软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(8):3570−3586 [doi: 10.13328/j.cnki.jos.007351] [CSTR: 32375.14.jos.007351]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

操作系统内核权能访问控制的形式验证*

徐家乐[1,2,4], 王淑灵[3,4], 李黎明[1,2], 詹博华[5], 吕 毅[1,2,4], 代艺博[1,2,4], 崔舍承[1,2,4], 吴 鹏[1,2,4], 谭 宇[6], 张学军[6], 詹乃军[7]

[1](基础软件与系统重点实验室 (中国科学院 软件研究所), 北京 100190)
[2](计算机科学国家重点实验室 (中国科学院 软件研究所), 北京 100190)
[3](天基综合信息全国重点实验室 (中国科学院 软件研究所), 北京 100190)
[4](中国科学院大学, 北京 100049)
[5](华为技术有限公司, 北京 100085)
[6](北京控制与电子技术研究所, 北京 100038)
[7](北京大学 计算机学院, 北京 100091)
通信作者: 王淑灵, E-mail: wangsl@ios.ac.cn

摘 要: 操作系统内核是构建安全攸关系统软件的基础. 任何计算机系统的正确运行都依赖于底层操作系统实现的正确性, 因此, 对操作系统内核进行形式验证是很迫切的需求. 然而, 操作系统中存在的多任务并发、数据共享和竞争等行为, 给操作系统内核的验证带来很大的挑战. 近年来, 基于定理证明的方法广泛用于操作系统各功能模块的形式验证, 并取得多个成功应用. 微内核操作系统权能访问控制模块提供基于权能的细粒度访问控制, 旨在防止未经授权的用户访问系统内核资源和服务. 在权能访问控制模块实现中, 所有任务的权能空间构成多个树结构, 同时每个任务权能节点包含多种嵌套的复杂数据结构, 以及权能函数中广泛存在的对权能结构的访问、修改、(递归) 删除等操作, 使得它的形式验证与操作系统其他功能模块相比更加困难. 将以并发精化程序逻辑 CSL-R 为基础, 通过证明权能应用程序接口函数 (API 函数) 和其抽象规范之间的精化关系, 来验证航天嵌入式领域某微内核操作系统权能访问控制的功能正确性. 首先对权能数据结构进行形式建模, 并在此基础上定义全局不变式来保持权能空间的一致性; 然后定义反映功能正确性需求的内核函数的前后条件规范和 API 函数的抽象规范; 最终验证权能 API 函数 C 代码实现和抽象规范之间的精化关系. 以上所有的定义和验证均在 Coq 定理证明器中完成. 在验证过程中发现实现的错误, 并得到微内核操作系统设计方的确认和修改.

关键词: 操作系统内核; 形式验证; 权能访问控制; 并发精化分离逻辑
中图法分类号: TP311

中文引用格式: 徐家乐, 王淑灵, 李黎明, 詹博华, 吕毅, 代艺博, 崔舍承, 吴鹏, 谭宇, 张学军, 詹乃军. 操作系统内核权能访问控制的形式验证. 软件学报, 2025, 36(8): 3570–3586. http://www.jos.org.cn/1000-9825/7351.htm

英文引用格式: Xu JL, Wang SL, Li LM, Zhan BH, Lyu Y, Dai YB, Cui SC, Wu P, Tan Y, Zhang XJ, Zhan NJ. Formal Verification of Capability-based Access Control in Operating System Kernel. Ruan Jian Xue Bao/Journal of Software, 2025, 36(8): 3570–3586 (in Chinese). http://www.jos.org.cn/1000-9825/7351.htm

Formal Verification of Capability-based Access Control in Operating System Kernel

XU Jia-Le[1,2,4], WANG Shu-Ling[3,4], LI Li-Ming[1,2], ZHAN Bo-Hua[5], LYU Yi[1,2,4], DAI Yi-Bo[1,2,4], CUI She-Cheng[1,2,4], WU Peng[1,2,4], TAN Yu[6], ZHANG Xue-Jun[6], ZHAN Nai-Jun[7]

* 基金项目: 国家重点研发计划 (2022YFA1005103); 国家自然科学基金 (62432005, 62032024)
本文由“形式化方法与应用”专题特约编辑陈明帅研究员、田聪教授、熊英飞副教授推荐.
收稿时间: 2024-08-26; 修改时间: 2024-10-14; 采用时间: 2024-11-27; jos 在线出版时间: 2024-12-10
CNKI 网络首发时间: 2025-04-03

[1](Key Laboratory of System Software (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)
[2](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)
[3](National Key Laboratory of Space Integrated Information System (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)
[4](University of Chinese Academy of Sciences, Beijing 100049, China)
[5](Huawei Technologies Co. Ltd., Beijing 100085, China)
[6](Beijing Institute of Control and Electronic Technology, Beijing 100038, China)
[7](School of Computer Science, Peking University, Beijing 100091, China)

Abstract: Operating system (OS) kernels serve as the foundation for designing safety-critical software systems. The correct functioning of computer system depends on the correctness of the underlying OS kernels, making their formal verification a critical task. However, behaviors such as multi-task concurrency, data sharing, and race conditions inherent in OS kernels pose significant challenges for formal verification. In recent years, theorem-proving methods have been widely applied to verify the functionality of OS kernel modules, achieving notable successes. The capability-based access control module in OS kernels provides fine-grained access control, designed to prevent unauthorized users from accessing kernel resources and services. Its implementation involves capability spaces for tasks, which form a set of tree structures. Each capability node includes nested, complex data structures and capability functions frequently perform operations such as access, modification, and recursive deletion of capability spaces. These factors make the formal verification of capability-based access control significantly more challenging compared to other OS modules. This study employs concurrent separation logic with refinement (CSL-R) to verify the functional correctness of a capability-based access control module in the aerospace embedded domain. The verification establishes refinement between the API functions of the capability module and their abstract specifications. First, the capability data structure is formally molded, followed by the definition of a global invariant to ensure the consistency of capability spaces. Next, the preconditions and postconditions for internal functions and the abstract specifications for API functions are defined to reflect functional correctness. Finally, the refinement between the C implementation of the API functions and their abstract specifications is rigorously proven. All definitions and verification steps are formalized using the Coq theorem prover. During the verification process, errors are identified in the C implementation, which are subsequently confirmed and corrected by the OS kernel designers.

Key words: operating system kernel; formal verification; capability-based access control; concurrent separation logic with refinement (CSL-R)

软件系统是现代信息社会的基础设施, 其安全可靠性是安全攸关领域计算机系统设计的首要需求. 而计算机软件在操作系统的基础上运行, 这些软件能否正确地提供服务不仅依赖于软件本身的正确性, 还依赖于底层操作系统的正确性. 操作系统内核提供操作系统最基本的功能, 任何一个小的错误都可能造成整个上层软件系统的崩塌. 形式化建模与验证方法广泛应用于计算机软硬件系统的正确性保障, 被认为是开发高可靠软件系统的有效方法[1,2].

近年来, 基于交互式定理证明的形式验证方法被成功应用于操作系统内核的正确性验证中[3–8]. 操作系统内核提供给应用软件层一系列 API (application programming interface, 应用程序接口) 函数, 应用层通过调用 API 函数来控制程序的运行, 而不触及操作系统内核实现的具体细节. 在文献 [3–8] 中, 操作系统的正确性被定义为 API 函数的具体实现和抽象规范之间的精化关系, 而 API 函数的抽象规范反映操作系统设计时要求的功能正确需求. 与一般软件系统相比, 操作系统的正确性验证面临更大的挑战, 主要原因包括: 操作系统含有中断等引起的复杂的多任务并发行为; 操作系统内核资源包含指针、链表、结构体等复杂数据结构, 这加剧了并发所造成的数据竞争和共享问题. 因此, 操作系统正确性的验证需要同时处理并发、精化、数据共享和竞争, 这一直是学术界形式验证的难题. 并发精化程序逻辑 CSL-R[7,9]解决了以上所提出的问题, 它将并发分离逻辑[10,11]进行扩展, 加入资源不变式和上下文精化关系的概念, 实现了包含可变共享资源的并发程序的可组合和精化关系的验证. 本文将使用 CSL-R 作为验证逻辑, 对航天嵌入式领域某微内核操作系统的权能访问控制模块进行形式建模和验证, 保证其实现的功能正确性.

访问控制作为操作系统信息安全的基础机制, 旨在防止未经授权的用户或进程访问系统内核资源. 传统的访问控制机制包括自主访问控制 (DAC) 和强制访问控制 (MAC), 但随着安全需求的不断增加, 这些传统方法显得不够灵活和精细. 权能访问控制 (capability-based access control) 是一种细粒度的访问控制机制, 它将访问权限直接与对象和操作绑定, 能够更精确地控制权限的分配和使用. 权能访问控制模块管理全部的内核服务以及对相关内核资源的访问. 每个权能是一个特权对象, 包含对特定资源的访问权限, 并且精细地定义了可以进行哪些操作, 如读取、写入或执行特定功能. 权能访问控制模块的正确性验证非常关键. 然而, 与操作系统其他功能模块例如任务管

理、同步和通信、时钟管理等相比, 权能访问控制的数据结构复杂得多. 在实现中, 每个任务的权能空间涵盖指针、数组、结构体、联合体等数据结构的嵌套; 同时, 所有具有继承关系的任务的权能空间递归形成一颗树状的结构, 而权能访问控制模块需要时刻维护这颗树, 用于追踪任务的原始权能和派生权能之间的关系. 例如, 当撤回一个任务的某个权能时, 必须要递归删除其所派生出的所有子任务具有的对应权能. 因此, 复杂的共享数据结构访问和修改、递归函数调用是权能访问控制实现中常见的操作, 是对其进行形式验证必须解决的问题.

在本文中, 我们将对权能的数据结构进行形式建模, 针对权能中的不同结构体、数组、联合体等分别定义断言进行描述. 这样做的一个优势是, 可以针对权能的特殊数据结构定义统一的引理, 有助于权能函数中共性操作的验证, 包括临界区内对任务某个权能值的修改或删除、递归删除树中某个类型的权能等. 我们将权能数据结构在高层抽象为一个映射, 省略了权能树、指针、数组等具体结构信息, 用于权能 API 函数的抽象规范的定义. 具有父子关系的任务权能之间的关系、底层权能结构和高层权能映射之间的一致关系, 由全局不变式来定义. 此不变式在验证过程中, 在临界区外是时刻保持成立的, 这保证了权能数据结构的良构性以及高层规范和底层实现的一致性. 系统 API 函数通常通过调用内部函数来完成功能, 因此, 我们首先定义并验证了内部函数的前后条件规范, 在此基础上验证了 API 函数的功能正确性, 即 API 函数的具体实现和其抽象规范满足精化关系. 以上的形式化定义和验证均在 Coq 定理证明器中进行. 我们在验证过程中发现了权能访问控制实现中存在的问题并得到了该微内核操作系统设计方的确认和修改.

本文第 1 节介绍操作系统内核形式验证的研究现状. 第 2 节介绍本文所需的基础知识, 包括 Coq 定理证明和并发精化分离逻辑. 第 3 节介绍权能访问控制的设计. 第 4 节展示权能访问控制的形式验证, 包括形式建模、全局不变式、内部函数和 API 函数的形式规范和验证. 第 5 节分析验证过程中发现的问题和经验. 最后总结全文和下一步工作.

1 操作系统内核验证相关工作

基于定理证明对操作系统进行形式化验证是近年来学术研究的主要关注点之一, 并已经取得多个成功的应用. seL4[3]隶属于 L4[12]微内核家族, 是第 1 个完全形式化证明了功能正确性的操作系统微内核, 验证的内核代码 8 000 多行, Isabelle 证明代码 20 万行左右. seL4 分为 3 层: 抽象规范层, 执行规范层, C 代码实现. 它从设计需求出发, 定义了描述操作系统性质的抽象规范, 使用 Haskell 语言实现了操作系统内核的执行规范, 此执行规范可以自动翻译到 Isabelle 中进行定理证明, 同时由于效率方面的原因, 重新手动实现了操作系统内核的 C 代码. seL4 验证了 Haskell 执行规范是抽象规范的精化, 而 C 代码实现是执行规范的精化, 从而验证了操作系统实现的功能正确性. seL4 考虑由中断带来的执行不确定和交错并行行为, 但是它处于内核态时并不实时响应中断需求, 而是采取轮询 (polling) 制, 从内核态显示返回用户态并将中断看作一个新的内核事件进行调用. seL4 还是第 1 个同时支持并验证了细粒度的权能访问机制的操作系统内核. 与本文相比, seL4 在设计时把权能作为基本对象, 对于权能控制和访问直接由底层系统支撑, 通过一套消息传递的系统调用来实现; 而本文所研究的微内核权能访问控制是通过在现有的内核上提供插桩式模块实现的, 提供权能管理基础服务, 并在此之上对现有的功能模块分别添加权能管理部分, 实现权能控制和管理. 两者相比, 前者执行效率比较高, 而后者通用性和可移植性较强.

耶鲁大学 Gu 等人[4]研制了一个经过验证的安全云计算操作系统内核 CertiKOS, 之后 Gu 等人[13]提出深层规范 (deep specifications) 的概念, 将操作系统验证分为不同的抽象层次并提供相应的语言和工具实现不同层次规范的定义、模块化组合和精化验证. 作为应用, 他们形式验证了 mCertiKOS 内核 (CertiKOS 内核[4]的简化单处理器版本) 的功能正确性, 花费人月数与 seL4 相比有很大降低, 此外还利用深层规范框架验证了带中断的设备驱动程序的正确性[5]. 在 CertiKOS 项目基础上, 他们实现了 CCAL 工具集[6,14], 提出新的技术来支持多线程和多核的并行抽象层, 成功设计并验证了一个基于细粒度锁的并行操作系统内核 mC2. 然而, 基于深层规范的验证框架需要引入大量的函数调用, 来达到具体实现和调用之间的抽象分层, 当应用于已有的操作系统内核的验证时, 如何重新进行抽象分层和定义深层规范是一个很难的问题.

为了处理并行程序的验证, Liang 等人[15–17]提出了基于依赖-保证 (rely-guarantee) 的模拟关系 RGSim, 同时支

持并行组合和并发上下文精化关系的验证. 在此基础上, Xu 等人提出了以分离逻辑为基础的并发精化程序逻辑 CSL-R, 并用该逻辑验证了带中断和抢占的商业化实时嵌入式操作系统内核μC/OS-II 的核心功能模块[7,9]. Sanán 等人提出了一个基于依赖保证 (rely-guarantee) 的验证框架[8], 用于验证事件触发的并发系统, 使用该框架验证了 Zephyr 实时操作系统的内存管理模块[18]. 最近的工作中, 他们在 Isabelle/HOL 中验证了 L4 微内核 API 的功能正确性[19]. 以上工作都没有专门针对权能访问控制的形式建模和验证.

2 基础知识

本文的形式验证基于并发精化分离逻辑 CSL-R, 在 Coq 定理证明器中进行. 下面就这两方面的基本知识进行介绍.

2.1 Coq 定理证明

应用交互式定理证明进行形式验证的基本思路是: 首先, 在定理证明器中建立程序语言的语法、语义和推理系统, 建立形式模型; 然后, 对待验证系统实现的正确性和安全性质进行逻辑刻画, 定义形式规范; 最后, 基于推理系统验证形式模型满足对应的形式规范. 常见的定理证明器有 Isabelle[20], Coq[21], PVS[22], Lean[23]等. 定理证明器的可信性源于形式化证明是可以机械检查的 (machine checked), 而用于公理的元逻辑系统本身规模很小, 构成了形式验证所依赖的最小信任基. 在应用过程中, 不推荐用户直接加入公理, 用户直接定义程序的语法和语义, 而推理规则的有效性则是基于元逻辑证明系统, 证明推理规则和语义的一致性来得到. 本文所使用的 Coq 是主流的定理证明工具之一, 基于归纳构造演算 (calculus of inductive constructions, CIC) 类型理论而实现. Coq 采用可计算函数逻辑 LCF (logic for computable functions) 方法, 利用“命题即类型、证明即项”的思想, 使类型化的项既可以表示逻辑定理, 又可以表示证明, 并通过依赖类型、多态等特性实现推理与证明. Coq 不仅被用于形式化验证数学定理, 如著名的四色定理, 而且还成功地应用于系统可靠性验证, 包括操作系统和编译器等[3,4,7].

Coq 证明系统中主要使用 Inductive 关键字定义数据类型, Definition 关键字定义标识符和函数, 支持的验证策略包括 destruct, simpl, auto, eauto 等. 其中 destruct 是分类规约, 将待证明命题中的变量分成多种基本情况进行讨论. simpl 可以将目标或者前提进行简化, 但需要注意的是, 当命题中存在复杂定义的时候, simpl 可能会展开该定义, 反而增加了命题复杂度和证明难度. discriminate 是矛盾规约, 假设前件中的多个前提存在矛盾的情况, 根据蕴含关系 p 为假则命题为真, 可以直接得到待证命题为真. 而 auto 策略是一种自动化证明策略, 它可以重复若干适用于该命题的定理, 从而对于命题的多个子目标采用统一的策略进行证明, 大大降低了命题证明的难度; eauto 会在 auto 功能之上尝试对目标中的存在变量进行赋值. 除此之外, Coq 中还可以自定义 Ltac 策略, 对相似的证明目标进行自动化验证处理.

2.2 并发精化程序逻辑 CSL-R

并发精化程序逻辑 CSL-R (concurrent separation logic with refinement) 由 Xu 等人[7,9]提出, 用于对带并发和中断的操作系统内核的功能正确性验证. 它对并发分离逻辑[10,11]进行扩展, 加入并发程序模拟关系 RGSim[16,17]中对共享资源的依赖保证条件和上下文精化关系的概念, 同时实现并发程序的可组合和精化关系的验证.

CSL-R 基于临界区机制管理并发程序之间共享的资源, 实现任务对共享资源的互斥访问. 它沿用并发分离逻辑[11]的以下规则, 对并发程序语句 *ENTER_CRITICAL*; c; *EXIT_CRITICAL* 进行验证:

$$\frac{\{P*Inv\}c\{Q*Inv\}}{\{P\}ENTER\_CRITICAL;c;EXIT\_CRITICAL\{Q\}},$$

其中, P、Q 和 Inv 均为分离逻辑公式[10], P 和 Q 分别表示 c 所在线程局部资源满足的前后条件, Inv 表示全局共享资源满足的不变式. 分离合取*表示局部资源和共享资源是不相交的. 以上规则表示, 进入临界区后, c 拿到全局资源 Inv 的唯一使用权, 当 c 执行完成后, 需要重新建立起全局资源不变式 Inv, 并在退出临界区后返还. 全局不变式 Inv 一般用于刻画共享资源的良构性质以及不同资源之间的关系. CSL-R 用于验证操作系统内核实现和抽象规范之间的精化关系, 它的不变式 Inv 同时描述底层实现和高层抽象规范所对应的共享资源以及它们之间满足的一致关系, 该关系在临界区外总是保持成立, 可以看作 RGSim 中所提出的环境和系统的依赖保证关系的特殊情况. 图 1

表示 P 是 s 的精化时 P 一步执行的情况, 当 P 执行一步操作时, 存在 s 的多步执行 (包括 0 步执行) 与其对应, 它们的状态之间满足不变式 Inv.

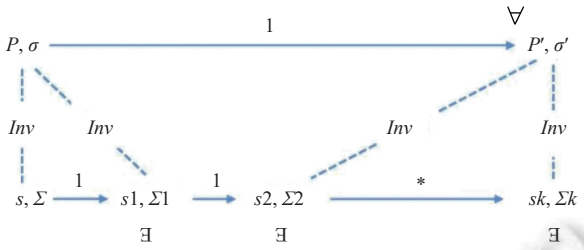


图 1 精化一致性关系

CSL-R 已经在 Coq 定理证明器中实现, 并用于带中断和抢占的操作系统内核 μC/OS-II 核心功能模块的验证[7,9]. 我们下面从建模、断言规范、推理规则这 3 个方面来简单介绍 CSL-R. CSL-R 提供类 C 语言和汇编原语对底层 C 语言内嵌汇编的内核实现进行建模, 可以很方便地将 C 代码逐行翻译到 CSL-R 语法. 同时, 它提供高层抽象规范语言, 建立 API 函数的抽象规范. 抽象规范语言中最基本的抽象操作形式为 $\gamma(vl)$, 其中 γ 为名字, 参数 vl 为值列表, 表示高层状态到返回值和高层状态的原子转换; $end\ v$ 表示当前 API 函数执行结束并返回值 v; $s1??s2$ 和 $s1;;s2$ 分别表示不确定选择和顺序执行等. CSL-R 使用分离逻辑来定义断言, 本文中用到的断言主要有以下几种 (这里使用 CSL-R 在 Coq 中的实现形式).

- $G\&x@t == l$, $L\&x@t == l$, 分别表示全局变量或局部变量 x 类型为 t 且它们的地址值为 l;
- $PV\ l@t| \rightarrow v$, 表示地址 l 处存储的值类型为 t 且值等于 v;
- $<||op||>$, 其中 op 是一个抽象规范语句, 表示高层还未执行的剩余代码为 op. CSL-R 允许在断言中表示高层抽象规范执行的情况, 对应底层实现的执行, 这是验证底层实现和高层规范之间精化关系的关键;
- $[|P|]$, 表示 P 是一个与状态无关的纯逻辑公式;
- $P**Q$ 以及其他的一阶逻辑复合公式. 其中, 分离合取 $P**Q$ 表示, 内存可以分为两个不相交的部分, P 和 Q 分别在这两部分上成立.

基于以上断言语法, CSL-R 定义了结构体、数组和链表的数据结构断言. 在本文中, 我们将对其进行扩展, 对权能的复杂数据结构进行建模.

CSL-R 中操作系统内核的正确性归结为, API 函数的实现是其抽象规范的精化, 而抽象规范需要足够表达内核设计的正确性需求. API 函数调用内部函数来实现功能, 相应地, API 函数的正确性依赖于内部函数的正确性. 针对内部函数的验证, 由于它有权限访问全局共享资源, 因此, 内部函数的前后条件规范分别描述局部资源和全局资源在函数执行前的初始情况和函数执行后修改的情况, 以及参数、局部变量和返回值的情况. 针对 API 函数的验证, 它没有权限访问全局共享资源, 因此它的前后条件中包括参数、局部变量和返回值的情况以及实现所需要精化的高层抽象规范执行的情况, 具有如下的形式:

$$\{OS\,[\bar{0},1,nil,nil]**Init(\bar{v})**EX(lv)**[|\omega(\bar{v})|]\}$$

$$s$$

$$\{\lambda\tilde{v}.OS\,[\bar{0},1,nil,nil]**EX'(lv)**[|end\,\tilde{v}|]\},$$

其中, OS 断言表示 API 调用时, 中断处于打开状态, 且该调用不位于任何中断处理程序内或者临界区内. $\bar{v}$, lv 分别表示函数的参数和局部变量. 函数调用前, 参数和局部变量分别满足初始条件 $Init$ 和 EX, 待执行的抽象规范为 ω, 与底层函数调用具有同样的参数值; 底层函数调用结束后, 返回值为 $\tilde{v}$, 同时高层规范执行完毕, 返回同样的值.

3 权能访问控制的设计

本文中所考虑的权能访问控制, 是通过在航天嵌入式领域某微内核操作系统上提供插桩式模块实现的. 它提

供权能管理基础服务, 包括权能内核对象和权能操作, 并在此之上分别对微内核其他功能模块, 如分区管理、任务管理、通信和时间管理的资源和服务进行权能管理. 以任务管理为例, 是对任务管理的各个任务程序上, 分别添加权能管理部分, 配合权能管理基础服务, 实现权能控制和管理. 前面提到的 seL4, 在设计时将权能作为基本对象, 权能和系统调用深度绑定, 所有的系统操作都是基于一套消息传递的系统调用, 系统调用在执行时需要先确认所需的权能合法. 两者相比, seL4 对于权能控制可以直接得到底层系统支撑, 执行效率比较高; 而本文插桩式的权能访问机制部署起来比较灵活, 通用性和可移植性更强. 下面我们对本文所研究的权能访问控制设计进行介绍.

权能访问控制实现对微内核资源的细粒度访问控制. 系统每创建一个任务时, 会同时创建一个与该任务绑定的权能空间, 该权能空间中存储当前任务对于内核各种资源访问的权限. 当任务在运行过程中调用 API 接口时, 权能访问控制组件会通过当前任务的优先级获取其对应的权能空间指针, 并进入存放 API 服务对应权能的地址, 查看当前任务是否具有该 API 服务所对应的权能. 如果是, 将可以正常使用该内核服务, 调用 API 接口执行操作; 否则将被拒绝, 返回无权限错误码. 为了防止用户对权能空间的修改, 所有与权能相关的数据结构对象都被放置在内核空间.

微内核权能机制提供基础的权能操作, 包括权能的移动、复制、删除、撤回等, 实现对任务权能空间的管理. 系统初始化后会创建一个根任务, 其权能空间包括对所有内核资源的操作权限. 其他任务的权能都是根任务权能的子集. 具体来说, 当一个父任务派生一个子任务时, 子任务通过权能操作从父任务继承权能, 因此子任务权能是父任务权能的子集.

我们从权能数据结构、权能操作函数和其他功能模块 API 函数的权能扩展这 3 个方面进行介绍.

3.1 权能数据结构

图 2 中定义了任务的权能数据结构, 由结构体 cspace 定义. 节点 cnode 存储任务的权能信息, 由结构体 cte 定义, 其中数组 slots[SlotNum] 中每个元素 (slot 被称为槽) 存储该任务的一个权能. 结构体 slot 定义了槽的类型和权能, 权能有两种不同类型的权能: 函数权能 (对应于 category 类) 和对象权能 (对应于 object 类), 它们的内容和值统一由联合体 content 定义. 函数权能表示是否具有调用某个内核函数的权限, 而对象权能管理系统运行过程中生成的各个内核对象的权限, 包括任务控制块对象、信号量对象、互斥量对象、消息队列对象等. 每个任务的权能空间还存储由其演化出的子任务的权能空间指针, 由 cspacechildren[MaxChildNum] 表示. 从子任务又可以递归地演化出多个其子任务, 它们的权能空间形成一棵树的结构.

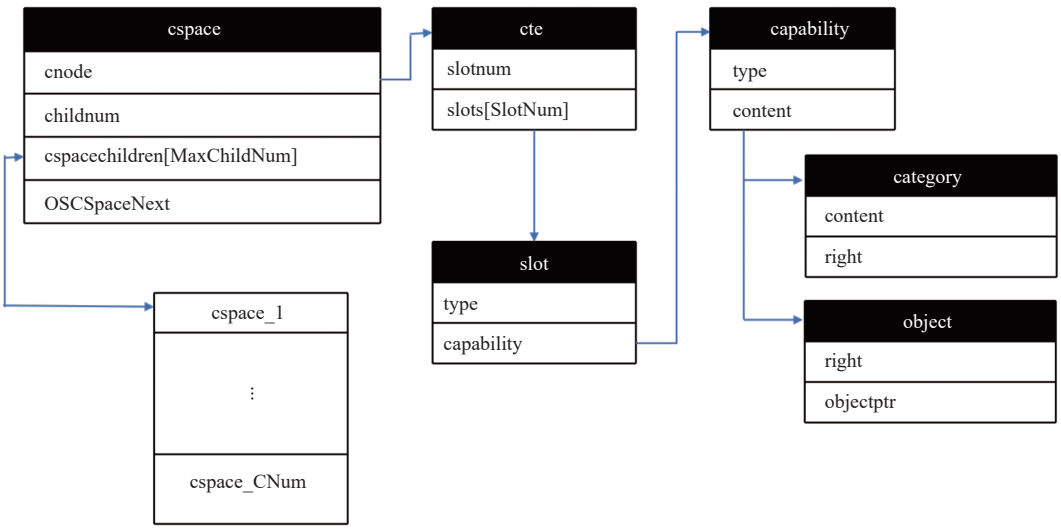


图 2 任务权能数据结构

系统的权能空间维护已分配的任务权能空间和待分配权能空间. 微内核在启动之后, 将初始化一个权能空间链表和一个保存所有已创建任务的权能空间指针的数组 CSpacePrioTbl. 在任务相继创建后, 系统的权能空间信息由多个数组和链表维护, 包括已分配任务的权能空间链表 CSpaceList、待分配权能空间的链表 CSpaceFreeList、任务优先级映射到任务权能空间指针的数组 CSpacePrioTbl, 以及任务优先级映射到任务权能空间中 slot 数组的空位置的数组 CSpaceObjectPointer. 上面我们提到, 每当任务使用内核服务时, 将检查它所对应的权能, 而对其权能的查找则是从它的优先级出发, 通过 CSpacePrioTbl 数组索引, 到达它的权能空间. 而对任务赋予新的对象权能时, 也是从其优先级出发, 通过 CSpaceObjectPointer 数组索引, 到达该任务权能 slot 数组的第 1 个空位置, 将其存储在该位置并同时将该指针移向下一个空位置.

从图 2 及以上介绍可以看出, 权能空间比较复杂, 包括结构体、数组、指针、联合体、链表等不同类型的数据结构以及它们的嵌套, 而且, 不同任务的权能空间构成多个树结构, 以维护任务及其后代之间权能的一致关系. 本文中, 我们将对 CSL-R 逻辑框架提供的数据结构断言进行扩展, 对权能数据结构进行形式建模.

3.2 权能操作函数

微内核操作系统允许任务调用权能函数, 实现赋予权能、移动权能、删除权能等操作, 其中每个操作都需要当前任务具备相关的权能操作的权限, 以及当不同任务之间移动或拷贝权能时, 需要源任务与目标任务之间存在父子关系. 微内核权能操作的概述如表 1 所示, 其中每个权能 API 函数都是通过调用对应的内核函数 (文中也称为内部函数) 来实现.

表 1 权能函数的 7 种操作

操作	API函数	描述
赋予	grant_cap	为任务赋予新的权能
铸造	cnode_mint	用指定的权能当作模板, 铸造一个新的权能, 这个权能的权限只能变小
复制	cnode_copy	将源任务的权能复制到目标任务对应的权能中
移动	cnode_move	将源任务的权能复制到目标任务对应的权能中, 然后将源任务的这个权能清空
铸造并移除	cnode_mutate	Mint+Move, 然后将源任务中的权能删除
删除	cnode_delete	删除源任务中特定 slots中的权能
撤销	cnode_revoke	删除当前权能节点所具备的权能以及递归地删除当前权能的后代节点具有的权能

3.3 其他功能模块 API 函数的权能扩展

在以上权能访问机制基础上, 操作系统内核其他涉及内核对象的功能模块需要进行权能管理扩展, 例如任务管理、信号量、互斥量、消息队列、定时器等模块. 具体扩展方法是, 在调用内核函数进行操作前, 必须检查执行操作的任务是否具备该操作相应的权限; 当具备相应权限时, 具体功能的实现与扩展前一致.

本文中我们关注的权能性质包括以下 3 类: 全局资源不变式, 即在任务执行过程中, 临界区之外, 权能数据结构总是满足一定的不变式; 功能正确性, 即权能操作实现了正确的功能; 其他函数的权能扩展性质, 即相应内核函数的调用只在任务具备相应权限时才能成功进行.

4 权能访问控制模块的形式验证

从权能访问控制模块的 C 代码实现出发, 我们在定理证明器 Coq 中开展了权能访问控制模块的形式建模和验证. 以下将从权能数据结构建模、权能全局不变式、权能内核函数和 API 函数的规范与验证这 3 个方面进行展开.

4.1 权能数据结构建模

我们由顶向下介绍权能数据结构的形式建模. 通过分析权能模块 C 代码的实现, 对于任务权能的查找和修改都是从全局数组变量 CSpacePrioTbl 出发, 经过优先级索引到达任务的权能空间进行操作. 因此, 与代码一致, 权能

数据结构具有如下的形式定义.

```
Definition CSpacePrioTbl l ptrvl vl cspacels :=
  G& CSpacePrioTbl @ (Tarray (Tptr CSPACE) 64) == l **
  Aarray l (Tarray (Tptr CSPACE) 64) ptrvl **
  CSpaceAarrayTptr 64 ptrvl vl cspacels
end
```

其中, 全局变量 CSpacePrioTbl 指向长度为 64 的 CSPACE 类型的数组起始地址 *l*, cspacels 是从任务权能空间指针到其优先级的反向映射 (它将在定义不变式中的一致关系时使用, 此处可先不考虑), Aarray 表示带类型和长度的数组, 它存储指向任务权能空间的指针, 指针值的序列为 ptrvl. CSpaceAarrayTptr 是所有任务的权能空间数据结构的并, 使用分离合取表示, *vl* 是所有任务权能的值序列. 递归地, 每个任务的权能数据结构由如下的结构断言 CSpace 定义, 其中参数 *l* 和 *vl* 分别表示该任务的权能空间的指针和值.

```
Definition CSpace l vl cspacels :=
match vl with
  | (cnode, childnum, lv, v) =&
  let (b, i) := l in
    Cnode (b, i) cnode **
    PV (b, i + typelen(Cnode) @Tint32| → Vint32 childnum **
    Aarray (b, i + typelen(Cnode) + 4) (Tarray (Tcom_ptr cspace) MaxChildNum) lv **
    PV (b, i + typelen(Cnode) + 4 + Int.mul MaxChildNum typelen(Tcom_ptr cspace)) @Tcom_ptr CSpace| → v
end
```

在以上的定义中, 为了突出关键结构定义, 我们省略了一些类型信息和表示关系的纯断言信息 (以下同). 可以看到, 每个任务的权能空间包括 4 部分, 存储在起始地址为 *l* (由内存块号 *b* 和偏移量 *i* 组成) 的一段内存内: 权能节点 Cnode, 两个 PV 断言定义基本类型的值和指针, 分别表示子任务的个数和下一个权能空间指针, Aarray 定义当前任务演化出的子任务的权能空间指针所组成的序列. 可以看到, 每个断言对应的偏移量都是起始量 *i* 和前面断言所占的内存的累加和. 权能节点 Cnode 对应的断言如下.

```
Definition Cnode l cnode :=
  match cnode with
  | (slotnum, slots)
  => let (b, i) := l in
    PV (b, i)@Tint32 | → Vint32 slotnum **
    Slots (b, i + 4) SlotNum slots
  end
```

Cnode 定义的方式类似, 但是与 CSpace 具有不同的结构, 包括类型为整型的权能槽 slot 的数目和存储权能值 slot 序列的 Slots 断言. 每一个 slot 表示一个权能, 它由一个 union 类型定义, 有两种不同的类型的权能, 由 Cap 断言定义. 以下定义中, cap_type 表示权能的类型, 两个整型值 *x* 和 *y* 构成一个 category 权能, *x* 和指针 *z* 构成一个 object 权能.

```
Definition Cap l v :=
  match v with
```

```
| (cap_type, (x, y, z)) =&
let (b, i) := l in
  PV (b, i)@Tint32 | → (Vint32 cap_type) **
  PV (b, i + 4) @ Tint32 | → Vint32 x **
  PV (b, i + 8) @ Tint32 | → Vint32 y **
  PV (b, i + 12) @ Tptr Tvoid | → z
end
```

需要说明的是, CSL-R 不允许一块内存上同时存放不同类型的数据, 这与 C 语言中 union 的定义不同. 所以在定义 Cap 断言的时候, 我们将 category 和 object 的第 2 分量 (第 1 分量类型相同) 分开定义. 上述定义如果适用于 category 权能, 那么 (x, y) 构成有效值对, z 为无效值; 如果适用于 object 权能, 那么 (x, z) 构成有效值对, y 为无效值.

至此为止, 我们定义了系统的权能数据结构. 它由多个数组、结构体、联合体嵌套而成, 中间还包括权能有关的基本数值. 前面我们提到, CSL-R 中提供了结构体和数组的一般性递归断言. 然而, 为了证明更加容易进行, 我们将权能有关的各个结构体和数组定义为具体的断言, 从而在证明时可以根据断言的具体情况进行权能数据结构的展开和合并, 来完成断言中权能的修改.

除了权能空间数据结构的断言, 我们还定义了与权能有关的其他断言, 来记录权能空间链表、各任务的父任务信息以及权能空间的空闲槽 slot 信息.

以上我们描述了操作系统内核权能模块底层实现的权能数据结构. 相对应地, 我们将底层权能数据结构抽象化, 定义高层的抽象数据状态 acspace, 类型为任务优先级到三元组的映射, 如下所示.

priority → ((list slot_val_type) * (list childs) * parent),

其中, 三元组包括: 该任务所有 slots 中存储的权能值构成的链表, 每个权能值定义和底层一致, 包括 slot 类型, 权能类型和权能的取值; 该任务所有子任务构成的链表 childs; 以及该任务的父亲 parent. 与底层相比, 具体的数据结构实现被抽象, 而保留了权能值以及任务之间的父子关系的信息.

4.2 权能全局不变式

权能的全局不变式描述与权能有关的所有共享资源, 以及不同资源之间的一致关系、底层和高层资源之间的一致关系. 以下 CapabilityInv 定义了权能不变式, 其中高层抽象状态 acspace 作为不变式的参数, 用于与其他功能模块内核层面的交互. 前两行的断言描述了底层的权能数据结构, Hcapspace 描述高层数据状态, 其余纯断言公式定义一致关系.

```
Definition CapabilityInv acspace :=
  EX l ptrvl vl cspacels pl objvl,
    CSpacePrioTbl l ptrvl vl cspacels ** CSpaceObjectPointer objvl **
    //分别为任务权能空间指针数组和任务权能空间 slot 空位置的数组
    CSpaceCapParent pl ** CSpaceGVars ** // 任务父亲信息数组, 权能全局变量
    Hcapspace acspace ** //高层权能数据状态
    [|RL_consistent(ptrvl, pl, vl)|] ** //底层各数据结构之间的一致性
    [|RLH_consistent(ptrvl, vl, acspace)|] **//底层和高层数据状态的一致性
    [|capability_wellform_descendents vl cspacels|].
```

最后一个断言表示, 每个权能节点与其后代子孙节点无环, 构成树状结构. 这既符合权能模块的设计需求, 又保证了递归操作的可终止性. 由于全局不变式定义内核共享资源, 因此只有在临界区内可以展开, 否则是保持封闭的形式, 也就是 CapabilityInv acspace.

4.3 权能内部函数和 API 函数的规范与验证

操作系统内核的正确性可以归结为, API 函数的实现是其高层规范的精化, 其中高层规范定义相应函数的功能正确性需求. 在实现中, API 函数会调用内核中的内部函数, 因此 API 函数的正确性依赖于内部函数的正确性. 我们本节将介绍 API 函数和内部函数在 CSL-R 中不同的证明模式.

下面的章节中, 我们将分别选择 grant_cap_kernel 和 cnode_revoke 作为代表, 来介绍我们的验证过程. 前者是为任务赋予新的权能的内部函数, 包含对某个具体任务的权能进行赋值的操作, 这些操作在其他权能函数中也广泛存在; 后者是递归删除某个任务及其所有后代的某个给定权能, 涉及递归不变式的定义和递归函数的验证.

4.3.1 grant_cap_kernel 的建模、规范和验证

如代码 1 所示, grant_cap_kernel 赋予任务 destTask 一个新的 CATEGORY 类型的权能 (class, right), 作为内核函数, 它具有对任务权能数据结构进行访问和修改的权利. 它首先检查参数的合法性, 若不满足, 将返回 grant_failure; 若满足, 那么将进行下面一系列的赋值操作: 从 CSpacePrioTbl 出发, 先到达任务 destTask 所对应的权能空间, 然后到达 class 索引的 slot, 最后将对应的权能类型置为 CATEGORY, 内容置为 class, 权能值修改为原来的值与参数 right 的或, 即添加 right 这个权能. 代码中"→"和".."分别表示指针取值和结构体成员访问. 以下 grant_cap_kernel 函数的实现在 Coq 中定义. 我们不难发现, CSL-R 所定义的 C 语言子集语法, 其外观与 C 语言基本类似, 增强了模型的可读性, 简化了从 C 代码实现到 Coq 模型的翻译过程.

代码 1. 内核函数 grant_cap_kernel 代码.

```
Int8u grant_cap_kernel·(Int8u destTask; Int32u class; Int32u right){
1. if(…){
2. return grant_failure;
3. }
4. …
5. CSpacePrioTbl[destTask]→cnode..slots[class]..capability..type = CATEGORY;
6. CSpacePrioTbl[destTask]→cnode..slots[class]..capability..content..category..content =class;
7. CSpacePrioTbl [destTask]→cnode..slots[class]..capability..content..category..right &=4095;s
8. CSpacePrioTbl[destTask]→cnode..slots[class]..capability..content..category..category..right ⊨ right;
}
```

内核函数的功能正确性由前后条件 (pre-/post-conditions) 规范定义, 其中前条件描述该函数顺利执行需要拥有的资源以及各种类型变量的值, 而后条件描述内核函数执行后对资源的修改. 对于 grant_cap_kernel 成功执行的情况, 它的功能正确性需求由以下前/后条件规范定义 (仅列出主要的断言).

```
{ CSpacePrioTbl l ptrvl vl cspacels ** … }
    grant_cap_kernel
{CSpacePrioTbl l ptrvl vl′ cspacels ** [|vl′ = (update_nth_cspace_slots_cap destTask class priotbl
    (CATEGORY, (class, (Right &4095) | right, z))|] **… }
```

前条件表示当前内核函数获得了权能空间资源的访问权限, 对应于 CSpacePrioTbl 断言表示的所有已有任务的权能空间; 后条件表示执行该操作后, 依旧持有权能资源的使用权限, 而且将权能值序列 *vl* 修改为 *vl′*, 其中 *vl′* 和 *vl* 满足 [|…|] 内定义的纯断言关系, update_nth_cspace_slots_cap 表示 *vl′*由 *vl* 更改对应权能得到, 它将任务 destTask 的权能空间中槽 class 对应的权能根据函数传入的参数进行权能类型、内容和值的修改. Right 和 *z* 都表示在初始 *vl* 中对应的权能值, 这里的 *z* 是占位符, 对 category 类型权能不起作用.

在 Coq 中对以上规范进行验证时, 需要按照第 4.1 节定义权能结构断言 CSpacePrioTbl 的顺序, 逐层按照参

数 class 的值进行索引并打开 (unfold) 断言, 直至到达该参数对应的 Cap 断言. 按照代码实现, 对 Cap 断言中的各个权能参数进行修改. 所有操作完成后, 需要建立后条件, 而为了得到以上规范定义的后条件, 我们需要再按照反着的顺序, 将断言进行逐层的合并 (fold), 重新得到断言 CSpacePrioTbl. 以上的证明过程非常复杂, 我们主要使用了以下两类引理 (这两个引理表示打开权能结构第 1 层, 到达断言 CSpace).

Definition CSpacePrioTbl_unfold_any_idx:=

forall *s l* ptrvl *vl* cspacels *i*,

i < len (ptrvl) →

s ⊨ CSpacePrioTbl *l* ptrvl *vl* cspacels →

s ⊨ CSpacePrioTbl *l*1 ptrvl1 *vl*1 cspacels ** CSpace ptr *v* ** CSpacePrioTbl *l*2 ptrvl2 *vl*2 cspacels

** [ptrvl = ptrvl1 ++ ptr::ptrvl2|] ** [|*vl* = *vl*1 ++ *v*::*vl*2|]**[|length ptrvl1 = *i*/\length *vl*1 = *i*|]

Definition CSpacePrioTbl_fold_any_idx:=

** [ptrvl = ptrvl1 ++ ptr::ptrvl2|] ** [|*vl* = *vl*1 ++ *v*::*vl*2|]**[|length ptrvl1 = *i*/\length *vl*1 = *i*|]

i < len (ptrvl)→

s ⊨ CSpacePrioTbl *l*1 ptrvl1 *vl*1 cspacels ** CSpace ptr *v* ** CSpacePrioTbl *l*2 ptrvl2 *vl*2 cspacels

** [ptrvl = ptrvl1 ++ ptr::ptrvl2|] ** [|*vl* = *vl*1 ++ *v*::*vl*2|]**[|length ptrvl1 = *i*/\length *vl*1 = *i*|]→

s ⊨ CSpacePrioTbl *l* ptrvl *vl* cspacels

其中, *s* ⊨ *P* 表示断言 *P* 在状态 *s* 下成立. 以上引理中, CSpacePrioTbl_unfold_any_idx 是将权能指针数组 ptrvl 和权能值数组 *vl* 根据下标 *i* 分成 [0, *i*–1]、[*i*:*i*] 和 [*i*+1, len(ptrvl)–1] 这 3 段, 其中 [*i*:*i*] 为访问的权能空间, 其对应的指针为 ptr, 对应的值为 *v*. 展开的效果是, 前后两端仍旧是权能指针数组, 而中间为一个权能单点节点, 由 CSpace 断言表示. 然后接着继续展开 CSpace 的结构, 一直到 Cap 结构, 这里我们省略了 CSpace 断言之后的展开过程. CSpacePrioTbl_fold_any_idx 是 CSpacePrioTbl_unfold_any_idx 的逆过程, 用于将展开的断言合并回去, 确保函数执行前后断言的一致性, 保持资源的良构性.

4.3.2 cnode_revoke 的建模、规范和验证

4.3.2.1 接口函数 cnode_revoke 的功能和形式模型

函数 cnode_revoke 的主要功能是, 撤销给定任务所对应的权能节点及其后代的对应权能节点. 具体来说, cnode_revoke 调用内核函数 cnode_revoke_kernel 进行权能撤销操作.

cnode_revoke 函数代码首先检查优先级 prio 是否越界, 如果越界了返回 err, 表示 revoke 操作失败. 若 prio 范围没有越界, 则调用内核函数 cnode_revoke_kernel, 尝试对于 prio 任务对应的权能节点 CSpacePrioTbl[prio] 及其后代节点进行相应的权能撤销操作, 并且将返回值存放在局部变量 *z* 中. 如果 *z* 的值是 success 表示撤销操作成功, 则 cnode_revoke 返回 revoke_success; 否则 cnode_revoke 返回 revoke_failure.

实现 1. 函数 cnode_revoke 代码.

```
Int cnode_revoke(Int8u prio, Int8u type, Int8u capability)
{
1.   Int8u z
2.   if(prio>=lowestprio){
3.      return err;
4.   }
5.   …
6.   z = cnode_revoke_kernel(prio, type, capability);
```

```
7.    if(z == success){
8.        return revoke_success;
9.    }else{
10.       return revoke_failure;
11.   }
}
```

内部函数 cnode_revoke_kernel 的函数代码 (实现 2) 首先取值 prio 任务对应的权能节点, 然后在不为空的情况下, 调用 deleterec 函数进行实际递归撤销操作, 并且返回 success, 其他情况均为错误情况, 返回 failure.

实现 2. 函数 cnode_revoke_kernel 代码.

```
Int cnode_revoke_kernel (Int8u prio, Int8u type, Int8u capability)
{
1.    cspace* pcspace;
2.    pcspace = OSCSpacePrioTbl[prio];
3.    if(pCSpace != NULL){
4.        deleterec(pcspace, type, capability);
5.    }else{
6.        return success;
7.    }
8.        return failure;
}
```

内部函数 deleterec 的函数代码 (实现 3) 先删除当前节点 cspace 对应的权能, 然后初始化一个变量 *i* 为 0, 将其用作循环变量, 递归调用 deleterec 函数来删除后代所有节点的权能.

实现 3. 函数 deleterec 代码.

```
void deleterec (CSPACE* cspace, Int8u type, Int8u capability)
{
1.    … /*分为 category 和 object 两种情况分类讨论, 删除 cspace 对应的权能*/
2.    Int8u i = 0;
3.    while(i < cspace → childnum){
4.        deleterec(cspace → cspacechildren[i], type, capability);
5.        i++;
6.    }
}
```

基于以上函数调用关系, 实现 1, 2 和 3 共同构成了 cnode_revoke 函数的实现.

4.3.2.2 接口函数 cnode_revoke 的形式规范

由于 API 函数 cnode_revoke 的正确性依赖于内核函数 cnode_revoke_kernel 的正确性, 内核函数 cnode_revoke_kernel 的正确性依赖于 deleterec 的正确性, 所以我们将按照由底向上的顺序依次建立各个函数的规范. deleterec 涉及 while 循环和递归调用, 对该函数的规范和验证, 同时具有以下两个需求.

- 使用递归断言来描述函数执行前后权能空间的改变, 对应 while 循环的不变式.

● 递归函数必须终止.

首先, 我们定义了 wellform_descendents 断言来说明当前节点与后代节点构成树状结构, 它需要一个辅助函数 wellform_descendents_list, 通过互递归定义.

```
wellform_descendents:
  | wellform_descendents_cons:
    forall map node ls listchild,
    wellform_descendents_list (child node ls map) listchild ls map →
    contains listchild node = false →
    wellform_descendents node (node :: listchild) ls map
with wellform_descendents_list:
  | wellform_descendents_list_leaf:
    forall ls map, wellform_descendents_list nil nil ls map
  | wellform_descendents_list_cons:
    forall node desc1 rest listchild1 listchild2 ls map,
    wellform_descendents node desc1 ls map →
    wellform_descendents_list rest listchild2 ls map →
    intersection listchild1 listchild2 = false →
    wellform_descendents_list (node :: rest) (listchild1 ++ listchild2) ls map.
```

wellform_descendents_list 表示一个列表中的所有节点都满足后代无环的性质. wellform_descendents 定义中, 断言 contains listchild node 为 false, 说明当前节点的后代节点列表中不包含当前节点, 从而说明当前节点和后代节点组成一个树状结构. 在 wellform_descendents_list 的定义中, 调用 wellform_descendents 来表示当前加入的节点 node 满足后代无环性质, 并且 node 的后代与 rest 列表的后代集合没有交集, 构成了一个由多个树组成的森林结构. 两者相互调用定义了后代无环的性质, 从而可以保证函数的递归执行总是可以终止的.

每个权能节点都应该和后代权能节点构成一个树状结构, 这在权能不变式 CapabilityInv 中得到保证: 不变式中的断言 capability_wellform_descendents 定义为, 系统中每个权能节点都满足 wellform_descendents 性质, 从而保证每个权能节点都满足无环性质.

然后, 我们定义了断言 *reset_descendents_cspace_slot* 来描述每次撤销操作前后权能空间所满足的性质, 表示撤销单一节点及其所有后代节点的对应权能. 同样它的定义需要互递归断言 *reset_descendents_cspace_list_slot* 来协助完成, 表示对于一个节点列表的所有结点及其后代的对应权能进行撤销. 断言 *reset_descendents_cspace_slot* 恰好定义了 deleterec 函数中 while 循环的不变式.

有了上述定义, 我们很容易建立内核函数 deleterec 和 cnode_revoke_kernel 的前后条件规范, 这里由于篇幅原因省略了.

最后, 我们列出 API 函数 cnode_revoke 的抽象规范, 形式如下.

```
cnode_revoke_spec(vl: vallist) :=
  cnode_revoke_invalid(vl) ?? /*失败的规范*/
  …/*失败的规范*/
  cnode_revoke_success(vl). /*成功的规范*/
```

其中, ??代表 cnode_revoke 函数执行结果的不确定性, 表示可能发生多种情况. 前几种规范代表执行失败情况的集合, 最后一种代表执行成功的规范. 以下将以成功的情形介绍为其建立的抽象规范. cnode_revoke_success 表示成功情形, 它对高层抽象状态进行修改, 返回 success.

$$\gamma_{\text{cnode\_revoke\_success}}(vl) \stackrel{\text{def}}{=} \lambda\Sigma,(v,\Sigma').\exists\, prio\ cspaceid\ acsapce\ acspace'.$$
$$vl = (prio :: nil) \wedge \Sigma(cspaceid) = acspace \wedge$$
$$reset\_descendents\_cspace\_slot\_abs\ prio\ acspace\ acspace' \wedge$$
$$\Sigma' = set\ \Sigma\ cspaceid\ acspace' \wedge v = \text{success},$$

其中, Σ 表示该操作执行之前的抽象状态, v 表示函数返回值. $\Sigma(cspaceid)$ 表示在抽象状态中获得权能状态 $acspace$, 由于底层资源进行递归撤销操作, 对应地, 高层资源对高层数据状态进行相同的撤销操作. 函数 *reset_descendents_cspace_slot_abs* 描述高层权能抽象状态 $acspace$ 对于当前权能节点 *prio* 进行递归撤销之后得到 $acspace'$, Σ' 将 Σ 原有的权能状态设置成 $acspace'$, 并且返回值为 success.

4.3.2.3 接口函数 cnode_revoke 的形式验证

下面介绍接口函数 cnode_revoke 的正确性定理表述, 包含该函数的前条件 *revokepre* 和 *revokepost*.

$$revokepre \stackrel{\text{def}}{=} \exists v_1,v_2,v_3,v_4.\ OS\,[\bar{0},1,nil,nil] * * [|vl = (prio :: type :: cap :: z)|] * *$$
$$PV\ prio@\ Tint32 \mapsto v_1 * * PV\ type@Tint32 \mapsto v_2 * *$$
$$PV\ cap@Tint32 \mapsto v_3 * * PV\ z@Tint32 \mapsto v_4 * * [|cnode\_revoke\_spec(vl)|]$$
$$revokepost \stackrel{\text{def}}{=} \lambda\hat{v},\exists v_1,v_2,v_3,v_4.\ OS\,[\bar{0},1,nil,nil] * * PV\ prio@\ Tint32 \mapsto v_1 * *$$
$$PV\ type@Tint32 \mapsto v_2 * * PV\ cap@Tint32 \mapsto v_3 * *$$
$$PV\ z@Tint32 \mapsto v_4 * * [|end\ \hat{v}|],$$

其中, $OS\,[\bar{0},1,nil,nil]$ 表示当前没有中断发生, 中断使能标志为 1, 这表明 cnode_revoke 函数是由用户程序调用, 中断可以随时发生. 参数 *prio*, *type*, *cap* 和局部变量 *z* 均为 int32 类型, 分别指向不同的值, 待执行的抽象规范为 *cnode_revoke_spec*, 与底层函数调用具有同样的参数值列表; 底层函数调用结束后, 返回值为 $\hat{v}$, 同时高层规范执行完毕, 返回同样的值.

值得一提的是, API 函数的验证与底层函数的验证相比, 会额外需要证明全局不变式在 API 函数调用后保持成立. 在 cnode_revoke 的证明中, 以全局不变式 RLH_consistent 为例, RLH_consistent 用来描述底层数据和高层状态之间的一致关系, 由于内核函数通过 *reset_descendents_cspace_slot* 对底层数据结构进行了递归撤销操作, 高层抽象规范通过 *reset_descendents_cspace_slot_abs* 操作对高层抽象状态进行了递归撤销操作, 证明 RLH_consistent 经过递归撤销操作之后仍然成立是本文验证中一大难点.

4.4 其他函数的权能扩展和验证

系统中关于任务管理、通讯等模块的函数需要对权能进行检查, 具体的做法就是在函数的入口处增加权能权限判断. 对于当前任务对应的权能槽判断是否有所需要的权能, 如果没有则立即结束调用, 如果有则进行下一步操作, 这样可以确保系统调用的安全性和可靠性, 阻止未授权的访问. 验证的过程和权能 API 函数非常类似, 都分别检查不同函数对应的权能. 因此, 这里不再进行详细展开.

5 验证代码规模、发现的问题与经验

本文所研究的航天嵌入式领域某微内核操作系统权能访问控制模块包括内核函数和 API 函数共计 17 个. 我们对这些函数进行了形式建模和验证, 以下统计了各个函数的代码行数和在 Coq 中验证代码的行数. 从表 2 统计可以看出, Coq 验证总代码行数大约为 35 000 行, 平均每一行 C 代码大约需要 50 行左右的 Coq 代码. 这需要在将来进一步自动化验证工作中进行改进并精简证明. 我们分析造成这方面的原因, 主要是: 在不同权能函数的验证中, 对权能数据结构进行频繁的展开, 尤其当某些操作, 例如 copy、move 等, 涉及当前任务、源任务和目标任务三方的权能的访问和修改, 因此需要在同一时间, 对系统权能结构进行 3 次展开. 而当这 3 个任务需访问和修改的权能 slot 下标的值不同时, 全局权能结构的展开更加复杂和庞大. 结束后, 也需要逐层逐个进行合并. 这都造成了验证代码的冗长.

表 2 Coq 验证代码统计

函数名称	函数代码行数	验证代码行数
grant_cap_kernel	18	709
grant_cap	35	1 160
cnode_mint_kernel	31	2 550
cnode_mint	45	1 305
cnode_copy_kernel	50	4 608
cnode_copy	55	2 480
cnode_move_kernel	55	5 445
cnode_move	57	3 208
cnode_mutate_kernel	31	1 356
cnode_mutate	43	1 580
cnode_delete_kernel	38	1 008
cnode_delete	33	1 940
deleterec	30	2 996
cnode_revoke_kernel	14	336
cnode_revoke	40	1 120
check_cap_type	63	2 390
get_obj_slot	24	1 294
总计	662	35 485

在下一步的工作中, 我们将研究权能验证过程的部分自动化, 为验证过程中大量相似的赋值操作和判断操作定义证明策略. 我们还考虑进一步将这些封装成通用的接口来供其他验证者使用. 具体思路是, 首先, 将数组、指针、结构体、联合体等不同类型的数据结构分别定义为统一的断言形式, 支持从特定类型数据结构到断言的自动生成; 其次, 以上不同数据结构的赋值语句和判断语句实质上都可以分解为求地址和从地址中取值两类操作的组合, 而又由于这些操作具有普适性, 我们将考虑针对寻址和求值定义统一的证明策略, 提高含有不同数据结构的赋值和判断语句的证明自动化程度. 以上关于不同数据结构断言的自动生成和自动验证策略的定义, 可以在操作系统其他模块甚至含有数据结构程序的验证中进行复用.

我们在验证过程中发现了权能访问控制模块代码实现中存在的问题, 并得到设计方的确认, 在新版本中得到更正并重新验证通过. 错误主要包括以下几种类型.

1) 临界区保护不充分问题. 例如, deleterec 函数的执行完全没有临界区保护, 另一个并行的任务可能在 cap 的重置过程中, 同时对这个权能结构进行操作, 从而造成数据争用问题. 其他权能函数具有同样的问题.

2) 递归引起的错误删除问题. cnode_revoke 函数在实现中检查并返回在参数 prio 任务中相应的参数 capability 权能的位置, 对当前任务和它的后代都删除这同一个位置的权能. 当 prio 为操作权能时没有问题. 但对于对象权能来说, prio 任务的 capability 权能的位置, 与它的后代的 capability 权能的位置很可能并不一样, 但是实现中使用相同的位置. 根据设计需求在 Coq 中定义函数的规范之后, 验证中发现了代码实现与规范的不一致, 于是便发现了 cnode_revoke 函数的实现错误, 提交给设计方进行修改. 在新版本中, 函数加入了一个新的参数, 用来存储权能的位置, 从而, 针对每个后代, 都可以重新去确认 capability 所存储的位置.

3) 权能操作 move 的判断条件问题. 实现中, 在调用权能移动操作时, 允许父任务和后代之间权能的双向移动. 由于任何两个任务, 都可以通过根任务相互到达, 那么根据传递性, 任意两个任务之间的权能也是可以互相移动的. 这就造成了, 证明中父任务和子任务之间权能的子集关系不再成立. 在新版本中已经修改, 只允许父任务操作子任务.

4) 条件判断不完备造成的空指针引用问题. 全局变量 CSpacePrioTbl 维护任务的 cspace, 初始值对于任何优先级为 i 的任务, 对应值为空指针. 当创建一个新的线程或者任务时, 将对应的 cspace 在 CSpacePrioTbl 中初始化为非空的值. 因此, 并不是对于所有的优先级, CSpacePrioTbl 都是非空的, 只有目前存在的任务的优先级值为非空. 然而, 每次调用用户权能函数的时候, 都没有对任务的权能的相应检查. 在验证中, 发现没有完备的条件判断, 权能全局不变式无法成立. 在修改版本中, 在 CSpacePrioTbl[i] 可能为空的地方, 都加上了是否为空的判断, 为空直接返回错误, 不为空进行操作.

6 总 结

本文介绍了航天嵌入式领域某微内核操作系统权能访问控制模块功能正确性的形式建模和验证工作. 我们以 CSL-R 验证框架为基础, 定义了权能复杂的数据结构断言和全局不变式, 对权能内部函数和 API 函数进行了形式建模, 并根据设计要求对每个函数定义了形式规范, 证明了权能 API 函数的代码实现与其抽象规范满足精化关系, 从而验证了权能访问控制的功能正确性. 在验证过程中, 我们发现了权能访问控制代码实现中存在的问题并得到设计方的确认和修改, 完成了修改后代码的验证, 对该微内核操作系统的权能访问控制的正确性提供了充分的保障. 接下来, 我们将根据本工作验证中取得的经验, 考虑部分验证代码的自动化, 提供更加通用的自动化接口, 以便在其他操作系统验证中复用, 提升验证效率.

References:

[1] Woodcock J, Larsen PG, Bicarregui J, Fitzgerald J. Formal methods: Practice and experience. ACM Computing Surveys, 2009, 41(4): 19. [doi: 10.1145/1592434.1592436]

[2] Gleirshcer M, Foster S, Woodcock J. New opportunities for integrated formal methods. ACM Computing Surveys, 2020, 52(6): 117. [doi: 10.1145/3357231]

[3] Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T, Tuch H, Winwood S. seL4: Formal verification of an OS kernel. In: Proc. of the 22nd ACM SIGOPS Symp. on Operating Systems Principles. Big Sky: ACM, 2009. 207–220. [doi: 10.1145/1629575.1629596]

[4] Gu L, Vaynberg A, Ford B, Shao Z, Costanzo D. CertiKOS: A certified kernel for secure cloud computing. In: Proc. of the 2nd Asia-Pacific Workshop on Systems. Shanghai: ACM, 2011. 3. [doi: 10.1145/2103799.2103803]

[5] Chen H, Wu X, Shao Z, Lockerman J, Gu RH. Toward compositional verification of interruptible OS kernels and device drivers. In: Proc. of the 37th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Santa Barbara: ACM, 2016. 431–447. [doi: 10.1145/2908080.2908101]

[6] Gu RH, Shao Z, Kim J, Wu X, Koenig J, Sjöberg V, Chen H, Costanzo D, Ramananandro T. Certified concurrent abstraction layers. In: Proc. of the 39th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Philadelphia: ACM, 2018. 646–661. [doi: 10.1145/3192366.3192381]

[7] Xu FW, Fu M, Feng XY, Zhang XR, Zhang H, Li ZH. A practical verification framework for preemptive OS kernels. In: Proc. of the 28th Int'l Conf. on Computer Aided Verification. Toronto: Springer, 2016. 59–79. [doi: 10.1007/978-3-319-41540-6_4]

[8] Sanán D, Zhao YW, Hou Z, Zhang FY, Tiu A, Liu Y. CSimpl: A rely-guarantee-based framework for verifying concurrent programs. In: Proc. of the 23rd Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Uppsala: Springer, 2017. 481–498. [doi: 10.1007/978-3-662-54577-5_28]

[9] Xu FW. Design and implementation of a verification framework for preemptive OS kernels [Ph.D. Thesis]. Hefei: University of Science and Technology of China, 2016 (in Chinese with English abstract).

[10] Reynolds JC. Separation logic: A logic for shared mutable data structures. In: Proc. of the 17th Annual IEEE Symp. on Logic in Computer Science. Copenhagen: IEEE, 2002. 55–74. [doi: 10.1109/LICS.2002.1029817]

[11] O'Hearn PW. Resources, concurrency and local reasoning. In: Proc. of the 15th Int'l Conf. on Concurrency Theory. London: Springer, 2004. 49–67. [doi: 10.1007/978-3-540-28644-8_4]

[12] Liedtke J. Toward real microkernels. Communications of the ACM, 1996, 39(9): 70–77. [doi: 10.1145/234215.234473]

[13] Gu RH, Koenig J, Ramananandro T, Shao Z, Wu X, Weng SC, Zhang HZ, Guo Y. Deep specifications and certified abstraction layers. In: Proc. of the 42nd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Mumbai: ACM, 2015. 595–608. [doi: 10.1145/2676726.2676975]

[14] Gu RH, Shao Z, Chen H, Kim J, Koenig J, Wu X, Sjöberg V, Costanzo D. Building certified concurrent OS kernels. Communications of the ACM, 2019, 62(10): 89–99. [doi: 10.1145/3356903]

[15] Liang HJ, Feng XY. A program logic for concurrent objects under fair scheduling. In: Proc. of the 43rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. St. Petersburg: ACM, 2016. 385–399. [doi: 10.1145/2837614.2837635]

[16] Liang HJ, Feng XY, Fu M. A rely-guarantee-based simulation for verifying concurrent program transformations. In: Proc. of the 39th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Philadelphia: ACM, 2012. 455–468. [doi: 10.1145/2103656.2103711]

[17] Liang HJ, Feng XY, Shao Z. Compositional verification of termination-preserving refinement of concurrent programs. In: Proc. of the Joint Meeting of the 23rd EACSL Annual Conf. on Computer Science Logic and the 29th Annual ACM/IEEE Symp. on Logic in Computer Science (LICS). Vienna: ACM, 2014. 65. [doi: 10.1145/2603088.2603123]

[18] Zhao YW, Sanán D. Rely-guarantee reasoning about concurrent memory management in Zephyr RTOS. In: Proc. of the 31st Int'l Conf.

on Computer-aided Verification. New York: Springer, 2019. 515–533. [doi: 10.1007/978-3-030-25543-5_29]
[19] Zhang LP, Zhao YW, Li JX. A comprehensive specification and verification of the L4 microkernel API. In: Proc. of the 30th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Luxembourg: ACM, 2024. 217–234. [doi: 10.1007/978-3-031-57249-4_11]
[20] Nipkow T, Wenzel M, Paulson LC. Isabelle/HOL: A Proof Assistant for Higher-order Logic. Berlin: Springer, 2002. [doi: 10.1007/3-540-45949-9]
[21] Huet GP, Kahn G, Paulin-Mohring C. The Coq proof assistant: A tutorial. 1997. https://flint.cs.yale.edu/cs430/coq/pdf/Tutorial.pdf
[22] Owre S, Rushby JM, Shankar N. PVS: A prototype verification system. In: Proc. of the 11th Int'l Conf. on Automated Deduction. Saratoga Springs: Springer, 1992. 748–752. [doi: 10.1007/3-540-55602-8_217]
[23] Jasti NVK, Kodali R. Lean production: Literature review and trends. Int'l Journal of Production Research, 2015, 53(3): 867–885. [doi: 10.1080/00207543.2014.937508]

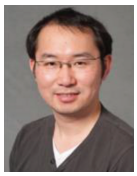
附中文参考文献:

[9] 许峰唯. 抢占式操作系统内核验证框架的设计和实现 [博士学位论文]. 合肥: 中国科学技术大学, 2016.

徐家乐(2000－), 男, 硕士生, CCF 学生会员, 主要研究领域为形式化方法.

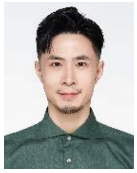
崔舍承(1999－), 男, 硕士, 主要研究领域为并发软件测试.

王淑灵(1981－), 女, 博士, 副研究员, CCF 专业会员, 主要研究领域为形式化方法, 混成系统, 软件验证.



吴鹏(1977－), 男, 博士, 副研究员, CCF 高级会员, 主要研究领域为形式化方法, 并发测试, 机器学习.

李黎明(1978－), 男, 博士, 助理研究员, 主要研究领域为形式化验证.



谭宇(1988－), 男, 博士, 工程师, 主要研究领域为形式化方法, 操作系统技术, 软件工程.

詹博华(1989－), 男, 博士, CCF 专业会员, 主要研究领域为形式化方法, 定理证明, 程序验证.

张学军(1978－), 男, 研究员, 主要研究领域为嵌入式软件, 软件工程.

吕毅(1972－), 男, 博士, 副研究员, CCF 专业会员, 主要研究领域为并发理论, 形式化方法.

詹乃军(1971－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为形式化方法, 信息物理融合系统, 程序验证.

代艺博(1999－), 男, 硕士, 主要研究领域为形式化验证, 并发理论.