}\}$。

在情况 $C.PC_p \in \{\textbf{3}, \textbf{19}, \textbf{20}, \textbf{26}, \textbf{27}\}$ 中，根据归纳假设：$4\ p \notin \text{LOCK}[C.\text{SEQ}\%3].\text{SET}$。如果 p在第**19** 行或**26** 行执行了一步，分别转到第**20** 行或第**27**行，那么这一步不会影响$\text{LOCK}[C.\text{SEQ}\%3].\text{SET}$。这意味着 $p \notin \text{LOCK}[C'.\text{SEQ}\%3].\text{SET}$。如果从 C 到C' 的这一步是由某个 $q \neq p$执行的，那么情况与上文情况 i 的论证类似。

假设 $C.PC_p \in \{\textbf{2}, \textbf{17}, \textbf{18}, \textbf{21}, \textbf{25}\}$，因此由于 p 执行了一步， $C'.PC_p \in \{\textbf{3}, \textbf{19}, \textbf{20}, \textbf{26}, \textbf{27}\}$。如果 $C.PC_p = \textbf{2}$，根据归纳假设 IH:10，活动状态$_p = false$，并且根据归纳假设 IH:4， $p \notin \text{LOCK}[C.\text{SEQ}\%3].\text{SET}$，这意味着 $p \notin \text{LOCK}[C'.\text{SEQ}\%3].\text{SET}$。如果 $C.PC_p = \textbf{17}$，则由该步可知 $C.\text{S}_p \neq C.\text{SEQ}$，并且根据归纳假设 IH:1 $C.\text{S}_p < C.\text{SEQ}$。根据归纳假设 IH:4，可知 $p \notin \text{LOCK}[C.\text{SEQ}\%3].\text{SET}$，这意味着 $p \notin \text{LOCK}[C'.\text{SEQ}\%3].\text{SET}$。如果 $C.PC_p = \textbf{18}$，则该步完成了 p 对$\text{LOCK}[C.\text{S}_p\%3]$.exit$_p$() 的执行，因此将 p 从 $\text{LOCK}[C.\text{S}_p\%3].\text{EXITSET}$ 中移除。根据归纳假设 IH:5 p ，其至多位于 $\text{LOCK}[C.\text{S}_p\%3].\text{TRYSET}$、$\text{LOCK}[C.\text{S}_p\%3].\text{CSSET}$ 或 $\text{LOCK}[C.\text{S}_p\%3].\text{EXITSET}$ 其中之一，这意味着 $p \notin \text{LOCK}[C'.\text{S}_p\%3].\text{SET}$。如果 $C.\text{S}_p = C.\text{SEQ}$，则有 $p \notin \text{LOCK}[C'.\text{SEQ}\%3].\text{SET}$；否则 $C.\text{S}_p < C.\text{SEQ}$ ，并且根据归纳假设 IH:4，$p \notin \text{LOCK}[C'.\text{SEQ}\%3].\text{SET}$。如果 $C.PC_p = \textbf{21}$，则要么 $C.$处于活动状态$_p = false$ ，要么 $C.\text{S}_p < C.\text{SEQ}$；无论哪种情况， $p \notin \text{LOCK}[C'.\text{SEQ}\%3].\text{SET}$，如上文所述。如果 $C.PC_p = \textbf{25}$，根据归纳假设 IH:11 $C.\text{S}_p < C.\text{SEQ}$，这意味着$p \notin \text{LOCK}[C'.\text{SEQ}\%3].\text{SET}$ ，如上文所述。

(b). $C'.\mathrm{S}_p < C'.\textsc{Seq} - 1 \Rightarrow p \notin \textsc{Lock}[C'.\mathrm{S}_p\%3].\textsc{Set}$。
为证明这一蕴含，假设 $C'.\mathrm{S}_p < C'.\textsc{Seq} - 1$。

如果 $C.\mathrm{S}_p < C.\textsc{Seq} - 1$，那么根据归纳假设 IH:4 $p \notin \textsc{Lock}[C.\mathrm{S}_p\%3].\textsc{Set}$，从 C 到 C' 的这一步是由某个 $q \neq p$完成的。q 执行的这一步不会影响 S_p，也不会将 p 放入与任何基础锁关联的集合中。因此可得 $p \notin \textsc{Lock}[C'.\mathrm{S}_p\%3].\textsc{Set}$。

由于 $C'.\mathrm{S}_p < C'.\textsc{Seq} - 1$，唯一可能的另一种情况是 $C.\mathrm{S}_p = C.\textsc{Seq} - 1$。在这种情况下，$\textsc{Seq}$因进程 q执行**第 24 行** 而递增。由于这一步会递增 $\textsc{Seq}$，因此可得 $C.\mathrm{S}_q = C.\textsc{Seq}$，进而得到 $p \neq q$。根据归纳假设 IH:4，由于 $\mathrm{S}_p < \textsc{Seq}$，$p \notin \textsc{Lock}[C.\textsc{Seq}\%3].\textsc{Set}$，因而 $p \notin \textsc{Lock}[(C.\textsc{Seq} - 1)\%3].\textsc{Set}$。根据归纳假设 IH:2，$p \notin \textsc{Lock}[(C.\textsc{Seq} + 1)\%3].\textsc{Set}$，这蕴含 $p \notin \textsc{Lock}[C'.\textsc{Seq}\%3].\textsc{Set}$。由于我们有 $C.PC_q = \mathbf{24}$，根据归纳假设 IH:2，$p \notin \textsc{Lock}[(C.\mathrm{S}_q - 1)\%3].\textsc{Set}$，这意味着 $p \notin \textsc{Lock}[(C'.\textsc{Seq} + 1)\%3].\textsc{Set}$（因为 q 执行的这一步将$\textsc{Seq}$ 递增 **1**，且 $C.\mathrm{S}_q = C.\textsc{Seq}$）。由此可知，$p$ 不在任何锁集合中，蕴含关系也就成立。

5. 证明 C' 中条件 5 成立
p 只会因在基础锁 ℓ 上调用/执行 $\ell.\mathbf{try}_p()$ 和 $\ell.\mathbf{exit}_p()$ 而被移至不同的集合。

当 p 执行行 **3**，并在行 **4** 调用 $\textsc{Lock}[C'.\mathrm{S}_p\%3].\mathbf{try}_p()$ 时，根据归纳假设：4 $p \notin \textsc{Lock}[C.\textsc{Seq}\%3].\textsc{Set}$。类似地，当它执行行 **9** 时，根据归纳假设：11 $C.\mathrm{S}_p = C.\textsc{Seq}-1$，因此根据归纳假设：4 $p \notin \textsc{Lock}[C.\textsc{Seq}\%3].\textsc{Set}$。由于 $C'.\mathrm{S}_p = C.\textsc{Seq}$ 是由这一步导致的，因此该条件在 C' 中成立。

当 p 在行 **4** 或 **10** 成功完成 $\textsc{Lock}[C.\mathrm{S}_p\%3].\mathbf{try}_p()$ 时，p 被移至 $\textsc{Lock}[C.\mathrm{S}_p\%3].\textsc{CSSet}$，并从 $\textsc{Lock}[C.\mathrm{S}_p\%3].\textsc{TrySet}$ 中移除。因此，该条件在 C' 中成立。

当 p 在第 **18** 行调用 $\textsc{Lock}[C.\mathrm{S}_p\%3].\text{退出}_p()$ 时，p 会从 $\textsc{Lock}[C.\mathrm{S}_p\%3].\textsc{CSSet}$ 移至 $\textsc{Lock}[C.\mathrm{S}_p\%3].\textsc{ExitSet}$。同样，当 p 成功完成 $\textsc{Lock}[C.\mathrm{S}_p\%3].\text{退出}_p()$ 时，p 会从 $\textsc{Lock}[C.\mathrm{S}_p\%3].\textsc{Set}$ 中移除。因此，C' 中的条件成立。

当进程 q 在第 **22** 行执行 $\textsc{Lock}[C.\mathrm{S}_q\%3].\text{重置}_q()$ 时，它只会清空这三个集合。因此，p 不属于 C' 中的任何一个集合，从而满足该条件。

6. 证明条件 6 在 C' 中成立
当进程 p 在 C 中从第 **3** 行迈出一步，前往 C' 中的第 **4** 行时，根据归纳假设：4 $p \notin \textsc{Lock}[C.\textsc{Seq}\%3].\textsc{Set}$。随后该蕴含成立，因为在第 **4** 行调用 $\textsc{Lock}[\mathrm{S}_p\%3].\text{尝试}_p()$ 会将 p 放入 $\textsc{Lock}[\mathrm{S}_p\%3].\textsc{TrySet}$。同样，当进程 p 从第 **9** 行迈出一步，前往第 **10** 行时，根据归纳假设**11**，我们有 $C.\mathrm{S}_p = C.\textsc{Seq} - 1 < C.\textsc{Seq}$，进而根据归纳假设：4 $p \notin \textsc{Lock}[C.\textsc{Seq}\%3].\textsc{Set}$。该蕴含再次成立，因为在第**10** 行调用 $\textsc{Lock}[\mathrm{S}_p\%3].\text{尝试}_p()$ 会将 p 放入 $\textsc{Lock}[\mathrm{S}_p\%3].\textsc{TrySet}$。

如果进程 $q \neq p$ 在第**22** 行执行 $\textsc{Lock}[(\mathrm{S}_q - 1)\%3].\text{重置}_q()$，从而将$\textsc{Lock}[(\mathrm{S}_q - 1)\%3].\textsc{State}$ 设为 (ϕ, ϕ, ϕ)，我们首先注意到，根据归纳假设：11 $\mathrm{S}_q \in \{\textsc{Seq} - 1, \textsc{Seq}\}$。如果 $C.PC_p = \mathbf{4}$且 $C.\mathrm{S}_p = C.\textsc{Seq} - 1$，那么根据归纳假设：13 $C.\mathrm{S}_q = C.\textsc{Seq} - 1$，这意味着 q 的这一步不会影响 $\textsc{Lock}[\mathrm{S}_p\%3].\textsc{State}$。如果 $C.\mathrm{S}_p = C.\textsc{Seq}$，那么由于 $\mathrm{S}_q \in \{\textsc{Seq} - 1, \textsc{Seq}\}$，$q$ 的这一步只会影响$\textsc{Lock}[(C.\textsc{Seq} - 1)\%3].\textsc{State}$ 或 $\textsc{Lock}[(C.\textsc{Seq} - 2)\%3].\textsc{State} = \textsc{Lock}[(C.\textsc{Seq} + 1)\%3].\textsc{State}$，而不会影响 $\textsc{Lock}[C.\textsc{Seq}\%3].\textsc{State}$。

7. 证明 C' 中条件 7 成立
我们将蕴含关系分为以下子部分，并论证每个部分的正确性。

(a). $(C'.PC_p = \mathbf{5} \wedge C'.\textsc{Stop}[\mathrm{S}_p\%3] = false) \Rightarrow p \in \textsc{Lock}[C'.\mathrm{S}_p\%3].\textsc{CSSet}$。这里我们首先论证，当 p 执行一步从而使得 $C'.PC_p = \mathbf{5}$时，该条件的正确性。存在两种可能：$C.PC_p = \mathbf{4}$ 或 $C.PC_p = \mathbf{6}$。在 $C.PC_p = \mathbf{4}$ 的情况下，根据蕴含关系的假设，我们知道 $C'.\textsc{Stop}[\mathrm{S}_p\%3] = false$。由于这一步不会改变

C.Stop[S_p%3], , 因此 p 已成功完成 Lock[S_p%3].try$_p$() 的执行。由此， p 从 Lock[$C.S_p$%3].TrySet 中移除并放入 Lock[$C.S_p$%3].CSSet，从而 $p \in$ Lock[$C'.S_p$%3].CSSet。在 $C.PC_p = \mathbf{6}$ 的情况下，根据归纳假设: 7 $p \in$ Lock[$C.S_p$%3].CSSet，因此 $p \in$ Lock[$C'.S_p$%3].CSSet。无论哪种情况， $p \in$ Lock[$C'.S_p$%3].CSSet。

如果从 C 到 C' 的这一步是由进程 $q \neq p$执行的，那么只有第 **22** 行的这一步才可能影响Lock[$C.S_p$%3].State。这种情况的论证与上文条件 6 的论证相同，由此可知 Lock[$C.S_p$%3].State 不受 q 的这一步影响。

(b). $C'.PC_p \in \{$**6-8**，**11-15**$\} \Rightarrow p \in$ Lock[$C'.S_p$%3].CSSet。

这里需要证明，当 p 执行从第 **5** 行到第 **6** 行的一步，或从第 **10** 行到第 **11**行的一步时，蕴含成立。考虑 $C.PC_p = \mathbf{5}$。由于 $C'.PC_p = \mathbf{6}$，我们注意到 C.Stop[S_p%3] $= false$，因此根据归纳假设: 7 $p \in$ Lock[$C.S_p$%3].CSSet ，进而$p \in$ Lock[$C'.S_p$%3].CSSet。如果 $C.PC_p = \mathbf{10}$, p 已完成 Lock[S_p%3].尝试$_p$() 并转到第 **11** 行，这会从 Lock[$C.S_p$%3].TrySet 中移除 p , 并将其放入 Lock[$C.S_p$%3].CSSet 中。无论哪种情况， $p \in$ Lock[$C'.S_p$%3].CSSet。

如果从 C 到 C' 的步是由进程 $q \neq p$执行的，那么只有第 **22**行的步才可能影响Lock[$C.S_p$%3].State。我们注意到 $C.PC_p \in \{$**6-8**，**11-15**$\}$。如果 $C.PC_p \in \{$**6-7**$\}$ 且 $C.S_p = C$.Seq-1, 我们使用与上文条件 6 相同的论证，并注意到 Lock[$C.S_p$%3].State不受 q 的步影响。同样，对于 $C.PC_p \in \{\mathbf{8}, \mathbf{11\text{-}15}\}$，归纳假设**13**意味着 $S_q =$ Seq $- 1$，我们再次注意到 Lock[$C.S_p$%3].State 不会受到 q 的步影响。否则，根据条件 11 $C.S_p = C$.Seq ，这种情况也可用与上文条件 6 相同的论证来说明。无论哪种情况， Lock[$C.S_p$%3].State 都不受该步影响。

(c). ($C'.PC_p \in \{\mathbf{16}, \mathbf{17}\} \wedge C'.S_p = C'$.Seq) $\Rightarrow p \in$ Lock[$C'.S_p$%3].CSSet。

这里需要证明，当 p 从 $C.PC_p \in \{\mathbf{7}, \mathbf{12}, \mathbf{15}\}$ 执行一步到达 $C'.PC_p = \mathbf{16}$ 时，该蕴含成立。当 $C.PC_p = \mathbf{7}$ 时，很容易看出该蕴含成立，因为这一步只修改了 CSOwner。类似地，对于从 $C.PC_p = \mathbf{12}$ 执行的一步也可得出相同结论。对于从 $C.PC_p = \mathbf{15}$ 执行的一步，根据归纳假设 **7**，该蕴含显然成立。

如果从 C 到 C' 的一步是由进程 $q \neq p$ 执行的，那么只有第 **22** 行的步骤可能影响 Lock[$C.S_p$%3].State。根据该蕴含的假设，我们知道 $C.S_p = C$.Seq。根据归纳假设:11 $C.S_q \in \{C$.Seq $- 1, C$.Seq$\}$，在这两种情况下，q 只影响 Lock[(C.Seq $-$ 1)%3].State 或 Lock[(C.Seq $-$ 2)%3].State $=$ Lock[(C.Seq $+$ 1)%3].State，而不会影响 Lock[C.Seq%3].State。因此，我们得到 $p \in$ Lock[$C'.S_p$%3].CSSet。

8. 证明 C' 中条件8成立

当进程 p 从 C 中的第 **17** 行执行一步，并满足 $C.S_p = C$.Seq ，转到 C' 中的第 **17** 行时，根据归纳假设: 7 $p \in$ Lock[C.Seq%3].CSSet。因此，该蕴含成立，因为在第**18** 行调用 Lock[S_p%3].退出$_p$() 会将 p 放入 Lock[S_p%3].ExitSet。

如果进程 $q \neq p$ 在第 **22** 行执行一步，执行 Lock[($S_q - 1$)%3].重置$_q$()，从而将Lock[($S_q - 1$)%3].State 设为 (ϕ, ϕ, ϕ)，我们首先注意到，根据归纳假设: 11 $S_q \in \{$Seq $- 1,$ Seq$\}$。根据归纳假设**11**，由于 $C.PC_p = \mathbf{18}$, $C.S_p \in \{C$.Seq $- 1, C$.Seq$\}$。如果 $C.S_p = C$.Seq $- 1$，那么根据归纳假设: 13 $C.S_q = C$.Seq $- 1$，这意味着 q 的这一步不会影响 Lock[S_p%3].State。如果 $C.S_p = C$.Seq，那么由于 $S_q \in \{$Seq $- 1,$ Seq$\}$, q 的这一步只会影响 Lock[(C.Seq $-$ 1)%3].State 或 Lock[(C.Seq $-$ 2)%3].State $=$Lock[(C.Seq $+$ 1)%3].State，而不会影响 Lock[C.Seq%3].State。

9. 证明 C' 中的条件 9 成立

我们分别证明条件 9 的每个合取项，如下所示 .

(a). (C'.活动状态$_p = false \vee C'.PC_p \in \{$**2-13**，**20**，**27**$\}) \Rightarrow C'$.CSOwner $\neq p$我们将蕴含拆分为以下子部分，并论证每个部分的正确性。

i. C'.活动状态$_p = false \Rightarrow C'$.CSOwner $\neq p$。假设 C.活动状态$_p = true$。这里我们需要论证，当 p 执行一步以将 C. 活动状态 p

设置为 $false$ 时，它要么在第 **20** 行，要么在第 **27**行执行此操作。因此， $C.PC_p \in \{\mathbf{20}, \mathbf{27}\}$。由归纳假设 9 可知 $C.\text{CSOWNER} \neq p$，且由于该步不影响 CSOWNER，故 $C'.\text{CSOWNER} \neq p$。

假设 C.活动状态$_p = false$，我们分两种情况：(a) p 执行了一步，且未修改活动状态$_p$；或者 (b) 从 C 到 C' 这一步由进程 $q \neq p$执行。第一种情况下，根据归纳假设 IH:10 第一部分的逆否命题，我们有 $C.PC_p \notin \{\mathbf{7}, \mathbf{12}, \mathbf{14}, \mathbf{19}\}$。因此， p 不会因这一步而修改 CSOWNER，于是蕴含关系成立。第二种情况下，如果 q 确实修改了 CSOWNER，则只会在第 **7**、**12**、**14** 或 **19** 行进行修改，而这些行都不会将 p 的名称写入 $C.\text{CSOWNER}$。因此，两种情况下，该蕴含关系在 C' 中都成立。

ii. $C'.PC_p \in \{\mathbf{2\text{-}13}、\mathbf{20}、\mathbf{27}\} \Rightarrow C'.\text{CSOWNER} \neq p$。假设 $C.PC_p = \mathbf{1}$，使得 p 执行一步并转到第 **2** 行。由此可知， C.状态$_p$ 必须为正常，否则 p 会调用 recover$_p$()，而不是 try$_p$()。根据归纳假设 IH:10 C.，$_p = false$处于活动状态，因此根据归纳假设 IH:9 $C.\text{CSOWNER} \neq p$，可得 $C'.\text{CSOWNER} \neq p$。

假设 $C.PC_p \in \{\mathbf{2\text{-}12}\}$。在这种情况下，从 C 到 C' 的步由 p 执行，且不会影响 CSOWNER。这包括第**7** 行和第**12**行的 CAS 操作；这些操作必定失败了，否则 p会进入临界区，而不是转到下一行。由归纳假设 9 可得 $C.\text{CSOWNER} \neq p$，因此$C'.\text{CSOWNER} \neq p$。

假设 $C.PC_p = \mathbf{19}$。在这种情况下， p执行第**19** 行时，会将 CSOWNER 设为 $\perp$，因此蕴含在 C'中成立。

假设 $C.PC_p = \mathbf{26}$。在这种情况下， p 转到第**27** 行，是因为第**26**行的**if** 条件不成立。因此，蕴含在 C'中成立。

假设 $C.PC_p = C'.PC_p$。在这种情况下，从 C到 C'的步由某个进程 q执行。如果q 修改了 CSOWNER，则只会在第**7**行、第**12**行、第**14**行或第**19**行进行修改；这些行都不会将p的名字写入 $C.\text{CSOWNER}$。因此，蕴含在 C'中成立。

(b). $C'.PC_p = \mathbf{14} \Rightarrow C'.\text{CSOWNER} = \perp$假设 $C.PC_p = \mathbf{13}$。由于 $C'.PC_p = \mathbf{14}$， p 离开了第 **13** 行的**等待直到**循环，转而执行第 **14** 行。只有在 $\text{CSOWNER} = \perp$ 时才可能发生这种情况。因此我们得到该蕴含。

假设 $C.PC_p = \mathbf{14}$。在这种情况下，从 C 到 C' 的一步由某个进程 q 执行。注意，根据归纳假设 IH:9 $C.\text{CSOWNER} = \perp$。我们要论证， q 无法在第 **7**、**12**、**14** 或 **19** 行执行一步来修改 CSOWNER。首先，$C.PC_q \neq \mathbf{19}$，否则根据归纳假设 IH:9 $\text{CSOWNER} = q \neq \perp$，会产生矛盾。其次， $C.PC_q \in \{\mathbf{7}, \mathbf{12}, \mathbf{14}\}$ 和 $C.\text{S}_q = \text{SEQ}$ 不可能同时成立。这是因为根据归纳假设 **7** 和归纳假设 **11**，我们已经有了 $p \in \text{LOCK}[C.\text{SEQ} \ \%3]$，因此 $\text{LOCK}[C.\text{SEQ}\%3]$ 的互斥条件排除了这种情况。最后，如果 $C.\text{S}_q = C.\text{SEQ} - 1$，根据 IH:14$PC_q \in \{\mathbf{1\text{-}4}, \mathbf{9}, \mathbf{20}, \mathbf{21\text{-}28}\}$，并且 q 在这些行中的任意一步都不会修改 CSOWNER。

(c). ($C'.PC_p \in \{\mathbf{15\text{-}19}\} \vee C'$.状态$_p$ = 从临界区恢复) $\Rightarrow C'.\text{CSOWNER} = p$我们将蕴含拆分为以下子部分，并论证每个部分的正确性。

i. $C'.PC_p \in \{\mathbf{15\text{-}19}\} \Rightarrow C'.\text{CSOWNER} = p$。为论证正确性，我们首先假设 $C'.PC_p \in \{\mathbf{15\text{-}19}\}$。 p 可能由于自身在程序不同位置执行的一步而到达给定行，或者 p 可能没有改变 PC_p，但这一步是由进程 $q \neq p$ 执行的。接下来，我们将逐一论证这些情况，并证明该条件的正确性。

假设 $C.PC_p = \mathbf{7}$。由于 $C'.PC_p \in \{\mathbf{15\text{-}19}\}$，我们特别注意到，第 **7** 行会因 CAS 成功而将 p 放入 CS （第 **16** 行）。因此，CAS 会将 p 的名称写入 CSOWNER，由此可得 $C'.\text{CSOWNER} = p$。

假设 $C.PC_p = \mathbf{12}$。由于 $C'.PC_p \in \{\mathbf{15\text{-}19}\}$，我们注意到，具体而言，第 **12** 行通过一次成功的比较并交换将 p 放入临界区（位于第 **16** 行）。因此，比较并交换写入了 p 的名称

进入 CSOWNER，因此我们有 $C'.\text{CSOWNER} = p$。

假设 $C.PC_p = \mathbf{14}$。由于 $C'.PC_p \in \{\mathbf{15\text{-}19}\}$，我们注意到，具体而言，第 **14** 行将 p 写入 CSOWNER 并递增 PC_p。因此，我们有 $C'.PC_p = \mathbf{15}$ 和 $C'.\text{CSOWNER} = p$。

假设 $C.PC_p = \mathbf{26}$。由于 $C'.PC_p \in \{\mathbf{15\text{-}19}\}$，我们注意到，具体而言，第 **26** 行将$p$ 放回 CS（第 **16** 行），因为 p 找到了 $C.\text{CSOWNER} = p$，且第 **26** 行的 **if** 条件成立。因此我们有 $C.\text{CSOWNER} = p$，进而有 $C'.\text{CSOWNER} = p$。

假设 $C.PC_p = C'.PC_p$。根据归纳假设 9，我们注意到 $C.\text{CSOWNER} = p$。因此，从 C 到 C' 的这一步是由进程 $q \neq p$ 执行的。我们在下面论证，q 的这一步不会修改$C.\text{CSOWNER}$；根据归纳假设：9 $C.\text{CSOWNER} = p$，这将意味着 $C'.\text{CSOWNER} = p$。首先，如果q 在第 **7** 行或第 **12** 行执行一步，那么 q 的比较并交换会失败，因为 $C.\text{CSOWNER} = p \neq \perp$。其次，$C.PC_q \neq \mathbf{14}$，因为我们知道 $C.\text{CSOWNER} = p$，并且根据归纳假设:9 $C.\text{CSOWNER} = \perp \neq p$，$C.PC_q$ 在 **14** 行，这就产生了矛盾。最后，$C.PC_q \neq \mathbf{19}$，同样因为我们知道$C.\text{CSOWNER} = p$，并且根据归纳假设:
9 $C.\text{CSOWNER} = q \neq p$，$C.PC_q$ 在 **19** 行，这就产生了矛盾。因此，根据以上论述，我们得出结论：q 的这一步没有修改 $C.\text{CSOWNER}$。所以我们有$C'.\text{CSOWNER} = p$。

ii. $C'.\text{状态}_p = \text{从临界区恢复} \Rightarrow C'.\text{CSOWNER} = p$。

假设 $C.\text{状态}_p \neq \text{从临界区恢复}$。由此可知，该步是一个崩溃步骤，并且 p在配置 C（即 $C.PC_p = \mathbf{16}$）中处于临界区，因此该步骤使 $C.\text{状态}_p$ 变为从临界区恢复。由于该崩溃步骤不会影响 $C.\text{CSOWNER}$，且因为 $C.PC_p = \mathbf{16}$，根据归纳假设：9 $C.\text{CSOWNER} = p$，这意味着 $C'.\text{CSOWNER} = p$。

假设 $C.\text{状态}_p = \text{从临界区恢复}$。注意，根据归纳假设：9 $C.\text{CSOWNER} = p$。由此可知，从 C 到 C' 的步骤要么是崩溃步骤，要么是进程 $q \neq p$执行的正常步骤。如果是崩溃步骤，那么条件在 C' 中成立，因为它在 C中成立。如果是进程 $q \neq p$执行的步骤，我们将在下文论证，q的步骤不会修改 $C.\text{CSOWNER}$；根据归纳假设 9：$C.\text{CSOWNER} = p$，这将意味着 $C'.\text{CSOWNER} = p$。首先，如果 q 在第 **7** 行或第 **12**行执行步骤，那么 q的比较并交换将失败，因为 $C.\text{CSOWNER} = p \neq \perp$。其次，$C.PC_q \neq \mathbf{14}$，因为我们知道 $C.\text{CSOWNER} = p$，且根据归纳假设：9 $C.\text{CSOWNER} = \perp \neq p$，$C.PC_q$ 若为**14**，就会导致矛盾。最后，$C.PC_q \neq \mathbf{19}$，同样因为我们知道 $C.\text{CSOWNER} = p$，且根据归纳假设：9 $C.\text{CSOWNER} = q \neq p$，$C.PC_q$ 若为**19**，就会导致矛盾。因此，根据上述论证，我们得出结论：q的步骤没有修改 $C.\text{CSOWNER}$。于是有 $C'.\text{CSOWNER} = p$。

10. C'中条件 10 成立的证明

我们将分别证明条件 10 的每个合取项，如下所示。

(a). $C'.PC_p \in \{\mathbf{3\text{-}20}, \mathbf{22\text{-}25}\} \Rightarrow C'.\text{活动状态}_p = true$

为了论证正确性，我们首先假设 $C'.PC_p \in \{\mathbf{3\text{-}20}, \mathbf{22\text{-}25}\}$。$p$可能因自身在程序不同位置执行的步而到达给定行，或者 p 可能没有改变 PC_p，但该步是由进程 $q \neq p$执行的。下文将分别讨论这些情况，并证明该条件的正确性。

假设 $C.PC_p = \mathbf{2}$。通过这一步，p 将 Active_p 设为 $true$，由此得到 $C'.\text{Active}_p = true$，满足该蕴含。

假设 $C.PC_p = \mathbf{21}$。通过这一步，p只能转到第 **22** 行，因为 $C'.PC_p \in \{\mathbf{3\text{-}20}, \mathbf{22\text{-}25}\}$。由此可知，第 **21** 行的 **if** 条件已满足，因此 $C.\text{Active}_p = true$。由于这一步不会影响 Active_p，因此我们有 $C'.\text{Active}_p = true$。

假设 $C.PC_p = C'.PC_p$。根据归纳假设 IH:10，我们有 $C.\text{活动状态}_p = true$。因此，从 C到 C' 的步是进程 $q \neq p$执行的正常步骤。由于 q 不会在代码的任何位置更改活动状态$_p$，我们有 $C.\text{活动状态}_p = C'.\text{活动状态}_p$。因此，$C'.\text{活动状态}_p = true$。

(b). $((C'.PC_p \in \{\mathbf{1}, \mathbf{21\text{-}27}\} \wedge C'.\text{状态}_p \in \{\text{正常，从余数部分恢复}\}) \vee C'.PC_p \in \{\mathbf{2}, \mathbf{28}\})$
$\Rightarrow C'.\text{活动状态}_p = false$W

我们将蕴含拆分为以下子部分，并论证每一部分的正确性 art.

i. $C'.PC_p \in \{\mathbf{1}, \mathbf{21\text{-}27}\} \wedge C'.\text{状态}_p \in \{\text{正常，从余数部分恢复}\} \Rightarrow C'.\text{活动状态}_p = false$。为了论证正确性，我们首先假设 $C'.PC_p \in \{\mathbf{1}, \mathbf{21\text{-}27}\} \wedge C'.\text{状态}_p \in$正常，从余数部分恢复}。 p 可能由于程序中不同位置的某个步骤而到达给定状态，或者 p 可能没有改变 PC_p ，但该步骤是由进程 $q \neq p$执行的。接下来，我们将分别讨论这些情况，并证明该条件的正确性。

假设 $C.PC_p = \mathbf{20}$。在这种情况下， p会将活动状态$_p$设为 $false$，因此我们有 $C'.\text{活动状态}_p = false$

假设 $C.PC_p = \mathbf{28}$。根据归纳假设 IH:10 $C.\text{活动状态}_p = false$，并且由该步可得 $C'.\text{活动状态}_p = false$。

假设 $C.PC_p \in \{\mathbf{1}, \mathbf{21\text{-}27}\} \wedge C.\text{状态}_p \in \{\text{正常，从余数部分恢复}\}$。根据归纳假设 IH:10，我们有$C.\text{活动状态}_p = false$。由此可知，从 C 到 C' 的步要么是进程 p 执行的正常步骤，且不会将活动状态$_p$变为 $true$（这只会发生在第 **2** 行），要么是崩溃步骤（不会改变活动状态$_p$），要么是进程 $q \neq p$执行的正常步骤，同样不会改变活动状态$_p$。因此，我们有 $C'.\text{活动状态}_p = false$。

ii. $C'.PC_p \in \{\mathbf{2}, \mathbf{28}\} \Rightarrow C'.\text{活动状态}_p = false$。与上文类似，我们考虑以下不同情形。

假设 $C.PC_p = \mathbf{1}$ and $C.\text{状态}_p =$ 正常。在这种情况下，根据归纳假设 IH:10，我们有 $C.\text{活动状态}_p = false$，它在 C' 中保持不变。

假设 $C.PC_p = \mathbf{27}$。在这种情况下，该步将 $false$写入活动状态$_p$。因此，我们有$C'.\text{活动状态}_p = false$。

假设 $C.PC_p = C'.PC_p$。由此可知，从 C到 C'的步是某个进程 $q \neq p$ 执行的一步，它不会改变活动状态$_p$的值。由于根据归纳假设 IH:10， $C.\text{活动状态}_p = false$ ，因此我们有 $C'.\text{活动状态}_p = false$。

11. 证明 C' 中条件 11 成立

我们将在下文分别证明条件 11 的每个合取项。

(a). $C'.PC_p \in \{\mathbf{4\text{-}8}, \mathbf{18}, \mathbf{22\text{-}24}\} \Rightarrow C'.\text{S}_p \in \{C'.\text{SEQ} - 1, C'.\text{SEQ}\}$与上述论证类似，我们考虑以下不同情形。因此，假设$C'.PC_p \in \{\mathbf{4\text{-}8}, \mathbf{18}, \mathbf{22\text{-}24}\}$。

假设 $C.PC_p = \mathbf{3}$。根据步 S_p ， $C.\text{SEQ}$ 取该值。因此我们有 $C'.\text{S}_p = C.\text{SEQ} = C'.\text{SEQ}$。

假设 $C.PC_p = \mathbf{17}$。我们注意到，对于上文所述的 $C'.PC_p$ 的各种可能情况，只有 $C'.PC_p = \mathbf{18}$ 才是真正可能的情况。因此，当 p 执行 C 中的行 **17** 时，**if** 条件成立。因此， $C.\text{S}_p = C.\text{SEQ}$ ，进而 $C'.\text{S}_p = C'.\text{SEQ}$。

假设 $C.PC_p = \mathbf{21}$。我们注意到，对于上文所述的 $C'.PC_p$ 的各种可能情况，只有 $C'.PC_p = \mathbf{21}$ 才是真正可能的情况。因此，当 p 执行 C 中的行 **21** 时，**if** 条件成立。因此， $C.\text{S}_p = C.\text{SEQ}$ ，进而 $C'.\text{S}_p = C'.\text{SEQ}$。

假设 $C.PC_p \in \{\mathbf{4\text{-}8}, \mathbf{18}, \mathbf{22\text{-}24}\}$。由于 $C'.PC_p \in \{\mathbf{4\text{-}8}, \mathbf{18}, \mathbf{22\text{-}24}\}$，如果从 C 开始的步

到 C' 的步是由 p 执行的，那么 p 的这一步不会改变 $\textsc{Seq}$ 的值，也不会改变 S_p 的值。因此，$C'.\mathrm{S}_p \in \{C'.\textsc{Seq}-1, C'.\textsc{Seq}\}$。否则，从 C 到 C' 的步是由进程 $q \neq p$ 执行的，且 $C.PC_p = C'.PC_p$。如果 q 没有因这一步而改变 $\textsc{Seq}$，那么该条件仍然成立。否则，q 执行了行 **24**，以递增 $\textsc{Seq}$。因此，$C.\mathrm{S}_q = C.\textsc{Seq}$。根据归纳假设**11**，我们有 $C.\mathrm{S}_p \in \{C.\textsc{Seq}-1, C.\textsc{Seq}\}$。如果 $C.\mathrm{S}_p = C.\textsc{Seq}-1$，根据归纳假设 13 $C.\mathrm{S}_q = C.\textsc{Seq}-1$，这就导致矛盾。因此，$C.\mathrm{S}_p = C.\textsc{Seq}$。由此可知，在 q 递增 $\textsc{Seq}$ 后，$C'.\mathrm{S}_p = C'.\textsc{Seq}-1$。因此，该条件成立。

(b). $C'.PC_p \in \{\mathbf{9}, \mathbf{25}\} \Rightarrow C'.\mathrm{S}_p = C'.\textsc{Seq}-1$与上述论证类似，我们考虑以下不同情形。因此，假设 $C'.PC_p \in \{\mathbf{9}, \mathbf{25}\}$。

假设 $C.PC_p = \mathbf{8}$。根据归纳假设：11 $C.\mathrm{S}_p \in \{C.\textsc{Seq}-1, C.\textsc{Seq}\}$。由于满足了使 p 移至第**9**行的**条件**，$C.\mathrm{S}_p = C.\textsc{Seq}-1$。这意味着 $C'.\mathrm{S}_p = C'.\textsc{Seq}-1$。

假设 $C.PC_p = \mathbf{5}$。由于 $C'.PC_p = \mathbf{9}$，$\textsc{Stop}[C.\mathrm{S}_p\%3] = true$。根据归纳假设：3 $\textsc{Stop}[C.\textsc{Seq}\%3] = false$。因此，根据归纳假设：11 $C.\mathrm{S}_p = C.\textsc{Seq}-1$，这意味着 $C'.\mathrm{S}_p = C'.\textsc{Seq}-1$。

假设 $C.PC_p = \mathbf{24}$。如果 $C.\mathrm{S}_p = C.\textsc{Seq}-1$，则该步不会改变任何内容，因此$C'.\mathrm{S}_p = C'.\textsc{Seq}-1$。否则，$C.\mathrm{S}_p = C.\textsc{Seq}$，且该步会递增 $\textsc{Seq}$。由此可知，在这两种情况下都有$C'.\mathrm{S}_p = C'.\textsc{Seq}-1$。

假设 $C.PC_p = C'.PC_p$。根据归纳假设**11**可知，$C.\mathrm{S}_p = C.\textsc{Seq}-1$，并且某个进程 q 从 C 执行了一步到达 C'。然而，注意根据归纳假设:13 $C.PC_q \in \{\mathbf{22}\text{-}\mathbf{25}\}$ 意味着$C.\mathrm{S}_q = C.\textsc{Seq}-1$。因此，由于这一步，$q$ 不会改变 $\textsc{Seq}$ 的值。

(c). $C'.PC_p \in [\mathbf{10}, \mathbf{15}] \Rightarrow C'.\mathrm{S}_p = C'.\textsc{Seq}$与上面的论证类似，我们考虑以下不同的情形。因此，假设$C'.PC_p \in [\mathbf{10}, \mathbf{15}]$。

假设 $C.PC_p = \mathbf{9}$。由归纳假设**11**可知 $C.\mathrm{S}_p = C.\textsc{Seq}-1$。由于这一步使 S_p递增，因此我们有 $C'.\mathrm{S}_p = C.\textsc{Seq}$。

假设 $C.PC_p = \mathbf{8}$。我们注意到 $C'.PC_p = \mathbf{13}$，而这只会在 $C.\mathrm{S}_p = C.\textsc{Seq}$的情况下发生；这可由归纳假设**11**推出。

假设 $C.PC_p = C'.PC_p$。由归纳假设**11**可知 $C.\mathrm{S}_p = C.\textsc{Seq}$，并且某个进程 q从 C到 C'执行了一步。然而，请注意，由归纳假设 IH:13 $C.PC_q \in \{\mathbf{22}\text{-}\mathbf{25}\}$可推出 $C.\mathrm{S}_q = C.\textsc{Seq}-1$。因此，由于这一步，$q$ 不会改变 $\textsc{Seq}$ 的值。

(d). $(C'.PC_p = \mathbf{26} \wedge C'.\textsc{CSOwner} = p) \Rightarrow C'.\mathrm{S}_p < C'.\textsc{Seq}$与上述论证类似，我们考虑以下不同情形。因此，假设$C'.PC_p = \mathbf{26} \wedge C'.\textsc{CSOwner} = p$。

假设 $C.PC_p = \mathbf{25}$。根据归纳假设：11 $C.\mathrm{S}_p = C.\textsc{Seq}-1 < C.\textsc{Seq}$。由于该步不会修改$\textsc{Seq}$，因此我们有 $C'.\mathrm{S}_p < C'.\textsc{Seq}$。

假设 $C.PC_p = \mathbf{21}$。由此可知，**Line if 21** 的条件不成立。由于 $C'.\textsc{CSOwner} = p$，该步不可能修改 $\textsc{CSOwner}$，因此我们有 $C.\textsc{CSOwner} = p$。根据归纳假设：9 $C.$活动状态$_p = true$。因此，$C.\mathrm{S}_p \neq C.\textsc{Seq}$，并且根据归纳假设：1 $C.\mathrm{S}_p < C.\textsc{Seq}$。由此可知，$C'.\mathrm{S}_p < C'.\textsc{Seq}$。

假设 $C.PC_p = C'.PC_p$。我们首先观察到 $C.\textsc{CSOwner} = p$，因为我们有 $C'.\textsc{CSOwner} = p$，并且没有进程 $q \neq p$ 能在从 C 到 C' 的步中将 $\textsc{CSOwner}$ 设置为 p，因为它在第 **7**、**12** 或 **14** 行只能写入自己的名称，或者在第 **19** 行写入值 $\perp$。根据归纳假设**11**可知 $C.\mathrm{S}_p < C.\textsc{Seq}$，并且某个进程 q 从 C 到 C'执行了一步。然而请注意，如果 q 执行了一步，它只能将 $\textsc{Seq}$ 增加 **1**。这意味着 $C'.\mathrm{S}_p < C'.\textsc{Seq}$。

12. 证明条件 12 在 C'($C'.PC_p \in \{$**4-6**$\} \wedge C'.\mathrm{S}_p = C'.\textsc{Seq} - 1$)$\Rightarrow$ ($\textsc{Stop}[C'.\mathrm{S}_p$%3$] = true \vee$ ($\exists q,\ \ C'.PC_q =$ **25** $\wedge C'.\mathrm{S}_q = C'.\mathrm{S}_p$)) 中成立

假设($C'.PC_p \in \{$**4-6**$\} \wedge C'.\mathrm{S}_p = C'.\textsc{Seq} - 1$)。我们按以下情况分析该条件。

(i). $C.PC_p =$ **3**由于行 **3** 只设置 $\mathrm{S}_p = \textsc{Seq}$ ，并且不修改 Seq，因此在假设($C'.PC_p \in \{$**4-6**$\} \wedge C'.\mathrm{S}_p = C'.\textsc{Seq} - 1$)成立的情况下，还会有一步通过递减来修改Seq 。由此可知，这种情况不会出现。(ii). $C.PC_p \in \{$**4-6**$\} \wedge C.\mathrm{S}_p \neq C.\textsc{Seq} - 1$

根据我们关于 C' 的假设以及归纳假设 IH:11 $C.\mathrm{S}_p = C.\textsc{Seq}$，可知某个进程 $q \neq p$在从 C 到 C'的这一步中，于行 **24** 递增了 Seq 。由于 q 在行 **24**递增了 Seq ，根据归纳假设**11**，我们有 $C.\mathrm{S}_q = C.\textsc{Seq} = C.\mathrm{S}_p$。因此，我们有 $\exists q,\ C'.PC_q =$ **25** $\wedge C'.\mathrm{S}_q = C'.\mathrm{S}_p$。

(iii). ($C.PC_p \in \{$**4-6**$\} \wedge C.\mathrm{S}_p = C.\textsc{Seq} - 1$) 根据归纳假设 IH:12，我们有 $\textsc{Stop}[C.\mathrm{S}_p$%3$] = true \vee$ ($\exists q$, $C.PC_q =$ **25** $\wedge C.\mathrm{S}_q = C.\mathrm{S}_p$)。如果 q 执行一步，条件仍然成立，因为 q 只会将 $true$ 写入 $\textsc{Stop}[C.\mathrm{S}_q$ %3] 和$C.\mathrm{S}_q = C.\mathrm{S}_p = C'.\mathrm{S}_p$。如果 p 从 C执行一步，它不会修改 S_p 或 Seq ，因此条件仍然成立。如果某个进程 $r \neq q$（其中 q 如上所述）执行一步，那么 r 不会修改 Seq、 S_p 或 S_q。由于我们有 ($C.PC_p \in \{$**4-6**$\} \wedge C.\mathrm{S}_p = C.\textsc{Seq} - 1$)，根据归纳假设**13**，这样的 r 满足 $C.PC_r =$ **25** 时具有 $C.\mathrm{S}_r = C.\textsc{Seq} - 1$。因此， r 执行的一步只会将$\textsc{Stop}[C.\mathrm{S}_p$%3]设为满足该条件的 $true$ 。

13. 证明 C' 中的条件 13 成立

We 将该蕴含分为以下子部分，并论证每个部分的正确性。

(a). $(C'.PC_p \in \{$**4-7**, **18**, **22-25**$\} \wedge C'.\mathrm{S}_p = C'.\textsc{Seq} - 1)$
$\Rightarrow \forall q, (\neg(C'.PC_q \in \{$**1**, **21**$\} \wedge C'.\mathrm{Active}_q = true \wedge C'.\mathrm{S}_q = C'.\textsc{Seq})$
$\wedge\ (C'.PC_q \in \{$**22-25**$\} \Rightarrow C'.\mathrm{S}_q = C'.\textsc{Seq} - 1))$

假设 ($C'.PC_p \in \{$**4-7**、**18**、**22-25**$\} \wedge C'.\mathrm{S}_p = C'.\textsc{Seq} - 1$)。我们按以下情况分析该条件。

(i). $C.PC_p =$ **3**由于行 **3** 仅设置 $\mathrm{S}_p = \textsc{Seq}$ ，而不修改 Seq，因此在假设 ($C'.PC_p \in \{$**4-7**、**18**、**22-25**$\} \wedge C'.\mathrm{S}_p =$$C'.\textsc{Seq} - 1$) 的情况下，我们将有一个通过递减来修改 Seq 的步。我们得出结论，该情况不会出现。对于考虑 $C.PC_p =$ **17** 或 $C.PC_p =$ **21** 的情况，同样的论证也成立，因此不再论证这些情况。

(ii). $C.PC_p \in \{$**4-7, 18, 22-24**$\} \wedge C.\mathrm{S}_p \neq C.\textsc{Seq} - 1$我们首先注意到，根据归纳假设**11**，当 $PC_p =$ **25**, $\mathrm{S}_p = \textsc{Seq} - 1$时。因此，情况 $C.PC_p =$**25** $\wedge C.\mathrm{S}_p \neq C.\textsc{Seq} - 1$ 不可能发生。接下来我们考虑这里剩余的子情况。根据我们关于 C' 的假设以及归纳假设11 $C.\mathrm{S}_p = C.\textsc{Seq}$，可知某个进程 q（可能与 p相同）在从 C 到 C' 的步中于第24行递增了 Seq 。根据归纳假设 IH:1，我们知道对于每个进程 r, $C.\mathrm{S}_r \leq C.\textsc{Seq}$。因此，我们有 $\forall r, \neg$（$C.PC_r \in \{$**1**, **21**$\} \wedge C.$活动状态$_r =$ $true \wedge C.\mathrm{S}_r = C.\textsc{Seq} + 1$），这显然成立。由此可知 $\forall r, \neg$（$C'.PC_r \in \{$**1**, **21**$\} \wedge C'.$活动状态$_r =$ $true \wedge C'.\mathrm{S}_r = C'.\textsc{Seq}$），因为从 C 到 C' 的步只影响了 Seq 的值。我们观察到，对于任意 r，如果 $C.PC_r \in \{$**22-24**$\}$，那么 $C.\mathrm{S}_r = C.\textsc{Seq}$。否则，根据归纳假设**11**$C.\mathrm{S}_r = C.\textsc{Seq} - 1$，但归纳假设**13**蕴含，对于改变了 Seq 的进程 q， $C.\mathrm{S}_q = C.\textsc{Seq} - 1$，这不可能成立。根据归纳假设**11**，我们知道情况 $C.PC_p =$ **25** $\wedge C.\mathrm{S}_p = C.\textsc{Seq}$ 不可能发生。因此，如果 $C'.PC_r \in \{$**22-24**$\}$，则有 $C'.\mathrm{S}_r = C'.\textsc{Seq} - 1$。而对于递增了 Seq 的进程 q，我们有 $C'.PC_q =$ **25** 和 $C'.\mathrm{S}_q = C'.\textsc{Seq} - 1$。由此可知，该蕴含在 C' 中成立。

(iii). $C.PC_p \in \{$**4-7**，**18**，**22-24**$\} \wedge C.\mathrm{S}_p = C.\textsc{Seq} - 1$
根据归纳假设**13**，我们有 $\forall q$, ($\neg(C.PC_q \in \{$**1**, **21**$\} \wedge C.$活动状态$_q = true \wedge C.\mathrm{S}_q = C.\textsc{Seq}) \wedge ($ $C.PC_q \in \{$**22-25**$\} \Rightarrow C.\mathrm{S}_q = C.\textsc{Seq} - 1)$)。如果一个 q 满足 $C.PC_q \in \{$**22-25**$\}$ ，它执行一步，则条件

会继续成立，因为 q 不会改变 SEQ 的值。如果 p 从 C 执行一步，它不会修改 S_p 或 SEQ，因此条件会继续成立。如果某个进程 $r \neq q$（其中 q 如上所述）执行一步，那么 r 不会修改 SEQ、S_p 或 S_q。因此，我们得到 C' 中的蕴含。

(b). $(C'.PC_p = \mathbf{7} \wedge C'.\mathrm{CSOWNER} \neq \bot)$
A $\Rightarrow \forall q, (\neg(C'.PC_q \in \{\mathbf{1}, \mathbf{21}\} \wedge C'.\mathrm{Active}_q = true \wedge C'.\mathrm{S}_q = C'.\mathrm{SEQ})$
$\wedge\ (C'.PC_q \in \{\mathbf{22\text{-}25}\} \Rightarrow C'.\mathrm{S}_q = C'.\mathrm{SEQ} - 1))$

假设 $(C'.PC_p = \mathbf{7} \wedge C'.\mathrm{CSOWNER} \neq \bot)$。我们按以下情况分析该条件

(i). $C.PC_p = \mathbf{6}$我们注意到，由于这一步，p 找到了 $\mathrm{CSOWNER} = \bot$，因为我们假设了 $C'.PC_p = \mathbf{7}$；否则p 会在循环中移至第 **5** 行。无论哪种情况，我们都可以得出结论：这种情况不会发生。

(ii). $C.PC_p = \mathbf{7} \wedge C.\mathrm{CSOWNER} = \bot$我们注意到，在这一步中，某个进程 q 将其名称写入了 CSOWNER。$q \neq p$，因为$C.PC_p = C'.PC_p = \mathbf{7}$。假设 $C.PC_q = \mathbf{14}$。根据 IH:14，如果 $C.\mathrm{S}_p = C.\mathrm{SEQ} - 1$，那么 $C.PC_p \neq \mathbf{7}$。如果 $C.\mathrm{S}_p = C.\mathrm{SEQ}$，根据 IH:11 $C.\mathrm{S}_q = C.\mathrm{SEQ}$，根据 IH:7 $q \in \mathrm{LOCK}[C.\mathrm{SEQ}\%3].\mathrm{CSSET}$，再次根据 IH:7 $p \in \mathrm{LOCK}[C.\mathrm{SEQ}\%3].\mathrm{CSSET}$，这与 $\mathrm{LOCK}[C.\mathrm{SEQ}\%3]$ 的互斥性质矛盾。因此，$C.PC_q = \mathbf{14}$ 不可能发生。假设 $C.PC_q = \mathbf{12}$。根据 IH:11 $C.\mathrm{S}_q = C.\mathrm{SEQ}$，并依据基础锁的互斥性质，$C.\mathrm{S}_p = C.\mathrm{SEQ} - 1$。因此，根据 IH:13，我们得到 C' 中的蕴含。假设 $C.PC_q = \mathbf{7}$。同样，我们有 $C.\mathrm{S}_q$ 或 $C.\mathrm{S}_p$ 其中之一，因为 $C.\mathrm{SEQ} - 1$。因此，再次根据 IH:13，我们得到 C' 中的蕴含。

(iii). $C.PC_p = \mathbf{7} \wedge C.\mathrm{CSOWNER} \neq \bot$在这种情况下，这一步由某个进程 $q \neq p$ 执行。根据归纳假设13，我们有 $\forall q$，$(\neg(C.PC_q \in \{\mathbf{1}, \mathbf{21}\} \wedge C.$活动状态$_q = true \wedge C.\mathrm{S}_q = C.\mathrm{SEQ})\ \wedge (C.PC_q \in \{\mathbf{22\text{-}25}\} \Rightarrow C.\mathrm{S}_q = C.\mathrm{SEQ} - 1))$。我们注意到，根据以上内容，任何 q 都无法改变 SEQ 的值，因此我们得出 C' 中的蕴含关系。

(c). $C'.PC_p \in \{\mathbf{8\text{-}15}\}$
$\Rightarrow \forall q,\ (\neg(C'.PC_q \in \{\mathbf{1}, \mathbf{21}\} \wedge C'.$活动状态$_q = true \wedge C'.\mathrm{S}_q = C'.\mathrm{SEQ})$
$\wedge\ (C'.PC_q \in \{\mathbf{22\text{-}25}\} \Rightarrow C'.\mathrm{S}_q = C'.\mathrm{SEQ} - 1))$

假设 $C'.PC_p \in \{\mathbf{8\text{-}15}\}$。我们按以下情况分析该条件。

(i). $C.PC_p = \mathbf{7} \wedge C.\mathrm{CSOWNER} \neq \bot$作为步 $C'.PC_p = \mathbf{8}$ 的结果。根据归纳假设13，我们有 $\forall q$，$(\neg(C.PC_q \in \{\mathbf{1}, \mathbf{21}\} \wedge C.$活动状态$_q = true \wedge C.\mathrm{S}_q = C.\mathrm{SEQ})\ \wedge (C.PC_q \in \{\mathbf{22\text{-}25}\} \Rightarrow C.\mathrm{S}_q = C.\mathrm{SEQ} - 1))$。注意到根据上述内容，$p$ 不会改变 SEQ 的值，因此我们得到 C'中的蕴含。

(ii). $C.PC_p = \mathbf{5} \wedge C.\mathrm{STOP}[\mathrm{S}_p\%3] = true$作为步 $C'.PC_p = \mathbf{9}$ 的结果。根据归纳假设13，我们有 $\forall q$，$(\neg(C.PC_q \in \{\mathbf{1}, \mathbf{21}\} \wedge C.$活动状态$_q = true \wedge C.\mathrm{S}_q = C.\mathrm{SEQ})\ \wedge (C.PC_q \in \{\mathbf{22\text{-}25}\} \Rightarrow C.\mathrm{S}_q = C.\mathrm{SEQ} - 1))$。注意到根据上述内容，$p$ 不会改变 SEQ 的值，因此我们得到 C'中的蕴含。

(iii). $C.PC_p \in \{\mathbf{8\text{-}15}\}$如果 $C.PC_p \neq C'.PC_p$，我们注意到 p不会改变 SEQ的值，也不会改变某个进程 $q \neq p$的变量 S_q和 Active_q，因此 C' 中的蕴含可由归纳假设13推出。因此假设 $C.PC_p = C'.PC_p$。在这种情况下，该步由某个进程 $q \neq p$执行。根据归纳假设13，我们有 $\forall q$，$(\neg(C.PC_q \in \{\mathbf{1}, \mathbf{21}\} \wedge C.\mathrm{Active}_q = true \wedge C.\mathrm{S}_q = C.\mathrm{SEQ})\ \wedge (C.PC_q \in \{\mathbf{22\text{-}25}\} \Rightarrow C.\mathrm{S}_q = C.\mathrm{SEQ} - 1))$。由上述内容可知，任何 q 都不能改变 SEQ的值，因此 C'中的蕴含成立。

14. 证明条件 14 在 C' 中成立
我们分别证明如下条件 14 的每个合取项。

(a). $(\forall q, (C'.PC_p \in \{\mathbf{7}, \mathbf{8}, \mathbf{13}\} \wedge C'.\mathrm{S}_p = C'.\mathrm{SEQ} \wedge C'.\mathrm{CSOWNER} = q)$
$\Rightarrow (C'.\mathrm{S}_q = C'.\mathrm{SEQ} - 1 \wedge q \in \mathrm{LOCK}[C'.\mathrm{S}_q\%3].\mathrm{CSSET}$

$$\underline{\wedge\, \forall r, (q \neq r \wedge C'.\mathrm{S}_r = C'.\mathrm{S}_q)}$$
$$\underline{\Rightarrow\, (C'.PC_r \in \{\textbf{1-4}, \textbf{9}, \textbf{20}, \textbf{21-28}\}}$$
$$\underline{\vee\, (C'.PC_r = \textbf{5} \Rightarrow \textsc{Stop}[C'.\mathrm{S}_r\%3] = true)))}$$

假设 $C'.PC_p \in \{\textbf{7}, \textbf{8}, \textbf{13}\} \wedge C'.\mathrm{S}_p = C'.\textsc{Seq} \wedge C'.\mathrm{CSOWNER} = q$。我们将论证分为以下几种情况，并分别论证每种情况的正确性。

(i). 假设只有在注意到 $\mathrm{CSOWNER} = \perp$ 后， $C.PC_p = \textbf{6}.p$ 才能从第 **6** 行到达第 **7** 行。由于我们有 $C'.PC_p \in \{\textbf{7}, \textbf{8}, \textbf{13}\} \wedge C'.\mathrm{S}_p = C'.\textsc{Seq} \wedge C'.\mathrm{CSOWNER} = q$，因此可推出 $C.\mathrm{CSOWNER} = \perp$ 和 $C'.\mathrm{CSOWNER} = q$，这是不可能的。由此可知，这种情况不会发生。

(ii). 假设 $C.PC_p \in \{\textbf{7}, \textbf{8}, \textbf{13}\} \wedge C.\mathrm{S}_p \neq C.\textsc{Seq}$。
根据归纳假设**11**，我们有 $C.\mathrm{S}_p = C.\textsc{Seq} - 1$，但根据我们的假设， $C'.\mathrm{S}_p = C'.\textsc{Seq}$。从算法可知， $\textsc{Seq}$ 永远不会递减，且除 p 外，没有其他进程能更改 S_p。由于 $C.PC_p \in \{\textbf{7}, \textbf{8}, \textbf{13}\}$ 和 $C'.PC_p \in \{\textbf{7}, \textbf{8}, \textbf{13}\}$，如果 p 确实从 C 执行一步到达 C'，那么它要么是在第 **7** 行前进到第 **8**行，要么是在第 **8** 行前进到第 **13** 行。我们注意到，在这两行中， p 都不会修改 S_p，因此可知，这种情况不会发生。

(iii). 假设对于任意 q， $C.PC_p \in \{\textbf{7}, \textbf{8}, \textbf{13}\} \wedge C.\mathrm{S}_p = C.\textsc{Seq} \wedge C.\mathrm{CSOWNER} \neq q$。首先注意到 $C.\mathrm{CSOWNER} = \perp$。由于 $C'.\mathrm{CSOWNER} = q$，因此 q 在第 7、12 或 14 行之一将自己的名字写入了 $\mathrm{CSOWNER}$。我们还注意到 $q \neq p$ ，因为如果 q 是 p，那么在第 7 行或第 12 行将自己的名字写入 $\mathrm{CSOWNER}$ 就意味着 $C'.PC_p = \textbf{16}$ ，而在第 14 行写入则意味着 $C'.PC_p = \textbf{15}$，这与我们的假设矛盾。如果 q 要在第 14 行写入自己的名字，根据 IH:11 $C.\mathrm{S}_q = C.\textsc{Seq}$。根据 IH:7 $q \in \textsc{Lock}[C.\textsc{Seq}\%3].\mathrm{CSSET}$。由于 $C.PC_p \in \{\textbf{7}, \textbf{8}, \textbf{13}\}$ 且 $C.\mathrm{S}_p = C.\textsc{Seq}$，根据 IH:7 $p \in \textsc{Lock}[C.\textsc{Seq}\%3].\mathrm{CSSET}$。但由于 $p \neq q$，由 $\textsc{Lock}[C.\textsc{Seq}\%3]$，的互斥性质可知， q 没有在第 14 行写入自己的名字。对第 12 行以及第 7 行和 $C.\mathrm{S}_q = C.\textsc{Seq}$ 的情况，同样的论证也成立。因此，根据归纳假设**11**，我们注意到 q 在第 7 行将自己的名字写入了 $\mathrm{CSOWNER}$，且 $C.\mathrm{S}_q = C.\textsc{Seq} - 1$。由此可得 $C'.\mathrm{S}_q = C'.\textsc{Seq} - 1$。根据归纳假设**7**，可得 $q \in \textsc{Lock}[C'.\mathrm{S}_q\%3].\mathrm{CSSET}$。对于任意满足 $C.\mathrm{S}_r = C.\mathrm{S}_q$ 和 $q \neq r$ 的 r ，有如下结论。根据 IH:7 $C.PC_r \notin \{\textbf{6-8, 11-15}\}$，并且若 $C.PC_r = \textbf{5}$，则 $C.\textsc{Stop}[C.\mathrm{S}_r\%3] = true$；这些结论均由 $\textsc{Lock}[C.\mathrm{S}_q\%3]$ 的互斥性质得出。根据 IH:11 $C.PC_r \neq \textbf{10}$，因为 $C.\mathrm{S}_r = C.\mathrm{S}_q = C.\textsc{Seq} - 1$。根据 IH:9 $C.PC_r \notin \{\textbf{15-19}\}$ ，因为 $C.\mathrm{CSOWNER} = \perp$。由此可知，对于每个满足 $q \neq r$ 和 $C.\mathrm{S}_r = C.\mathrm{S}_q$ 的 r， $C.PC_r \in \{\textbf{1-4, 9, 20, 21-28}\}$；或者，如果 $C.PC_r = \textbf{5}$，则 $\textsc{Stop}[C.\mathrm{S}_r\%3] = true$。这意味着该蕴含在 C' 中成立。

(iv). 假设 $C.PC_p \in \{\textbf{7}, \textbf{8}, \textbf{13}\} \wedge C.\mathrm{S}_p = C.\textsc{Seq} \wedge C.\mathrm{CSOWNER} = q$。
在这种情况下，我们只需论证进程执行一步会如何影响内部蕴含的情况，其中进程为 r ，满足 $q \neq r$ 和 $C.\mathrm{S}_q = C.\mathrm{S}_r$。根据 IH:14，我们知道，对于这样的 r $C.PC_r \in \{\textbf{1-4}、\textbf{9}、\textbf{20}、\textbf{21-28}\}$，或（$PC_r = \textbf{5} \Rightarrow \textsc{Stop}[C.\mathrm{S}_r\%3] = true$）。
如果 r 在行 **4** 执行一步并移至行 **5**，根据归纳假设 IH:14，我们有 $q \in \textsc{Lock}[C.\mathrm{S}_q\%3].\mathrm{CSSET}$。由此可知， r 执行这一步并非因为完成了 $\textsc{Lock}[C.\mathrm{S}_r\%3].\mathbf{try}_r()$，而是因为跳出了行 **wait till** 循环。这只有在 $\textsc{Stop}[C.\mathrm{S}_r\%3] = true$ 时才会发生，这意味着 C' 中的条件成立。

如果 r 在行 **5** 执行一步，根据归纳假设 IH:14，我们知道 $\textsc{Stop}[C.\mathrm{S}_r\%3] = true$。由此可知， r 移至行 **9**，满足 C' 中的条件。
如果 r 执行一步，在行 **9** 将其自身的 S_r 递增，因此显然满足条件 y.

(b). 我们将该蕴含拆分为以下子部分，并论证每个部分的正确性。

(i). $(\underline{(C'.PC_p \in \{\textbf{8}, \textbf{13}\} \wedge C'.\mathrm{S}_p = C'.\textsc{Seq} \wedge C'.\mathrm{CSOWNER} = \perp)} \Rightarrow (\forall r,\ ($
$C'.\mathrm{S}_r \neq \underline{C'.\textsc{Seq} - 1 \vee C'.PC_r \in \{\textbf{1-4}, \textbf{9}, \textbf{20}, \textbf{21-28}\}} \vee \underline{(C'.PC_r = \textbf{5}} \Rightarrow C'.\textsc{Stop}[C'.\mathrm{S}_r\%$
$3] = true)) \wedge \exists q,\ q \in \underline{\textsc{Lock}[(C.\textsc{Seq} - 1)\%3].\mathrm{CSSET})}$ 假设
$C'.PC_p \in \{\textbf{8}, \textbf{13}\} \wedge C'.\mathrm{S}_p = C'.\textsc{Seq} \wedge C'.\mathrm{CSOWNER} = \perp$。我们将论证分成以下不同情况，并论证每种情况下的正确性。

假设 $C.PC_p = \mathbf{7}$。对于 p 从第**8** 行移至第**7**行， $C.\text{CSOWNER} \neq \perp$。然而，我们有 $C'.\text{CSOWNER} = \perp$。因此， p 没有在 $C.\text{CSOWNER} = \perp$ 下从第**7** 行执行一步。p 也不能在 $C.\text{CSOWNER} \neq \perp$ 下从第**7** 行执行一步，因为我们有 $C.\text{CSOWNER} = \perp$，且这一步无法修改 CSOWNER。因此，这种情况不会发生。

假设 $C.PC_p \in \{\mathbf{8}, \mathbf{13}\}$ 和 $C.\text{S}_p \neq C.\text{SEQ}$。根据归纳假设**11**，我们有 $C.\text{S}_p = C.\text{SEQ} - 1$。然而，这一步不可能由 p 执行，而任何其他 $q \neq p$ 都不会修改 $C.\text{S}_p$。因此，这种情况不会发生。

假设 $C.PC_p \in \{\mathbf{8}, \mathbf{13}\}$、 $C.\text{S}_p = C.\text{SEQ}$和 $C.\text{CSOWNER} \neq \perp$。由于 PC_p在这一步中没有改变，因此这一步是由某个 $q \neq p$执行的。这个 q 必定只在行 **19**将 $\perp$ 写入 CSOWNER，因为我们假设了 $C'.\text{CSOWNER} = \perp$。根据归纳假设：9 $C.\text{CSOWNER} = q$。因此，$C'.PC_q = \mathbf{20}$，并且我们注意到，根据归纳假设 **IH:14** 合取式的第一部分，蕴含关系在 C'中成立。

假设 $C.PC_p \in \{\mathbf{8}, \mathbf{13}\}$、 $C.\text{S}_p = C.\text{SEQ}$ 和 $C.\text{CSOWNER} = \perp$。由此可知，从 C 到 C' 的这一步是由进程 $r \neq p$ 执行的，因为 PC_p 从 C 到 C' 没有变化。在这种情况下，我们只需说明进程 r 执行满足 $C.\text{S}_r = C.\text{SEQ} - 1$ 的一步会如何影响内部蕴含。根据 **IH:14**，对于这样的 r $C.PC_r \in \{$**1-4**，**9**，**20**，**21-28**$\}$ 或 $(PC_r = \mathbf{5} \Rightarrow \text{STOP}[C.\text{S}_r\%3] = true)$，我们知道。

如果 r 在行 **4** 执行一步并转到行 **5**，根据 **IH:14**，我们有 $\exists q,\ q \in \text{LOCK}[C.\text{S}_r\%3].\text{CSSET}$。由此可知， r 的这一步并非由于 $\text{LOCK}[C.\text{S}_r\%3].\textbf{try}_r()$ 完成，而是由于在行 **4** 跳出了 **一直等待，直到**循环。只有当 $\text{STOP}[C.\text{S}_r\%3] = true$ 时才会发生这种情况，这意味着条件在 C' 中成立。

如果 r 执行在第 **5** 行执行一步，根据归纳假设 **IH:14**，我们知道 $\text{STOP}[C.\text{S}_r\%3] = true$。由此可得 r move到第 **9** 行，满足 C' 中的条件。

如果 r 在第 **9** 行执行一步，它会递增自己的 S_r，因此显然满足条件。

(ii). $\underline{C'.PC_p \in \{\mathbf{14}, \mathbf{15}\}}$
$\underline{\Rightarrow (\forall r, (C'.\text{S}_r \neq C'.\text{SEQ} - 1 \vee C'.PC_r \in \{\mathbf{1\text{-}4}, \mathbf{9}, \mathbf{20}, \mathbf{21\text{-}28}\}}$
$\underline{\vee\ (C'.PC_r = \mathbf{5} \Rightarrow C'.\text{STOP}[C'.\text{S}_r\%3] = true))}$
$\underline{\wedge\ \exists q, q \in \text{LOCK}[(C.\text{SEQ} - 1)\%3].\text{CSSET})}$

假设 $C'.PC_p \in \{\mathbf{14}, \mathbf{15}\}$。我们将论证分为以下不同情形，并论证每种情形的正确性。

假设 $C.PC_p = \mathbf{13}$。由于 p 从第 **13** 行移至第 **14**行，因此可知 p 在第 **13**行注意到了 $\text{CSOWNER} = \perp$ 。于是， $C.\text{CSOWNER} = \perp$。因此，根据归纳假设 **IH:14**，该蕴含在 C'中成立。

假设 $C.PC_p \in \{\mathbf{14}, \mathbf{15}\}$。我们注意到，如果 p从 C执行了一步到达 C'，那么该蕴含在 C' 中仍然成立，因为它在 C中成立。否则，从 C 到 C' 的这一步是由进程 $r \neq p$ 执行的，因为 PC_p 从 C 到 C'并未改变。根据归纳假设 **9**，我们注意到，没有进程 q能够在第 **18**行调用 $\text{LOCK}[(\text{SEQ} - 1)\%3].$退出$_q()$；否则就会有 $\text{CSOWNER} = q$。因此，我们有 $\exists q,\ q \in \text{LOCK}[($ $C'.\text{SEQ} - 1)\%3].\text{CSSET})$。在这种情况下，我们只需论证进程 r 执行一步会如何影响内部蕴含，其中 $C.\text{S}_r = C.\text{SEQ} - 1$。这部分论证从略，因为它与上文情况 **(i)** 的论证类似。

由上述针对各个归纳步骤的论证可知，该不变量在算法每次运行的每个配置中都成立。

□