基于锁保证条件扩展并发分离逻辑的并发文件系统形式化验证

郑新民[1,2]，李明树[1]，杨秋松[1]，李文波[1]
（1.基础软件国家工程研究中心（中国科学院软件研究所），北京 100190；2.中国科学院大学，北京 100049）

摘　要：在形式化验证过程中，需要依托于逻辑，对文件系统需要满足的性质进行形式化规范描述以及证明推导。并发分离逻辑（CSL）支持并发文件系统的形式化验证，对锁不变量需满足的性质进行正确性证明，但是不支持对锁不变量进行单独推导，增加了并发文件系统整体验证设计和实现的难度和复杂度。使用锁保证条件扩展并发分离逻辑，说明锁不变量在文件系统调用执行时保持不变。使用三段式形式化规范描述方法，精确描述文件系统调用执行时需满足的性质。在证明过程中，给出形式化规范的正确性定理，即并发文件系统代码实现的执行符合形式化规范。实验结果表明，相比于已有经过验证的并发文件系统，所提文件系统提升了并发度，具有更好的性能表现，同时形式化规范具有正确性保证，并且整个形式化验证的工作量在可控范围内。

关键词：形式化验证；并发分离逻辑；文件系统
中图分类号：TP311
文献标志码：A
DOI：10.11959/j.issn.1000–436x.2025201

Formal verification of a concurrent file system based on lock guarantee conditions extended concurrent separation logic

ZHENG Xinmin[1,2], LI Mingshu[1], YANG Qiusong[1], LI Wenbo[1]
1. National Engineering Research Center of Fundamental Software (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China
2. University of Chinese Academy of Sciences, Beijing 100049, China

Abstract: During the formal verification, logic is relied to formally specify and reason about the properties that the file system must satisfy. Concurrent separation logic (CSL) is used in the formal verification of concurrent file systems and to prove the properties that lock invariants must hold. However, CSL did not support independent derivation of lock invariants, which increased the overall design and implementation complexity of concurrent file system verification. To address this, CSL was extended with lock guarantee conditions (CSL-L), ensuring that lock invariants remain unchanged during the execution of file system calls. A three-phase formal specification method was adopted to precisely describe the properties that must be hold during file system operations. During the proof process, a correctness theorem for the formal specification was established, ensuring that the execution of the concurrent file system implementation conforms to its formal specification. Experimental shows that, compared with existing verified concurrent file systems, the proposed file system improves concurrency and achieves better performance, while maintaining the correctness of the formal specification and keeping the overall verification effort within a manageable range.

Keywords: formal verification, concurrent separation logic, file system

0　引言

文件系统是操作系统的一个重要组成部分，测试[1]、模型检查[2]、符号执行[3]等方法能够发现文件系统中的一些错误，但是不能保证没有错误。通过对数学定理的证明，形式化验证方法可以保证文件系统的正确性。因此，文件系统对形式化验证方

收稿日期：2025–02–25；修回日期：2025–11–07
通信作者：郑新民，xinmin@iscas.ac.cn

法提出了新的挑战[4-6]。

基于逻辑，文件系统的形式化验证在近些年得到了广泛的研究，验证的对象也由顺序文件系统[7-10]发展到并发文件系统[11-14]。输入输出（IO）并发文件系统的形式化验证[11]最开始在顺序文件系统形式化验证方法的基础上进行改进。中央处理器（CPU，central processing unit）并发文件系统的形式化验证也是由逻辑来推动的，特别是并发分离逻辑（CSL）[15-16]。

基于崩溃条件扩展并发分离逻辑，并发、崩溃安全日志系统GoJournal[13]和事务文件系统DaisyNFS[14]分别对系统的正确性进行验证。在崩溃条件扩展并发分离逻辑内，锁资源与元数据、数据资源混合在一起，再加上需要在前置条件、后置条件以及崩溃条件内对需满足的性质进行多次形式化规范描述及推导，加大了难度和复杂度。与此同时，DaisyNFS将多个文件系统调用合并为一个事务来实现并发控制，并且只支持“安全”的事务子集，不支持共享内存上的并发，并发度较低。

以并发分离逻辑为基础，虽然可以在形式化规范中描述锁在文件系统调用执行过程中需满足的性质，但是并发分离逻辑本身没有明确说明锁在文件系统调用执行过程中如何进行性质描述及推导。

本文使用锁保证条件对并发分离逻辑进行扩展，即锁保证条件扩展并发分离逻辑（CSL-L），通过锁保证条件说明文件系统调用执行时锁需满足的性质。

为了支持在锁保证条件中对锁的状态进行表示，本文定义树形结构锁不变量。为了实现并发任务对锁不变量的推导，本文将树形结构锁不变量分为全局树形结构锁不变量和局部树形结构锁不变量。全局树形结构锁不变量是磁盘所有文件上的锁的状态的全局图。局部树形结构锁不变量依据并发任务的执行，在全局树形结构锁不变量的基础上进行分配和回收。树形结构锁不变量在文件级别实现锁，虽然相较于数据块锁粒度较粗，但是可以更好地保证文件并发控制的一致性。

依据文件系统调用的执行过程，本文使用三段式形式化规范描述方法，描述文件系统调用在执行过程中需满足的性质。对于证明，本文基于CSL-L给出形式化规范的正确性定理，证明文件系统的执行满足形式化规范。本文的主要贡献如下。

1) CSL-L，使用锁保证条件对锁需要满足的性质进行形式化规范描述和推导证明。

2) 树形结构锁不变量，通过抽象树结构支持对磁盘文件锁状态进行描述。

3) 验证并发文件系统（VCFS, verified concurrent file system），使用辅助验证工具Coq[17]形式化验证的并发文件系统，对系统的正确性提供保证。

1 相关研究工作

1.1 顺序文件系统

霍尔逻辑[18]以及崩溃霍尔逻辑[7]支持顺序文件系统的形式化验证。通过前置条件和后置条件，可以说明文件系统调用在执行前后需要满足的性质。文献[7-10]均是顺序文件系统的形式化验证研究，旨在证明文件系统不同特性的正确性，包括数据完整性、机密性等。无论是霍尔逻辑还是崩溃霍尔逻辑，逻辑本身的表达能力有限，不能支持对锁不变量以及带有复杂数据结构的并发文件系统需满足的性质进行形式化规范描述，进而进行正确性证明。

1.2 IO并发文件系统

IO并发文件系统并发执行从磁盘读取数据的操作，它的形式化验证方法是在顺序文件系统形式化验证方法的基础上演变而来的，使用相同的逻辑。例如，CIO-FSCQ（IO并发文件系统）[11]使用崩溃霍尔逻辑[7]形式化规范描述以及推导IO并发文件系统需满足的性质。IO并发文件系统不使用锁以及不需要实现并发控制，因此，崩溃霍尔逻辑可以支持IO并发文件系统的形式化验证，但是，不足以支撑CPU并发文件系统的形式化验证。

1.3 CPU并发文件系统

相比于IO并发文件系统，CPU并发文件系统的形式化验证，需要对锁以及并发任务执行需满足的性质进行形式化规范描述及证明。

并发分离逻辑对霍尔逻辑和分离逻辑[19]进行扩充，进而支持CPU并发文件系统的形式化验证。并发分离逻辑及其扩展[20]用于一系列CPU并发文件系统的形式化验证研究[13-14,21-23]。GoJournal使用崩溃条件扩展并发分离逻辑，通过崩溃条件说明发生崩溃时日志系统的状态以及如何进行一致性恢复。DaisyNFS是在GoJournal的基础上经过形式化验证的事务并发文件系统，在事务上实现并发，不

支持共享内存上的并发，并且只支持“安全”的事务子集，并发度较低。

并发分离逻辑以及扩展并发分离逻辑，不支持单独对文件系统调用执行全过程中锁需满足的性质进行形式化规范描述以及推导证明，从而提升了形式化规范的复杂度以及证明推导的难度。

2　形式化验证方法

2.1　验证框架

本文给出的形式化验证框架如图 1 所示。其中，左侧是 VCFS 调用的执行语义，定义并发控制行为，包括锁的获取和释放。中间是形式化规范，基于 CSL-L 和树形结构锁不变量，精确描述 write 等文件系统调用执行需满足的性质。右侧是形式化规范的正确性证明。

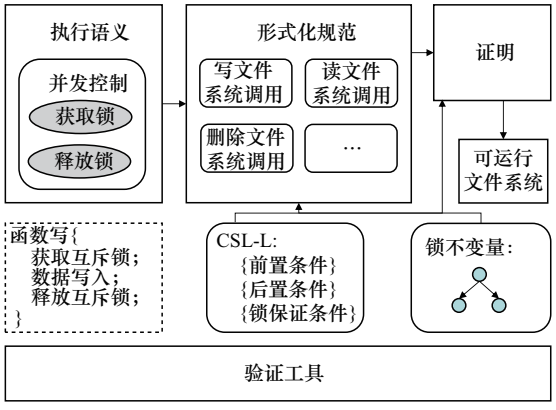

图1　形式化验证框架

2.2　锁不变量

在并发文件系统中，一种常见的策略是使用锁实现并发控制。本文使用文件锁，可以保护文件元数据以及与文件相关的任何数据，并且能更好地支持文件并发控制的一致性。

为了对文件系统调用执行期间锁需满足的性质进行形式化规范描述以及证明推导，本文使用树形结构锁不变量对锁进行刻画以及行为描述，可以直观地表示磁盘所有文件上的锁的状态。图 2 为树形结构锁不变量的示意，包含 1 个根节点、2 个子节点和 3 个叶子节点。根节点对应磁盘根目录，子节点对应子目录，叶子节点对应子目录中的文件。每一个节点上包含该文件或者目录的标识号（inum）、互斥锁计数（mutex）、共享锁计数（shared）、时间戳（time）以及子文件或者目录标识号数组（child_inums）。互斥锁和共享锁计数用于表示当前文件或目录上锁的状态，时间戳记录执行锁操作时的时间。

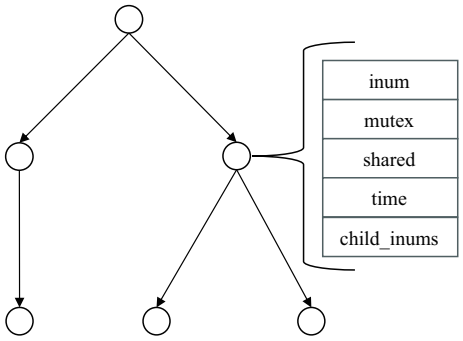

图2　树形结构锁不变量的示意

本文将树形结构锁不变量分为全局树形结构锁不变量（GlobalTree）和局部树形结构锁不变量（LocalTree）。全局树形结构锁不变量表示磁盘上所有文件的锁的状态。在文件系统调用执行过程中，依据目标文件的锁的获取和释放行为，动态修改全局树形结构锁不变量对应节点内锁的状态信息。例如，对于 write 文件系统调用，在获取互斥锁时，满足并发控制的前提下，修改树不变量对应文件节点内的互斥锁计数，置为 1（mutex=1），在释放互斥锁时，将互斥锁计数置为 0（mutex=0）。局部树形结构锁不变量依据文件系统每个并发任务的执行，随着锁的获取和释放，基于全局树形结构锁不变量，分配和回收局部树形结构锁不变量。

随着并发任务的执行，在锁的获取和释放操作频繁的情况下，动态维护锁不变量会影响验证效率，这是因为在验证过程中需要对锁不变量进行单独推导。但是，相比于锁抽象表示的常规方法，得益于树形结构锁不变量的特点，可以在树结构上快速实现并发控制。

在分配局部树形结构锁不变量时，分配算法依据全局树形结构锁不变量节点内的锁状态，满足并发控制条件后，才能执行局部树形结构锁不变量的分配，分配过程保证了局部树形结构锁不变量的不相交。算法 1 分配局部树形结构锁不变量。算法 2 依据锁的类型递归修改树形结构锁不变量节点内的锁的状态信息。算法 3 依据锁的类型递归判断树形结构锁不变量节点内的锁的状态信息是否满足并发控制条件。

算法1　局部树形结构锁不变量分配算法LockAlloc(GlobalTree, inum, type, LocalTree)

输入　全局树形结构锁不变量，目标文件或目录的标识，锁的类型；

输出　局部树形结构锁不变量；

```
1)if (type == mutex) //判断是否是互斥锁
2)   if (LockDetect(GlobalTree, inum, mutex)
          == 1)//如果不满足并发控制条件
3)        LocalTree = null;
          //不分配局部树形结构锁不变量
4)   else//如果满足并发控制条件
5)      LocalTree = sub_globalTree(inum);
          //分配局部树形结构锁不变量
6)        LockChange(LocalTree,inum,mutex);
          //修改局部锁不变量内互斥锁的状态
7)        LockChange(GlobalTree,inum,mutex);
          //修改全局锁不变量内互斥锁的状态
8)   if (type == shared)//判断是否是共享锁
9)      if (LockDetect(GlobalTree,inum, shared)
            == 1) //如果不满足并发控制条件
10)       LocalTree = null;
11)   else//如果满足并发控制条件
12)      LocalTree = sub_globalTree(inum);
13)       LockChange(LocalTree, inum, shared);
14)       LockChange(GlobalTree,inum,shared);
```

算法2　树形结构锁不变量锁状态变更算法LockChange(tree, inum, type)

输入　树形结构锁不变量tree，目标文件或目录的标识，锁的类型；

输出　变更锁状态后的树形结构锁不变量；

```
1)if (type == shared) //如果是共享锁
2)   tree(inum).shared +=1;//增加共享锁计数
3)else if (type == mutex) //如果是互斥锁
4)   tree(inum).mutex = 1;//互斥锁计数为1
5)nums = tree(inum).child_inums.length();
     //计算子节点数量
6)for (i = 0; i<nums; i++)
7)   LockChange(tree, child_inums[i], type);
       //递归执行锁状态的变更
8)end for
```

算法3　树形结构锁不变量锁状态检查算法LockDetect(tree, inum, type)

输入　树形结构锁不变量，目标文件或目录的标识，锁的类型；

输出　锁状态检查结果；

```
1)if (type == shared && tree(inum).shared)
       //如果共享锁计数不为0
2)     返回 1;
3)   else if (type == mutex &&tree(inum).mutex)
       //如果互斥锁计数为1
4)     返回 1;
5)   nums = tree(inum).child_inums.length();
     //计算子节点数量
6)   for (i = 0; i<nums; i++)
7)   LockDetect(tree,child_inums[i],type);
8)   //递归检查锁的状态
9)   end for
```

在局部树形结构锁不变量分配过程中，除了增加局部树形结构锁不变量节点内的锁计数信息外，同时增加全局树形结构锁不变量节点内的锁计数。此计数用于下一个局部树形结构锁不变量分配时的判断依据，并且随着局部树形结构锁不变量的回收而对应减小。

图3为并发任务发起后局部树形结构锁不变量的分配过程示意。当客户端发起文件操作请求时，在算法1中调用算法3，检查没有其他并发任务对同一文件进行操作后，分配局部树形结构锁不变量，然后调用算法2修改全局树形锁结构不变量和局部树形结构锁不变量内的锁计数。从图3可以看出，在算法3的保证下，3个并发任务所分配的局部树形结构锁不变量不相交。

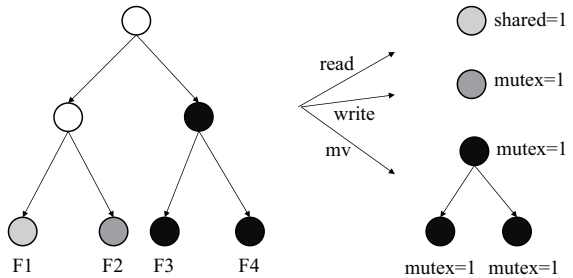

图3　并发任务发起后局部树形结构锁不变量的分配过程示意

2.3　CSL-L

并发使构建一个逻辑变得困难，需要解决以下问题：可以有效地推理许多并发模式（完整性）；必须支持在并发任务执行时保持不变的形式化规范

（可靠性）。

本文使用锁保证条件对并发分离逻辑进行扩展。在表示形式上，本文使用{Guard}表示锁保证条件，因此，将CSL的{Pre} Procedure {Post}形式扩展为{Pre} Procedure {Post} {Guard}。相比于在前置条件和后置条件中说明变化的资源需要满足的性质，在锁保证条件内，通过树形结构锁不变量，说明文件系统调用执行时不变化的锁资源需要满足的性质。图4给出了锁保证条件更直观的表示，其中，树形结构锁不变量包括一个根节点和2个叶子节点。在树形结构锁不变量节点信息内，mutex=1表示已经获取到互斥锁，child_nums指示节点间的关系。

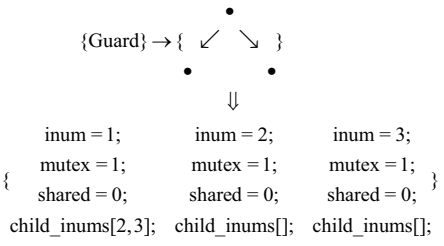

图4　锁保证条件示意图

在CSL-L内，依托并发任务，将锁保证条件内的锁不变量视为不变化的资源，和前置条件、后置条件内变化的资源一起，都需要按照文件系统代码实现的要求执行正确的操作。CSL-L要求并发任务只能访问自己的本地资源（包括锁资源），保证不同任务之间的资源的隔离，而这需要用到框架规则和并发规则。以write文件系统调用为例，扩展后的CSL-L要求框架规则需要做出对应修改。在修改后的框架规则内，不会推导其他资源{frame}。

$$\frac{\{\mathrm{Pre}\}\ \mathrm{write}\ \{\mathrm{Post}\}}{\{\mathrm{Pre}*\mathrm{frame}\}\ \mathrm{write}\ \{\mathrm{Post}*\mathrm{frame}\}}$$

$$\Downarrow$$

$$\frac{\{\mathrm{Pre}\}\ \mathrm{write}\ \{\mathrm{Post}\}\ \{\mathrm{Guard}\}}{\{\mathrm{Pre}*\mathrm{frame}\}\ \mathrm{write}\ \{\mathrm{Post}*\mathrm{frame}\}\ \{\mathrm{Guard}*\mathrm{frame}\}} \tag{1}$$

同样，需要修改CSL-L的并发规则，以并发任务$\mathrm{write}_1 \| \mathrm{write}_2$为例，分离合取符*保证2个并发写任务之间的资源不会产生干扰。对于锁保证条件，局部树形结构锁不变量的分配过程保证了2个写并发任务获取不同的私有锁不变量，即$\{\mathrm{Guard}_1\}$和$\{\mathrm{Guard}_2\}$。

$$\begin{array}{c}\{\mathrm{Pre}_1*\mathrm{Pre}_2\}\\ \{\mathrm{Pre}_1\}\quad\{\mathrm{Pre}_2\}\\ \mathrm{write}_1 \ \|\ \mathrm{write}_2\\ \{\mathrm{Post}_1\}\quad\{\mathrm{Post}_2\}\\ \{\mathrm{Post}_1*\mathrm{Post}_2\}\end{array} \Rightarrow \begin{array}{c}\{\mathrm{Pre}_1*\mathrm{Pre}_2\}\\ \{\mathrm{Pre}_1\}\quad\{\mathrm{Pre}_2\}\\ \mathrm{write}_1 \ \|\ \mathrm{write}_2\\ \{\mathrm{Post}_1\}\quad\{\mathrm{Post}_2\}\\ \{\mathrm{Guard}_1\}\quad\{\mathrm{Guard}_2\}\\ \{\mathrm{Post}_1*\mathrm{Post}_2\}\\ \{\mathrm{Guard}_1*\mathrm{Guard}_2\}\end{array} \tag{2}$$

并发规则中每个并发任务的前置条件、后置条件以及锁保证条件都只描述其私有资源。由于分离合取保证了不相交，因此并发任务C_i的每个Pre_i、Post_i以及Guard_i中的资源都不会被并发任务C_j修改（$i \neq j$）。

2.4　形式化规范

基于CSL-L以及树形结构锁不变量，本文编写形式化规范。VCFS的形式化规范以状态的形式描述每个并发任务执行所带来的磁盘数据变化。

Ext4等文件系统的实现，以write文件系统调用的执行为例，首先要获取目标文件的互斥锁，然后执行真正的磁盘数据写入操作，最后释放文件上的互斥锁。执行过程为

$$\{P\}\ \mathrm{acquire}();\ \mathrm{realwrite}();\ \mathrm{release}();\{Q\} \tag{3}$$

形式化规范与系统调用行为的对应关系如图5所示。

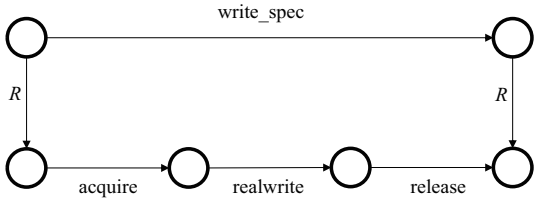

图5　形式化规范与系统调用行为的对应关系

基于上述过程，本文对VCFS形式化规范进行三段式拆分，分别基于CSL和CSL-L，精确描述文件系统调用执行过程中每一阶段需满足的性质。并发任务执行所涉及的3个阶段分别为

阶段1 acquire():

CSL: $\{L_{\mathrm{all}}\}\ C_1 \| \cdots \| C_n \{L_{\mathrm{all}}*L_1*\cdots*L_n\}$

阶段2 execute():

CSL-L: $\{P\}\ C_1 \| \cdots \| C_n \{Q\}\ \{L_1*\cdots*L_n\}$

阶段3 release():

$$\frac{\text{CSL:}\{L_{\text{all}}{*}L_1{*}\cdots{*}L_n\}\,C_1\|\cdots\|C_n\{L_{\text{all}}\}}{\{P\}\,C_1\|\cdots\|C_n\{Q\}\;L_{\text{all}}} \tag{4}$$

阶段1使用CSL，对于每一个并发任务$C_i(i=1,\cdots,n)$，在前置条件中全局树形结构锁不变量L_{all}的基础上，完成局部树形结构锁不变量$L_1{*}\cdots{*}L_n$的分配。阶段2使用CSL-L，通过锁保证条件$\{L_1{*}\cdots{*}L_n\}$，描述并发任务实际执行时锁不变量需满足的性质。此阶段中，锁资源作为不变化的资源单独进行形式化规范描述。阶段3使用CSL描述局部树形结构锁不变量$L_1{*}\cdots{*}L_n$的回收。在阶段1和阶段3中，本文将锁资源视为变化的资源，在前置条件和后置条件内进行性质描述。

首先，本文给出阶段1的形式化规范。形式化规范使用并发分离逻辑，对局部树形结构锁不变量的分配行为需满足的性质进行描述。以write文件系统调用对互斥锁的获取为例，形式化规范描述了局部树形结构锁不变量的分配

```
Pre:: { GlobalTree(inum).mutex = 0 }
Procedure::acquire(inum,mutex)
Post:: { GlobalTree(inum).mutex = 1,
         LocalTree(inum).mutex = 1 }
```
(5)

前置条件说明在全局树形结构锁不变量内，目标文件节点内的互斥锁计数为0，没有其他并发任务持有互斥锁。后置条件说明完成了局部树形结构锁不变量的分配，全局树形结构锁不变量内对应文件的互斥锁计数为1，局部树形结构锁不变量内互斥锁计数为1。

其次，本文给出阶段2的形式化规范描述，使用CSL-L以及局部树形结构锁不变量，描述实际磁盘写入操作（realwrite）执行时需满足的性质。形式化规范为

```
Pre:: { a ↦ v0 }
Procedure::realwrite(inum,a,v1)
Post:: { a ↦ v1 }
Guard:: { LocalTree(inum).mutex = 1 }
```
(6)

在形式化规范中，前置条件说明地址a的初始值为v_0。后置条件说明完成了磁盘写入，地址a的值变为v_1。锁保证条件说明磁盘写入执行期间局部树形结构锁不变量保持不变，互斥锁计数始终为1。锁保证条件同时隐含阶段1完成了局部树形结构锁不变量的分配，即锁的获取过程已经完成。

最后，本文给出阶段3的形式化规范。形式化规范使用CSL，对局部树形结构锁不变量的回收行为需满足的性质进行描述。同样，以write文件系统调用对互斥锁的回收操作为例，形式化规范描述了局部树形结构锁不变量的回收。

```
Pre:: { GlobalTree(inum).mutex = 1,
        LocalTree(inum).mutex = 1 }
Procedure::release(inum,mutex)
Post:: { GlobalTree(inum).mutex = 0 }
```
(7)

前置条件说明在全局树形结构锁不变量和局部树形结构锁不变量内，目标文件节点内的互斥锁计数为1。后置条件说明完成了局部树形结构锁不变量的回收，同时，全局树形结构锁不变量内对应文件的互斥锁计数为0。

在write文件系统调用三段式形式化规范中，包含没有写入权限或者其他并发任务正在执行写入的情形，此时锁的获取过程不会分配局部树形结构锁不变量。在此情况下，不满足realwrite形式化规范的锁保证条件，不执行write文件系统调用的实际磁盘写入操作。

2.5 证明

通过机器可检查的证明，可以证明形式化规范的正确性，即验证文件系统代码实现的执行是否满足形式化规范。在形式化规范正确性证明过程中，对并发任务的执行进行推导。形式化规范的正确性定理如下。

定理1 形式化规范是正确的，当且仅当对于并发文件系统实现和形式化规范，实现虽然在执行过程中可能因为并发多次重启，但是实现的执行满足形式化规范。也就是说，定义$m(s,c)$表示磁盘文件数据c的状态和形式化规范s内描述得一致，op表示代码实现由c_0执行到c_n，对应形式化规范为s_0、s_n，文件锁cl以及形式化规范sl，则基于CSL-L的式(8)成立。

$$\{m(s_0,c_0)\}\,\text{op}\,\{m(s_n,c_n)\}\{m(\text{sl},\text{cl})\} \tag{8}$$

其中，$m(s_0,c_0)$表示op操作执行前，磁盘文件数据c_0的状态与形式化规范s_0一致。后置条件说明op操作执行后，磁盘文件数据c_n的状态与形式化规范s_n一致。锁保证条件说明在op操作执行过程中，磁盘文件锁cl的状态与形式化规范sl内描述的

一致。

证明　数学归纳法：将并发任务在执行过程中访问的资源看作一个集合$A_i(i=0,\cdots,n)$，n为执行的步骤。当$n=0$时，表示任务执行前集合A_0的初始状态与形式化规范s_0描述的一致，即$m(s_0,c_0)$，并且锁的状态与形式化规范描述的一致，$m(\mathrm{sl},\mathrm{cl})$成立。假设$n=m-1$时，$m(s_{m-1},c_{m-1})$成立，即集合$A_{m-1}$中的资源的状态与形式化规范$s_{m-1}$一致，并且$m(\mathrm{sl},\mathrm{cl})$成立。当$n=m$时，根据代码实现执行$c_{m-1}\rightarrow c_m$，修改集合$A_{m-1}$中的资源，集合$A_{m-1}$变为$A_m$，集合$A_m$中的资源的状态与形式化规范$s_m$进行比较，$m(s_m,c_m)$成立，并且锁的状态没有发生改变，$m(\mathrm{sl},\mathrm{cl})$仍然成立。证毕。

本文使用资源来形式化表示文件系统调用的执行过程。推导证明过程跟踪文件系统调用执行时访问的任何资源的状态，文件系统调用执行前的初始状态和执行n个步骤后所产生的状态均与形式化规范描述的一致。

3　实现

本文遵循Coq辅助验证工具中验证软件的标准方法，形式化开发了VCFS。首先，依据文件系统的需求，编写执行语义。其次，编写CSL-L定义以及树形结构锁不变量。再次，基于CSL-L和树形结构锁不变量编写三段式形式化规范，精确描述文件系统调用执行需满足的性质。最后，编写形式化规范的正确性证明，证明文件系统的执行满足形式化规范。

VCFS层次结构如图6所示，包括：并发文件系统接口，接收应用发出的文件系统调用请求，供应用实现数据的存储访问；每个文件系统调用的内部实现，包括并发控制、目录实现、字节级文件实现等模块；日志系统；磁盘。

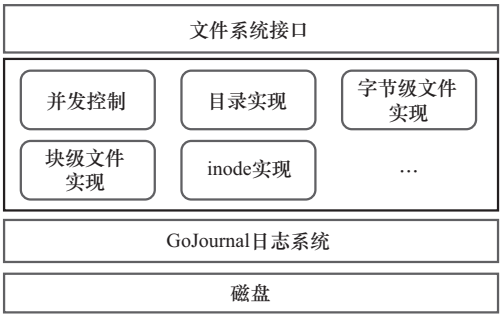

图6　VCFS层次结构

3.1　模块化

为了降低编写形式化规范以及证明的负担，本文对目标文件系统进行模块化验证。如表1所示，本文将文件系统内部实现进行拆分，在几个模块中分别进行验证。在模块化验证VCFS时，本文对模块内的功能进行如下实现：在并发控制模块使用文件锁描述文件、目录的锁状态；在字节级文件实现模块将字节级操作转换为块操作；在块级文件实现模块将数据块组织为元数据和数据；在目录实现模块将目录实现为特殊类型文件。

表1　**模块功能描述**

模块	功能描述
并发控制	实现并发访问控制
目录实现	在块级文件上实现目录
字节级文件实现	实现了文件的字节级接口，其中每个文件都是一个字节数组
块级文件实现	实现块级文件接口，向更高级别公开了一个接口，其中每个文件都是块的列表
inode实现	文件、目录索引节点实现

3.2　并发控制

VCFS在文件锁的粒度进行并发控制以及保证一致性。并发控制模块的具体实现考虑图7中的示例，以文件为单位。标号为1的文件最初包含值a，标号为4的文件包含值b。线程1正在向文件4提交b'的写入，线程2同时向文件4提交b''的写入。线程2进行写入时线程1还未完成写入，因此不向磁盘提交修改。文件2上的读操作，线程1、线程2均能完成相应的读请求。

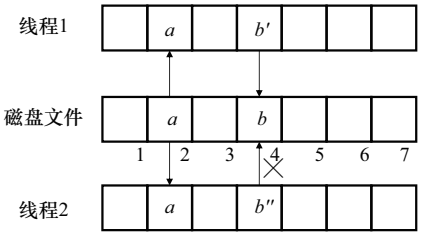

图7　并发控制实现示意

由于上述并发访问控制竞争条件，推导并发执行文件写入操作时，线程2通过互斥锁计数，观察到线程1正在执行写入，在要访问的地址写入值b'。线程2访问同一文件执行写入操作时，需要持

有目标文件的互斥锁，以此允许线程2的执行。在从全局树形结构锁不变量为线程2分配局部树形结构锁不变量时，为线程1分配的互斥锁会被观察到，不会释放线程1已经持有的互斥锁，进而不为线程2分配锁不变量。

3.3　死锁和活锁

并发控制模块在实现中需要处理的一个重要问题是死锁。以rename文件系统调用为例，将源文件或目录移动到目标位置，在这个过程中，rename文件系统调用实现变得棘手的是：它涉及多个文件，因此引入了死锁的可能性。

本文使用强制排序策略，在树形结构锁不变量节点内加入时间戳，时间戳通过Haskell的System.CPUTime获取，取值为CPU时钟周期数，进而在纳秒级实现并发控制。

在并发任务执行时，当出现要获取锁不变量的文件已经被其他线程上锁的情况，比较树形结构锁不变量节点内的时间戳，依据时间戳进行锁不变量的控制，包括执行锁抢占，进而避免线程循环相互等待。如果因为时间戳冲突而导致锁抢占失败，释放已经获取的锁，并终止并发任务的执行。

对于活锁问题，同样使用树形结构锁不变量节点内的时间戳信息，在获取锁时如果遇到时间戳冲突，释放已经获取的锁，然后终止并发任务的执行，避免活锁问题的发生。

4　实验评估

本节将从性能、多客户端、正确性以及工作量方面评估形式化验证方法。本文实验在运行Linux 4.4.0的Intel Core i7-4790 CPU、24 GB内存、240 GB INTEL D3-S4510 SERIES SSD平台实现。对于实验平台，本文安装在形式化验证过程所必需的Coq等环境中，从多角度设计实验方案，编写测试程序，统计实验数据，进而进行对比分析。

4.1　性能

本文将VCFS的性能与Ext4文件系统和DaisyNFS进行比较。选择Ext4文件系统，是因为它是商用主流文件系统，作为性能对比参照。DaisyNFS是经过形式化验证的、完整的文件系统，在事务层面实现并发，具有良好的性能。

首先，对于文件顺序读写、随机读写性能评估，本文创建1 MB大小的文件，每次顺序读写、随机读写操作针对文件内大小为1 KB的内容进行，统计执行一次操作所用的时间。如图8所示，Ext4文件系统具有最好的性能表现，VCFS的性能较DaisyNFS有一定的提升。

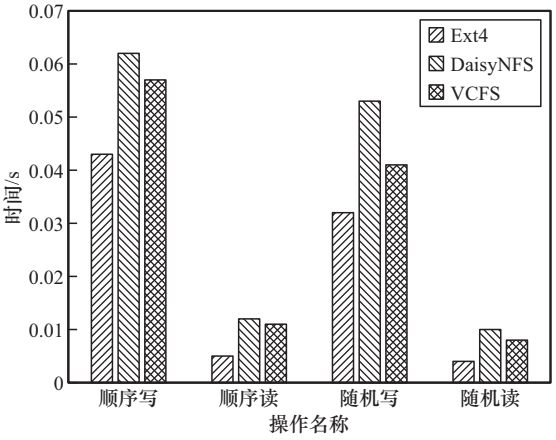

图8　顺序读写、随机读写性能对比

其次，对VCFS执行文件的创建（create）、删除（unlink）操作进行性能评估。在大小为1 KB的文件上完成上述操作，统计1 s内一共完成创建或者删除操作的次数，统计结果如图9所示。从图9可以看出，Ext4文件系统相对于DaisyNFS、VCFS具有更高的吞吐量，VCFS较DaisyNFS有一定的提升。

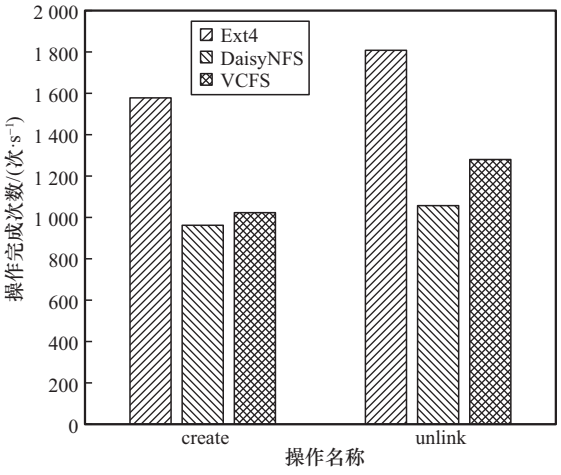

图9　create、unlink性能对比

在上述性能评估结果中，Ext4文件系统因为进行了很多性能优化（如延迟写入、批量写入等），因此具有更好的性能表现。VCFS相比于DaisyNFS，不使用事务，支持共享内存上的并发控制，进而提升了经过形式化验证的文件系统的性能。

4.2　多客户端

在并发文件系统使用过程中，往往同时执行多个客户端发出的请求。本文对VCFS在并发执行多客户端发出的请求时的性能表现进行评估。

首先，本文使用小文件的性能测试方法，测试多个客户端同时发出请求时VCFS的整体吞吐量，即每秒内完成操作的次数，实验结果如图10所示。从图10中可以看出，随着客户端数量的增加，VCFS的吞吐量也相应提高。尽管整体表现不如Ext4文件系统，但是相较于经过验证的DaisyNFS有一定的提升。

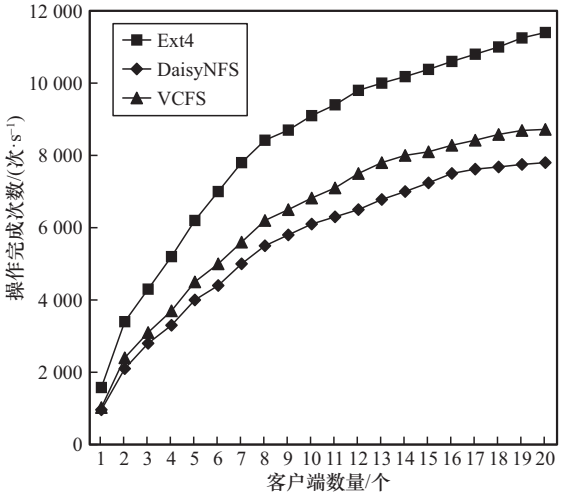

图10　高并发操作下的性能表现

其次，本文针对混合负载场景，评估高并发执行大文件读写和小文件创建、删除操作下文件系统的表现。对于大文件，统计执行一次顺序写入或者读取1 MB大小文件内1 KB内容所用的时间。对于小文件，统计执行300次创建或者删除操作所用的总时间。实验结果如图11所示。本文对3个文件系统执行2个客户端发出的全部请求所用的总时间进行比较，在混合负载场景下，VCFS执行上述全部操作的整体表现优于DaisyNFS。

再次，本文使用大文件的测试方法，评估存在锁争用时文件系统的性能表现。在实验过程中，2个客户端高并发写入同一文件，进而实现对同一文件的锁争用。本文将实验结果与不存在锁争用场景下的写入进行对比，实验结果如图12所示。从图12可以看出，存在锁争用时，一次写入操作所用的时间大于不存在锁争用时所用的时间。相比于DaisyNFS，在存在锁争用情况下，VCFS具有更好的性能表现。

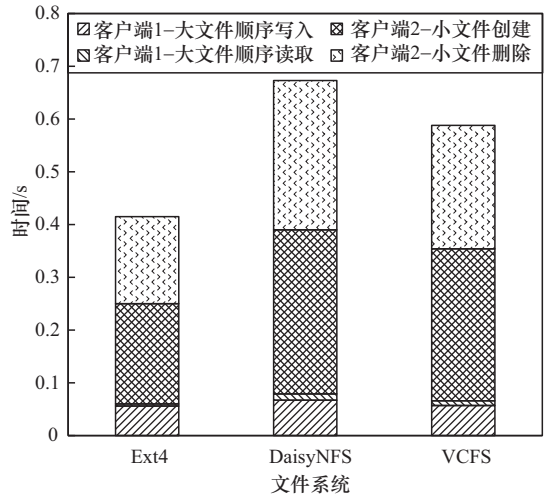

图11　混合负载场景下的性能表现

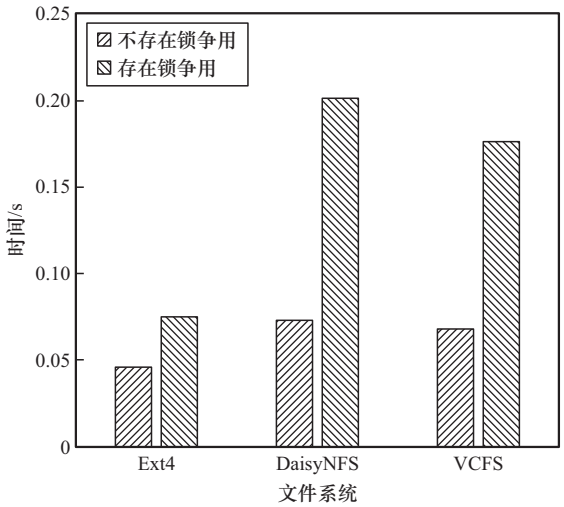

图12　大文件并发访问场景下的性能表现

最后，本文通过实验评估高并发场景下的死锁问题。本文开始尝试在实际使用场景触发死锁，但是，经过多次尝试，未能触发死锁。因此，本文通过模拟实验来评估多客户端并发操作下的死锁问题。模拟过程如下。首先，客户端1执行rename(*a*/*b*,*c*/*b*)操作，先获取到*a*/*b*的锁，对锁不变量节点内的互斥锁计数和时间戳进行修改。其次，客户端2执行rename(*c*/*b*,*a*/*b*)操作，先获取到*c*/*b*的锁，然后使用相同的时间戳尝试获取*a*/*b*的锁。此时，时间戳冲突，在此种情况下，客户端2释放了已经获取的锁，终止了执行。模拟测试结果显示VCFS避免了死锁问题的发生。对于活锁，在此模拟过程下，会终止并发任务的执行，不会出现客户端2释放以及再次获取*c*/*b*的锁的情况，进而避免了活锁问题。

依据上述实验，经过验证的VCFS，相比于已有经过形式化验证的并发文件系统，提升了并发的支持力度，进而提升了多客户端高并发场景下文件系统的整体性能表现。

4.3 正确性

本文运行了文件系统测试工具fsstress，以检查是否在VCFS中发现了错误。运行fsstress没有出现任何问题，并且在VCFS的已验证代码中没有发现任何错误。

另外，本文通过2个实验评估VCFS形式化规范实际应用的正确性。一个实验是，挂载VCFS，使用git命令克隆github代码文件，执行代码修改及提交等操作，操作均能完成。另一个实验是，使用VCFS进行文件存储管理，对文件或目录执行包括create、write等文件系统调用执行文件创建、写入等操作，操作均返回成功。

从实验结果可以看出，使用CSL-L对VCFS进行性质描述的形式化规范具备正确性保证，经过形式化验证的VCFS具备实际应用必须的功能支持，适用于诸多场景。

4.4 工作量

本文在Coq辅助验证工具中形式化开发VCFS。在VCFS形式化验证实现过程中，在FSCQ的基础上，添加CSL-L符号定义以及框架规则、并发规则，添加树形结构锁不变量的定义以及相关操作（初始化等），编写三段式形式化规范对VCFS需满足的性质进行描述，编写形式化规范的正确性证明。

从表2可以看出，CSL-L和树形结构锁不变量的开发量在一个可控的范围内。相比于CSL-L和树形结构锁不变量，形式化规范以及证明的体量要大得多。但是，对于具有一定相似度的文件系统调用（如创建文件和创建文件夹），形式化规范和证明在一定程度上可以重用，因此形式化规范和证明的工作量也在可控范围内。

表2 VCFS验证设计和实现工作量

项目	代码行数/行
CSL-L	130
树形结构锁不变量	370
形式化规范	2 700
证明	4 400

5 结束语

本文提出一种新的方法对并发文件系统进行形式化验证，使用锁保证条件对并发分离逻辑进行扩展，通过树形结构锁不变量，使用三段式形式化规范描述了文件系统需满足的性质，并对形式化规范的正确性进行证明。通过实验评估，本文方法可以提升经过形式化验证的并发文件系统的性能，提升并发度。本文使用Coq辅助验证工具对VCFS进行形式化验证，需要人工编写形式化规范及其证明。未来考虑使用自动化工具来优化验证过程，降低验证成本。对于数据库存储系统，已有验证研究对多版本并发控制事务库的正确性进行验证，使用无锁技术为长时间运行的只读事务实现高性能。对于写密集型场景，基于锁的技术可以更好地保证事务提交的正确性，本文方法可以用于写密集型场景下数据库文件系统的正确性验证。对于分布式存储系统，目前使用基于并发分离逻辑的验证方法。在存储节点间同步写入时，可以通过本文提出的逻辑扩展以及锁不变量，对同步写入的正确性进行验证。

参考文献：

[1] JOINER D K. Review of fuzz testing to find system vulnerabilities[J]. The ITEA Journal of Test and Evaluation, 2024, 45(4): 1-24.

[2] YANG J F, TWOHEY P, ENGLER D, et al. Using model checking to find serious file system errors[J]. ACM Transactions on Computer Systems, 2006, 24(4): 393-423.

[3] YANG J F, SAR C, TWOHEY P, et al. Automatically generating malicious disks using symbolic execution[C]//Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06). Piscataway: IEEE Press, 2006: 243-257.

[4] FREITAS L, WOODCOCK J, BUTTERFIELD A. POSIX and the verification grand challenge: a roadmap[C]//Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008). Piscataway: IEEE Press, 2008: 153-162.

[5] JOSHI R, HOLZMANN G J. A mini challenge: build a verifiable filesystem[J]. Formal Aspects of Computing, 2007, 19(2): 269-272.

[6] KELLER G, MURRAY T, AMANI S, et al. File systems deserve verification too![J]. ACM SIGOPS Operating Systems Review, 2014, 48(1): 58-64.

[7] CHEN H G, ZIEGLER D, CHAJED T, et al. Using Crash Hoare logic for certifying the FSCQ file system[C]//Proceedings of the 25th Symposium on Operating Systems Principles. New York: ACM Press, 2015: 18-37.

[8] CHEN H G, CHAJED T, KONRADI A, et al. Verifying a high-performance crash-safe file system using a tree specification[C]//Proceedings of the 26th Symposium on Operating Systems Principles. New York: ACM Press, 2017: 270-286.

[9] ILERI A M, CHAJED T, CHLIPALA A, et al. Proving confidentiality in a file system using DiskSec[C]//Proceedings of the USENIX Symposium on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2018: 323-338.

[10] SONG D W, MAMOURAS K, CHEN A, et al. The design and implementation of a verified file system with end-to-end data integrity[J]. arXiv Preprint, arXiv: 2012.07917, 2020 .

[11] CHAJED T. Verifying an I/O-concurrent file system[R]. 2017.

[12] HANCE T, LATTUADA A, HAWBLITZEL C, et al. Storage systems are distributed systems (so verify them that way!)[C]//Proceedings of the USENIX Symposium on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2020: 99-115.

[13] CHAJED T, TASSAROTTI J, THENG M, et al. GoJournal: a verified, concurrent, crash-safe journaling system[C]//Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Berkeley: USENIX Association, 2021:.1-17.

[14] CHAJED T, TASSAROTTI J, THENG M, et al. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning[C]// Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Berkeley: USENIX Association, 2022: 1-17.

[15] BROOKES S. A semantics for concurrent separation logic[C]// CONCUR 2004 - Concurrency Theory. Berlin: Springer, 2004: 16-34.

[16] VAFEIADIS V. Concurrent separation logic and operational semantics[J]. Electronic Notes in Theoretical Computer Science, 2011, 276: 335-351.

[17] REYNOLDS C, MONAHAN R. Reasoning about logical systems in the Coq proof assistant[J]. Science of Computer Programming, 2024, 233: 103054.

[18] HOARE C A R. An axiomatic basis for computer programming[J]. Communications of the ACM, 1969, 12(10): 576-580.

[19] REYNOLDS J C. Separation logic: a logic for shared mutable data structures[C]//Proceedings 17th Annual IEEE Symposium on Logic in Computer Science. Piscataway: IEEE Press, 2002: 55-74.

[20] JUNG R, KREBBERS R, JOURDAN J H, et al. Iris from the ground up: a modular foundation for higher-order concurrent separation logic[J]. Journal of Functional Programming, 2018, 28: e20.

[21] CHAJED T, TASSAROTTI J, KAASHOEK M F, et al. Verifying concurrent, crash-safe systems with Perennial[C]//Proceedings of the 27th ACM Symposium on Operating Systems Principles. New York: ACM Press, 2019: 243-258.

[22] CHANG Y, JUNG R, SHARMA U, et al. Verifying vMVCC, a high-performance transaction library using multi-version concurrency control[C]//Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Berkeley: USENIX Association, 2023: 1-16.

[23] SHARMA U, JUNG R, TASSAROTTI J, et al. Grove: a separation-logic library for verifying distributed systems[C]//Proceedings of the 29th Symposium on Operating Systems Principles. New York: ACM Press , 2023: 113-129.

[作者简介]

郑新民（1989– ），男，黑龙江齐齐哈尔人，中国科学院大学博士生，主要研究方向为操作系统、形式化验证。

李明树（1966– ），男，吉林德惠人，博士，中国科学院软件研究所研究员、博士生导师，主要研究方向为操作系统深度设计、可信软件过程、基础软硬件核心技术与应用等。

杨秋松（1977– ），男，河北沧州人，博士，中国科学院软件研究所研究员、博士生导师，主要研究方向为软件工程、形式化方法、模型检测、安全操作系统等。

李文波（1975– ），男，内蒙古临河人，博士，中国科学院软件研究所副研究员，主要研究方向为操作系统深度设计、软硬件深度协同等。