软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(9):4179−4192 [doi: 10.13328/j.cnki.jos.007132]

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

微内核操作系统互斥量模块功能正确性的形式化验证*

张林雁[1], 李希萌[1,2], 施智平[1,2], 关 永[1,3], 曹钦翔[4], 张倩颖[1,3]

[1](首都师范大学 信息工程学院, 北京 100048)

[2](电子系统可靠性技术北京市重点实验室 (首都师范大学), 北京 100048)

[3](北京成像理论与技术高精尖创新中心 (首都师范大学), 北京 100048)

[4](上海交通大学 约翰霍普克罗夫特计算机科学中心, 上海 200030)

通信作者: 李希萌, E-mail: lixm@cnu.edu.cn

摘 要: 操作系统在许多安全攸关领域为软件系统提供关键性底层支撑, 操作系统中一个微小的错误或漏洞都可能引起整个软件系统的重大故障, 造成巨大经济损失或危及人身安全. 为了减少此类安全事故的发生, 对操作系统正确性进行验证十分必要. 传统测试手段无法穷尽系统中的所有潜在错误, 因而操作系统验证有必要使用具有严格数学理论基础的形式化方法. 在操作系统中, 互斥量可协调多任务对资源的访问, 是一种常用的任务同步方式, 其功能正确性对于保障多任务应用的正确性十分关键. 基于定理证明方法, 在交互式定理证明器 Coq 中对某抢占式微内核操作系统的互斥量模块进行代码级形式化建模, 给出其接口函数的形式化规范, 并实现这些接口函数的功能正确性验证.

关键词: 互斥量; 功能正确性; 形式化验证; 定理证明; Coq 定理证明器

中图法分类号: TP316

中文引用格式: 张林雁, 李希萌, 施智平, 关永, 曹钦翔, 张倩颖. 微内核操作系统互斥量模块功能正确性的形式化验证. 软件学报, 2024, 35(9): 4179–4192. http://www.jos.org.cn/1000-9825/7132.htm

英文引用格式: Zhang LY, Li XM, Shi ZP, Guan Y, Cao QX, Zhang QY. Formal Verification of Functional Correctness for Mutexes in Microkernel. Ruan Jian Xue Bao/Journal of Software, 2024, 35(9): 4179–4192 (in Chinese). http://www.jos.org.cn/1000-9825/7132.htm

Formal Verification of Functional Correctness for Mutexes in Microkernel

ZHANG Lin-Yan[1], LI Xi-Meng[1,2], SHI Zhi-Ping[1,2], GUAN Yong[1,3], CAO Qin-Xiang[4], ZHANG Qian-Ying[1,3]

[1](Information Engineering College, Capital Normal University, Beijing 100048, China)

[2](Beijing Key Laboratory of Electronic System Reliability Technology (Capital Normal University), Beijing 100048, China)

[3](Beijing Advanced Innovation Center for Imaging Theory and Technology (Capital Normal University), Beijing 100048, China)

[4](John Hopcroft Center for Computer Science, Shanghai Jiao Tong University, Shanghai 200030, China)

Abstract: Operating systems are the key foundational components of the software stacks employed in many safety-critical scenarios. A tiny error or loophole in the operating system may cause major failures of the overall software system, resulting in huge economic losses or endangering human lives. Thus, the correctness of the operating system should be verified to reduce the number of such accidents. Traditional testing methods cannot guarantee the exhaustive detection of potential errors in the target system. Therefore, it is necessary to adopt formal methods based on strict mathematical theories for verifying operating systems. In an operating system, mutexes are utilized to coordinate the access of shared resources by tasks and they are a typical means of task synchronization. The functional correctness of mutexes is the key to the correct functioning of multi-task applications. Based on the theorem proof method, this study conducts formal verification on the code of the mutex module of a preemptive microkernel in an interactive theorem prover Coq, gives the formal specifications of the interface functions of this module, and formally proves the functional correctness of these interface functions.

* 基金项目: 国家自然科学基金 (62002246, 62272322, 62272323, 62372311, 62372312, 61902240)
本文由“形式化方法与应用”专题特约编辑曹钦翔副教授、宋富研究员、詹乃军研究员推荐.
收稿时间: 2023-09-11; 修改时间: 2023-10-30; 采用时间: 2023-12-13; jos 在线出版时间: 2024-01-05
CNKI 网络首发时间: 2024-03-15

Key words: mutex; functional correctness; formal verification; theorem proof; Coq theorem prover

如今计算机系统在交通、医疗、军事等关键领域得到广泛应用, 大大改善了各类作业的自动化程度、控制精度等指标. 与此同时, 计算机系统的规模和复杂程度也在不断增长. 操作系统是计算机系统中的基础性软件平台, 操作系统的正确性是计算机系统正常运行的重要前提. 近年来, 操作系统缺陷所引发的安全事故屡屡出现. 对操作系统关键模块的正确性进行验证, 在学术界和工业界受到越来越多的重视.

互斥量是操作系统中的重要事件对象, 通常支持创建、删除、获取、释放等操作. 借助互斥量可实现不同任务对全局资源的互斥访问, 这对多任务应用的正确性具有重要支撑作用. 对于安全攸关场景中所用操作系统, 其互斥量模块中各个函数的代码是否能够实现预期功能, 在此基础上, 互斥量模块是否能在协调资源访问方面发挥其应有作用, 成为需要回答的关键问题.

对系统进行正确性验证的传统手段包括测试、仿真等, 这些手段通过对系统的具体执行, 发现系统中的问题. 由于无法遍历系统所有可能的执行路径, 这些方法难以确保系统中所有潜在错误均被发现, 无法真正实现系统的可靠验证, 往往不足以提供安全攸关系统所需的正确性保障.

形式化方法[1]能够克服上述局限性, 它依托数学手段, 给出计算机系统的模型和规范, 并能够以数学结果的形式给出模型满足规范的验证结论. 形式化方法在操作系统验证工作中所起的作用不容小觑. 模型检验和定理证明是较为典型的两大类形式化方法. 模型检验自动化程度高, 但在验证规模较大的系统时可因状态爆炸问题而导致效能削弱. 定理证明依托数学推理完成验证, 其过程往往较为繁琐, 但不受系统状态空间规模、目标性质复杂程度制约. 许多定理证明工具能够确保验证中的推理过程可分解为数理逻辑规则所允许的基本步骤, 提供高度可信的验证结果.

某微内核操作系统是基于开源抢占式操作系统 μC/OS-II 设计和实现的国产操作系统, 其互斥量模块对 μC/OS-II 互斥量进行封装和扩展, 在 μC/OS-II 互斥量基础上额外支持嵌套互斥量被重复获取与释放的功能. 为此, 该系统引入新的互斥量数据结构, 其中包含 μC/OS-II 中对应数据结构的引用以及表示额外属性值的成员. 该系统的互斥量接口函数通常执行一系列属性值检查, 并调用 μC/OS-II 中相应函数完成内部功能.

本文使用定理证明方法, 在交互式定理证明器 Coq 中实现某微内核操作系统互斥量模块接口函数的功能正确性证明. 本文的验证工作主要包括下列技术环节.

1) 对某微内核操作系统互斥量模块各接口函数进行形式化定义, 并给出其形式化规范.

2) 对某微内核操作系统的互斥量数据结构构造其所满足的不变式条件.

3) 证明某微内核操作系统互斥量模块各接口函数的功能正确性.

基于并发精化分离逻辑 CSL-R, 已有工作对 μC/OS-II 的常用 API 函数完成了验证[2], 这是首次在定理证明器中验证了支持多级中断和任务抢占的操作系统. 并发精化分离逻辑 CSL-R 能够处理对象系统中的多级中断. 此外, 它将操作系统抽象规范与具体实现之间精化关系的验证归结为对操作系统中每个函数的验证, 这在一定程度上提升了验证的组合性, 提高了验证效率.

本文的验证工作建立在 μC/OS-II 验证框架基础上, 力图实现对 μC/OS-II 中相关函数形式化规范、证明代码的复用. 此外, 本文扩展该验证框架, 以便在验证中表达关于用户使用内核方式的假设条件. 最后, 目标操作系统接口函数中较少使用临界区 (以避免实时性能的影响), 这导致在程序推理过程中无法获取所需的全局资源. 本工作采取小临界区技术解决该问题. 总体上, 本文提供一种方法, 假设某微内核 X 已使用并发精化分离逻辑加以验证, 而微内核 X'由 X 扩展得到, 使用本文方法能够充分利用 X 的验证结果完成对 X'的形式化验证. 本文的技术工作在 Coq 定理证明器中实现为约 17k 行代码.

下文讨论验证工作中的主要技术环节、对 CSL-R 验证框架的扩展、难点问题的解决、主要经验教训, 并简要说明通过验证在目标操作系统中所发现的主要问题. 具体而言, 第 1 节介绍基于定理证明方法验证操作系统的相关工作. 第 2 节介绍验证工具 Coq、互斥量运行机制以及精化程序逻辑 CSL-R 的相关知识. 第 3 节介绍支持用户假设条件的验证框架扩展. 第 4 节讨论某微内核操作系统互斥量模块关键数据、不变量的描述. 第 5 节通过示例反映接口函数形式化规范及其证明过程. 第 6 节给出验证代码量具体信息, 简要介绍验证中发现目标系统的主要问题, 并讨论完成本工作过程中的经验和教训. 第 7 节对本工作和本文进行总结.

1 相关工作

近年来, 已有许多团队基于定理证明方法对操作系统功能正确性以及安全性进行验证, 下面具体介绍相关验证工作.

Baumann 等人通过使用验证工具 VCC[3]验证了 PikeOS 系统内核中一个更新任务优先级的系统调用函数的功能正确性[4], 并使用 VCC 验证了一个商用的虚拟机软件 Hyper-V 的部分代码[5], 此验证工作未对 PikeOS 和 Hyper-V 进行大规模验证. Klein 等人将操作系统 seL4 分为 3 个层次, 分别为: 抽象层规约、执行层规约和底层 C 语言实现的操作系统代码, 通过 C 代码层与执行层规约之间的精化关系和执行层规约与抽象层规约之间的精化关系实现了 seL4 微内核的功能正确性验证[6−8]. Gu 等人将操作系统 mCertiKOS 分解成 37 层, 通过逐层精化关系完成了对此操作系统的功能正确性验证工作[9−11], 该团队还验证了一个基于操作系统 mCertiKOS 实现的更细粒度的并发操作系统的功能正确性[12], 并在前者基础上验证了系统中设备驱动程序的正确性[13]. 此外, Liu 等人对操作系统 mCertiKOS 进行扩展, 提出一个新的组合框架用以推理抢占调度程序, 并在该框架中验证了支持用户级抢占的实时操作系统内核的功能正确性[14]. Andronick 等人实现了抢占式实时系统 eChronos 的功能正确性[15], 其中包括中断处理程序、上下文切换等内容. 在前面所述工作中, seL4 和 mCertiKOS 均不是抢占式操作系统, 它们不允许中断在任务执行的任何时机得到服务. eChronos 的验证工作中未对互斥量模块功能正确性进行形式化验证. 本文工作则是对抢占式微内核操作系统中互斥量模块接口函数功能正确性进行验证. 除此之外, Xu 等人提出了并发精化分离逻辑 (CSL-R), 并使用此框架验证了支持抢占式的微内核操作系统 μC/OS-II 的功能正确性, 主要包括中断处理、任务调度、互斥量等模块 API 函数, 其中验证的互斥量模块包括互斥量创建、删除等 5 个函数证明[2], 但该系统中互斥量不支持嵌套. 顾海博扩展了 μC/OS-II 的验证框架, 并在扩展框架中证明了目标系统关键模块的 8 条全局性质的正确性, 其中互斥量模块性质有两条[16]. 张啸然对实时操作系统中底层 C 代码实现与高层规约之间的精化关系进行验证, 并通过在高层证明规约满足有上界优先级反转性质得到底层有上界优先级反转的性质[17]. 与已有的 μC/OS-II 验证工作相比, 本文通过扩展 CSL-R 程序逻辑等方式, 在操作系统内核的验证中更好地表示用户所提供数据, 并反映与之相关的假设条件. 此外, 本文通过小临界区技术有效处理无临界区代码的验证. 本文给出一种充分基于已有微内核验证结果, 实现扩展微内核验证的解决方案. 在具体验证工作方面, 本文验证了一系列全新实现的接口函数和内部函数代码的功能正确性.

另外一些工作侧重于操作系统信息安全性, 而非功能正确性的验证. 例如, Zhao 等人证明了满足 ARINC 653 标准的分离内核具有信息流安全性[18−20]. Ma 等人形式化验证了基于 ARM TrustZone 的 TEE 系统具有内存隔离的安全性[21]. 由于本工作不涉及信息安全性验证, 具体内容不在此进行讨论.

2 基础知识

本文以操作系统中的互斥量为验证对象, 以交互式定理证明器 Coq 为验证工具, 基于 μC/OS-II 的并发精化程序逻辑 CSL-R 验证框架对目标操作系统进行验证. 下面就相关基础知识予以介绍.

2.1 交互式定理证明器 Coq

Coq[22]是基于名为归纳构造演算 (CIC) 的类型理论而实现的交互式定理证明器. Coq 拥有强大的表达能力, 并提供易用的交互式证明开发环境. Coq 提供了自然数、命题逻辑、列表等, 其中包含大量可在证明任务中复用的基本定理, 在一定程度上减少了证明人员的工作量. 用户可在 Coq 中定义程序语言, 并且证明许多相关性质. 在 Coq 中现阶段已完成多项复杂证明任务, 如四色定理的证明[23]、CompCert[24]编译器的正确性证明等.

Coq 实现了一种被称为 Gallina[22]的高级数学语言, 在基于定理证明的形式化验证任务中, 该语言可对系统模型、目标性质进行描述, 并用于表达系统模型满足目标性质的定理及其证明. 此外, Coq 提供一系列证明策略以简化证明的构造. 给定某个证明目标后, Coq 可根据用户提供的证明策略或已经证明的定理将当前目标分解成许多更简单的子目标, 通过证明这些子目标可以完成原目标的证明. 除了 Coq 中原有的证明策略, 使用者还可以借助

Coq 中的 Ltac 命令定义新的自动证明策略, 此命令可以组合策略并指定其执行顺序, 还能根据目标的不同形式执行对应的策略, 这样使得证明过程更加简洁, 证明代码更易于维护.

2.2 互斥量运行机制

互斥量可用来保护支持任务互斥式访问的共享资源. 当存在多个任务要访问同一共享资源时, 只有互斥量的占有者才能访问共享资源, 即任务访问共享资源前需先获得互斥量, 任务结束访问资源时需释放互斥量.

当任务 A 要访问共享资源时, 此时就需请求获取互斥量. 若互斥量是可用的, 任务 A 获取互斥量成功, 该任务能直接访问共享资源. 若互斥量已被其他任务占用, 任务 A 就会被阻塞, 此时可分为两种情况: (1) 任务 A 一直被阻塞, 直到互斥量被占有者任务释放, 任务 A 才能成功获取互斥量. (2) 任务 A 等待超时, 此任务未成功获取互斥量, 即任务 A 不能访问共享资源. 释放互斥量的任务和获取此互斥量的任务必须是同一个任务. 当任务 A 对共享资源访问结束时, 应释放互斥量.

若互斥量支持嵌套, 任务可多次获取或释放此互斥量, 且重复获取互斥量的次数等于释放互斥量的次数. 当任务 B 想要获取的互斥量的嵌套数处于合法范围内时, 此任务可成功获取互斥量. 当任务 B 要释放互斥量, 需要满足释放互斥量的任务 B 是占用该互斥量的任务才可释放互斥量.

以上互斥量运行机制可确保同一时刻只有一个任务正在访问共享资源.

2.3 精化程序逻辑 CSL-R

并发精化分离逻辑 CSL-R[2]支持抢占式操作系统的精化验证. 抢占式操作系统的功能正确性可通过其接口函数的形式化规范进行描述. 每个接口函数的形式化规范可描述为一段抽象程序, 使得具体程序是抽象程序的精化 (refinement), 即具体层执行行为的集合是抽象层执行行为集合的子集, 或具体层执行可被抽象层执行模拟. 接口函数的形式化规范 (抽象程序) 相对简单, 容易看出其是否满足接口设计的功能需求. 对于正确性而言, 形式化规范对功能需求的满足往往可以由精化关系传递到具体代码对功能需求的满足.

抽象程序由定制的程序语言编写, 支持原子操作、顺序组合、非确定性选择等结构, 以及任务调度 (切换) 等原语. 抽象程序中往往忽略部分实现细节, 如内存的使用、与需求无关的控制条件、加工局部数据的中间计算步骤等. 抽象程序语言的定义依赖于系统抽象状态的定义, 对于 μC/OS-II 操作系统, 抽象状态 Σ 包括任务状态、事件状态、当前任务等组成部分:

$$\Sigma ::= \{ecbls \to \beta, ctid \to t, tcbls \to \alpha\},$$

其中, *ecbls* 为抽象状态中事件状态这一部分内容的标识, 映射到事件状态 β, β 反映 μC/OS-II 中所有活跃事件对象 (具体实现为事件控制块, ECB) 所处状态; *ctid* 为抽象状态中当前任务这一部分内容的标识, 映射到当前任务的地址 *t*; *tcbls* 为抽象状态中任务状态这一部分内容的标识, 映射到任务状态 α, α 反映 μC/OS-II 中所有被创建出的任务 (具体实现为任务控制块, TCB) 所处状态.

在 μC/OS-II 内核中, 以关中断方式实现临界区. 在 CSL-R 中, 通过全局不变式描述全局资源在进出临界区时所满足的条件. 这些条件包括数据结构的良构性、数据值所满足的约束、具体状态和抽象状态间的关联等. 一方面, 全局不变式中的部分条件可视为系统执行过程中保持成立的性质. 另一方面, 全局不变式为验证提供关于全局资源状态的关键辅助信息.

CSL-R 中关于程序语句的逻辑判断形如:

$$\Gamma;\chi;I;\rho;p_i|-\{p\}\,s\,\{q\},$$

其中, Γ 给出系统中每个函数的规范, χ 为调度策略, *I* 为全局不变式, *p* 为前断言, ρ 为关于函数返回的后断言, p_i 和 *q* 分别为关于中断返回和顺序执行终止的后断言. 在验证 API 函数时, p_i 和 *q* 均为 false, 表示程序退出方式不是中断返回或顺序终止 (而是执行返回语句).

在程序语句 *s* 逻辑判断的具体实例中, 前条件 *p* 常包含形如<||*s*; *s*′||> 的断言, 其中的抽象程序语句 *s* 对具体程序语句 *s* 进行描述, 而抽象程序语句 *s*′对 *s* 完成后, 仍有待执行的具体程序语句进行描述. 相应地, 后条件 *q* 中包含<||*s*′||>, 表示在具体程序语句 *s* 结束后, 仍有待执行的具体程序由抽象程序 *s*′描述.

对每个临界区进行验证的典型模式如下. 在进入每个临界区时, 借助逻辑规则在断言中引入全局不变式条件, 表示全局共享资源良构且可以开始被当前任务独占使用. 对于临界区中的程序语句, 借助逻辑规则由其前断言推出后断言. 适时执行逻辑断言中<||…||> 所包含的抽象程序, 相应更新抽象状态, 以确保退出临界区时, 全局共享资源仍然良构, 全局不变式条件可再次被建立. 在退出临界区时, 借助相应逻辑规则将全局不变式条件从断言中去除, 表示当前任务保障全局共享资源依然良构, 进而将这部分资源的所有权移交出去.

CSL-R 验证框架中假设操作系统内核函数分为 API 函数、内部函数、中断处理程序这 3 类, 其中 API 函数和内部函数均可调用内部函数, 但 API 函数和内部函数均不能调用 API 函数. 对于 API 函数, 框架构造形如 OS[…]∗ $x| \to v*$ < ||aop(v,…)|| > 的前断言, 其中 OS[…] 反映临界区、中断相关状态, $x| \to v$ 是函数参数 x 的分离逻辑断言, v 为 x 对应的实参值, aop(v,…) 为函数对应的抽象程序 (即函数的规范), 抽象程序接收实参 v 作为输入, 符号*将 3 种逻辑断言进行分离, 表示将程序状态分为 3 个不相交的程序状态后可分别满足 3 种逻辑断言.

对 API 函数的验证从上述前断言开始, 对函数体语句进行前向推理 (forward reasoning), 若推理进行至函数末尾时, 逻辑断言中的抽象程序已无剩余, 且函数的后断言得到满足, 则函数通过验证. 在语义层面, 这保证函数代码为函数前断言中抽象程序 aop(v,…) 的精化.

3 验证框架扩展

操作系统内核正常运行有时依赖于用户调用内核函数时遵循一定规则. 如用户向内核函数传递一个指向某结构体的指针时应保障, 该指针若不为 NULL, 则确实指向良构的结构体实例 (用户已经为其分配内存). 相应地, 操作系统内核的验证有时需要假设用户对内核的使用方式满足一定条件.

在 μC/OS-II 验证框架中, API 函数形如 OS[…] * $x| \to v$ * <||aop(v, …)||> 的前断言 (见第 2 节) 尚无法支持此类假设条件的表达. 为解决此问题, 本文在原验证框架的高层规范语言中加入用来表达假设条件的特性, 允许在抽象程序 aop(v, …) 中对 API 函数正确工作所依赖的用户假设进行表达, 从而以假设条件成立为前提, 对操作系统的各个互斥量接口函数进行验证. 以下简要说明这一扩展所涉及的关键技术环节.

3.1 语法和语义

首先, 我们在高层规范语言中引入断言语句 assert 和中止语句 abort, 体现为 Coq 归纳定义的构造子 spec_assert 和 spec_abort, 如下所示.

```
Inductive spec_code :=
| spec_assert : absexpr → spec_code
| spec_abort : spec_code
    …
```

其中, spec_code 为归纳构造的类型, 表示抽象程序语句. 构造子 spec_assert 用于构造 assert 语句——spec_assert e 假设表达式 e 所表达的假设条件得到满足. 若假设条件不满足, 则抽象程序的执行将产生由 spec_abort 构造的中止语句.

此外, 我们修改了高层规范语言中原子操作的类型. 原子操作原本为全函数, 其在某个抽象状态 osabst 下执行, 必得到新的抽象状态. 而调整后原子操作为偏函数, 其结果为可选的抽象状态 (option osabst)——若原子操作结果为 None, 表示未能产生结果状态, 则抽象程序的执行亦将产生由 spec_abort 构造的中止语句, 表示原子操作中所表达的假设条件未得到满足.

上述的执行行为通过抽象程序语句的语义规则进行描述.

```
Inductive spec_step: ossched → spec_code → osabst → spec_code → osabst → Prop :=
| spec_prim_step_abt:
forall sc O (step:osabstep) v Of vl OO,
```

```
  step vl O (v, None) →
  join O Of OO →
  spec_step sc (spec_prim vl step) OO spec_abort OO
…
| spec_assert_step_abt:
  forall O Of (b:absexpr) OO sc,
  ~(b O) →
  join O Of OO →
  spec_step sc (spec_assert b) OO (spec_abort) OO
…
```

其中, spec_prim_step_abt 在原子操作结果为 None 时构造产生 spec_abort 的一步执行, spec_assert_step_abt 在 assert 语句中显式假设的条件不满足的情况下构造产生 spec_abort 的一步执行. 定义中 O、Of、OO 为部分抽象状态 (其定义域是抽象状态 Σ 定义域的子集). 表达式 join O Of OO 表示 O、Of 的定义域无重叠部分, OO 的定义域为 O、Of 定义域的并集, 且对于任意 a, 若 a 在 O 定义域中, 则 OO(a)=O(a), 而若 a 在 Of 定义域中, 则 OO(a)=Of(a). 此外, spec_prim vl step 表示以 vl 为输入的原子操作 step, 是抽象程序语句的一种.

3.2 程序逻辑

在第 2.3 节中已经介绍, 在使用 CSL-R 的逻辑规则对程序语句进行推理时, 往往执行抽象程序中与具体代码相对应的部分, 并证明全局不变式条件得到保持. 若抽象程序中通过 assert 语句以及原子操作所表达的假设条件不满足, 则抽象程序的执行产生标志语句 spec_abort. 我们添加逻辑规则, 表达关于 spec_abort 语句的逻辑判断可无条件建立起来. 这样, 若某接口函数的抽象程序 (即其形式化规范) 中所表达的假设条件未得到满足, 则关于其代码的推理中不再产生进一步的证明义务.

程序逻辑的可靠性体现在其保证具体程序对抽象程序的精化条件, 具体反映在具体程序的执行步骤可被抽象程序的执行所模拟. 在 CSL-R 的可靠性证明中, 相应定义了函数级、任务级、系统级的模拟关系. 我们调整这些模拟关系的定义, 表达若抽象程序中所表达的假设条件不满足 (体现在抽象程序可执行到 spec_abort), 那么具体程序进一步的执行步骤无需被抽象程序所模拟. 在抽象程序语法、语义、程序逻辑规则、精化条件调整的基础上, 我们再次证明了程序逻辑的可靠性, 技术工作的细节不再赘述.

4 互斥量模块抽象状态和不变式条件

某微内核操作系统互斥量模块的接口函数作为 CSL-R 验证框架中的 API 函数进行处理. 这些函数形式化规范的主体是函数的抽象程序. 这些抽象程序含义的表达依赖于互斥量模块数据状态在抽象状态中的表示. 相应地, 需在全局不变式中引入互斥量模块抽象数据状态与具体状态的关联、互斥量模块抽象数据状态与 μC/OS-II 内部事件对象状态的关联. 我们也在全局不变式中引入其他条件, 以反映互斥量模块执行保持的性质.

4.1 互斥量模块抽象状态

引入互斥量模块数据状态后的抽象状态 Σ 定义如下所示.

$$
\begin{cases}
\Sigma ::= \{mls \to \rho, ecbls \to \beta, ctid \to t, tcbls \to \alpha, muxhp \to \mu\} \\
\rho ::= \{m_1 \to (eid_1, ow_1, pip_1, tp_1, cnt_1), \ldots, m_n \to (eid_n, ow_n, pip_n, tp_n, cnt_n)\} \\
\mu ::= \{(a_1, typ_1, vl_1), \ldots, (a_n, typ_n, vl_n)\} \\
\beta ::= \{eid_1 \to ed_1, \ldots, eid_n \to ed_n\} \\
ed ::= mutex(pr, w, q) | ... \\
\alpha ::= \{t_1 \to (p_1, ts_1, msg_1), \ldots, t_n \to (p_n, ts_n, msg_n)\} \\
w ::= \bot | (tk, pt) \\
vl ::= nil | v :: vl.
\end{cases}
$$

以上定义中, *mls* 是抽象状态中互斥量状态这一部分的名称. 互斥量状态以 ρ 表示, 具体为互斥量索引号 (即为 C 代码中互斥量结构体数组下标) 到互斥量抽象数据结构的映射. 在互斥量抽象数据结构 (*eid*, *ow*, *pip*, *tp*, *cnt*) 中, *eid* 表示 μC/OS-II 内部互斥量的事件标识符, *ow* 表示互斥量占有者, *pip* 表示互斥量置顶优先级, *tp* 表示互斥量是否为嵌套型, *cnt* 表示互斥量的当前嵌套数.

此外, 事件标识符 *eid* 到 μC/OS-II 中事件对象的抽象表示 *ed* 的映射由 β 表示, *ed* 可以为互斥量或 μC/OS-II 中其他类型的事件对象. 对于互斥量 mutex(*pr*, *w*, *q*), *pr* 为 μC/OS-II 中所记录的互斥量置顶优先级, *w* 为 μC/OS-II 中所记录的互斥量占有者, *q* 为 μC/OS-II 中维护的互斥量等待任务列表. 其中, *w* 为 $\perp$ 时表示互斥量无占有者, *w* 为 (*tk*, *pt*) 时表示互斥量当前占有者为任务 *tk*, 且该任务优先级为 *pt*.

以上定义中, 任务状态 α 是由任务标识符 *t* 到任务控制块抽象表示的映射. 任务控制块的抽象表示包括任务的优先级 *p*、当前状态 *ts* 和消息域 *msg*, 任务状态包括就绪态和等待态.

最后, 上述定义中 *muxhp*、μ 等要素将在第 4.3 节详细介绍.

4.2 互斥量模块不变式条件

某微内核操作系统互斥量数据结构中包含 μC/OS-II 互斥量事件的标识符 eid, 系统工作过程中确实保证存在由 *eid* 所标识的 μC/OS-II 互斥量. 基于抽象状态, 可相应给出下列不变式条件.

$$\lambda\Sigma.\forall ecbls, mls.\Sigma(ecbls)=\beta \rightarrow \Sigma(mls)=\rho \rightarrow \\ \forall p, eid, ow, pip, tp, cnt.\ \rho(p)=(eid, ow, pip, tp, cnt) \rightarrow (\exists pr, w, q.\beta(eid)=\mathrm{mutex}(pr, w, q)).$$

其中, $\Sigma(ecbls)$ 表示获得事件状态 β, $\Sigma(mls)$ 用以获取互斥量状态 ρ. 在 β 中可通过表示互斥量的序号 *p* 得到互斥量抽象数据结构 (*eid*, *ow*, *pip*, *tp*, *cnt*), 其中互斥量标识符 *eid* 可找到其对应的事件对象 mutex(*pr*, *w*, *q*).

此外, 在不变式中定义了互斥量数据结构抽象层与具体层的一致性关系, 即互斥量抽象数据结构中各成员的值与具体层相应成员的内存值是相同的. 这里不再详细介绍.

在系统运行过程中, 全局资源能够始终保持一些性质. 互斥量模块的关键全局资源包括互斥量状态 (如是否被占有、优先级等)、互斥量占有者状态等内容, 本文在全局不变式中引入逻辑断言, 对这些内容所满足的性质进行描述, 以下是对部分性质的举例.

性质 1: 占有某个互斥量的任务不会在此互斥量的等待列表中. 性质 1 具体形式化描述如下.

$$\lambda\Sigma.\forall ecbls.\Sigma(ecbls)=\beta \rightarrow \\ \forall eid, pr, tk, pt, q.\beta(eid)=\mathrm{mutex}(pr, (tk, pt), q) \rightarrow not\_in\ tk\ q.$$

在上述描述中, $\Sigma(ecbls)$ 表示获取事件状态 β, 其中互斥量标识符 *eid* 所对应的互斥量事件对象为 mutex(*pr*, (*tk*, *pt*), *q*), *tk* 是占用互斥量的任务标识符, *not_in tk q* 表示占有者任务 *tk* 不在等待互斥量的任务列表 *q* 中.

性质 2: 当互斥量可用时, 任何任务优先级都不是该互斥量置顶优先级. 性质 2 具体形式化描述如下.

$$\lambda\Sigma.\forall ecbls, tcbls.\Sigma(ecbls)=\beta \rightarrow \Sigma(tcbls)=\alpha \rightarrow \\ \forall eid, pr, q, t, p, ts, msg.\beta(eid)=\mathrm{mutex}(pr, \perp, q) \rightarrow \alpha(t)=(p, ts, msg) \rightarrow p \neq pr.$$

在上述形式化描述中, 事件状态 β 中 *eid* 对应的互斥量对象记录了互斥量置顶优先级 *pr*、占有者信息为空 ($\perp$)、互斥量等待任务列表 *q*. $\Sigma(tcbls)$ 表示从抽象状态中获取任务状态 α, 其中任务标识符 *t* 对应的任务对象中优先级为 *p*, $p \neq pr$ 用以表示任务优先级不是互斥量的置顶优先级.

性质 3: 当互斥量被任务占用时, 该任务不在互斥量等待列表中, 如果此时互斥量的占有者任务优先级不是互斥量的置顶优先级, 则任意任务的优先级都不是该互斥量的置顶优先级. 性质 3 具体形式化描述如下.

$$\lambda\Sigma.\forall ecbls, tcbls.\Sigma(ecbls)=\beta \rightarrow \Sigma(tcbls)=\alpha \rightarrow \\ \forall eid, pr, q, tk, pt, p, ts, msg.\beta(eid)=\mathrm{mutex}(pr, (tk, pt), q) \rightarrow \alpha(tk)=(p, ts, msg) \rightarrow not\_in\ tk\ q \rightarrow p \neq pr \rightarrow \\ (\forall t', p', ts', msg'.\alpha(t')=(p', ts', msg') \rightarrow p' \neq pr).$$

上述描述中, 事件标识符 *eid* 对应互斥量对象记录互斥量置顶优先级 *pr*、占有者任务标识符 *tk*、互斥量等待任务列表 *q*. 任务标识符 *tk* 对应任务对象中优先级为 *p*, 且该优先级不是置顶优先级 *pr*, *not_in tk q* 表示任务 *tk* 不

在互斥量等待列表中. p'表示任意任务标识符 t'对应任务优先级, 此优先级不是互斥量置顶优先级.

性质 4: 任意两个不同的互斥量具有不同的置顶优先级. 性质 4 具体形式化描述如下.

$$\lambda\Sigma.\forall ecbls.\Sigma(ecbls)=\beta\rightarrow$$
$$\forall eid,eid',w,w',pr,pr',q,q'.\beta(eid)=\text{mutex}(pr,w,q)\rightarrow\beta(eid')=\text{mutex}(pr',w',q')\rightarrow$$
$$eid\neq eid'\rightarrow pr\neq pr'.$$

上述形式化描述中, 在事件状态 β 中, 通过任意两个不同的事件标识符 (*eid* 与 *eid′*) 可分别找到对应的互斥量对象, 两者中的互斥量置顶优先级 *pr* 与 *pr′*是不一样的.

本节所述的 4 条性质侧重于互斥量数据结构的良构性, 它们的另外一个重要作用是在各个互斥量函数的验证中提供必要的辅助信息 (类似于循环不变式所起作用). 需要指出的是, 本文主要处理微内核操作系统函数代码级验证问题, 并未涉及互斥量的一些典型总体性质的验证. 其中一类总体性质是无优先级反转 (PIF)——该性质可基于本工作提供的互斥量接口函数的抽象程序加以验证. 此外, 用户程序以不当方式使用互斥量接口可导致死锁, 本文的验证工作覆盖互斥量接口的实现, 但不覆盖使用互斥量的用户程序, 因此不涉及无死锁的验证. 验证工作所基于的 CSL-R 程序逻辑也并不建立保持活性性质的精化关系.

4.3 互斥量模块堆结构

在所验证的抢占式微内核操作系统中, 借助描述互斥量属性信息的结构体对互斥量属性进行设置. 用户为此类结构体分配内存, 并将结构体实例的指针传入内核函数, 使得后者可以借助结构体成员的数据完成互斥量属性的设置. 在此前 μC/OS-II 的验证中, 用户传入内核的数据结构往往近似建模为内核中分配和释放的数据结构. 本工作中, 借助单独定义的堆结构对用户传入内核的数据结构进行准确描述.

堆结构表达为一系列三元组 (*a*, *typ*, *vl*) 构成的列表, 表示在内存位置 *a* 处存在类型为 *typ* 且成员值在值表 *vl* 中的结构体实例——如描述互斥量属性信息的结构体实例, 其中, *vl* 为 *nil* 表示空值表, *vl* 为 *v*::*nil* 表示值表非空. 将此种堆结构引入抽象状态中 (见第 4.1 节抽象状态定义中的 μ). 此外, 在全局不变式中引入堆中每个结构体的良构性断言. 对于互斥量模块中以结构体指针为参数的接口函数, 在其抽象程序 (即形式化规范) 中, 表达“结构体指针的值是堆中某个内存位置”的假设条件, 以反映用户传入的指针确实指向已为其分配内存, 类型正确的结构体实例. 此外, 通过抽象程序中对堆中某个三元组的读、写操作反映接口函数中关于具体结构体成员的读写操作.

5 接口函数的形式化规范与证明

本节以函数 pthread_mutex_lock 为例介绍互斥量接口函数形式化规范的建立和功能正确性的验证. 函数 pthread_mutex_lock 用于获取互斥量, 通常在用户应用需要访问全局资源前进行调用. 本节的介绍中, 略去了 C 代码的部分细节, 以凸显接口函数代码的总体控制流, 以及接口函数与 μC/OS-II 中对应函数的调用关系.

5.1 接口函数 pthread_mutex_lock 的主要功能与代码形式化

函数 pthread_mutex_lock 的主要功能是获取其形参 *p* 所表示的互斥量, 并返回操作是否成功. 具体而言, *p* 用作系统全局的互斥量结构体数组 Mux_struct 中的下标, 引用该数组中一个具体的互斥量结构体 Mux_struct[*p*]. 互斥量结构体中, pevent 成员为指向 μC/OS-II 中事件控制块 (ECB, 表示 μC/OS-II 中的事件对象) 的指针.

函数代码 (如算法 1) 首先检查 Mux_struct[*p*].pevent 是否为空指针, 若是则返回 ERR, 表示互斥量获取操作失败. 若 Mux_struct[*p*].pevent 不是空指针, 则检查互斥量是否支持嵌套, 若互斥量类型是嵌套型 (Mux_struct[*p*].tp == recursv), 则在当前任务是互斥量占有者并且互斥量嵌套数在合法范围内前提下, 将互斥量嵌套数自增. 而后, 调用 μC/OS-II 的函数 OSMutexPend, 尝试设置 ECB 中互斥量占有者为当前任务. 若 OSMutexPend 返回 PEND_OK, 表示获取互斥量成功, 则 pthread_mutex_lock 返回 NO_ERR; 否则 pthread_mutex_lock 返回 ERR.

算法 1. 函数 pthread_mutex_lock 代码.

```
int pthread_mutex_lock (int32u p)
```

```
{
1     int8u z;
2     if (Mux_struct[p].pevent == NULL){
3         return ERR;
4     }
5     if (Mux_struct[p].tp == recursv){
6         /*若当前任务是互斥量占有者且互斥量嵌套数在合法范围内, 则嵌套数自增*/
7     }
8     z = OSMutexPend (Mux_struct[p].pevent, 0);
9     if (z == PEND_OK){
10        return NO_ERR;
11    }
12    return ERR;
}
```

在 C 代码中判断互斥量事件控制块指针 Mux_struct[p].pevent 是否为空指针是在临界区之外进行的, 在对该条件判断进行推理时, 无法获取全局共享资源 Mux_struct[p] 的良构性. 使用小临界区技术解决该问题. 具体而言, 该条件判断形式化如下. 其中, 符号“$;_s$”与“$==_e$”均为 Coq 中的一种记法 (notation), “$;_s$”表示一句程序语句的结束, “$==_e$”用以比较符号两端表达式取值是否相等.

```
ENTER_CRITICAL;s
pevt′=e efield (Mutex_struct′[p′]) pevent;s
EXIT_CRITICAL;s
If (pevt′==e NULL) {
RETURN ′ERR
};s
```

这里将从互斥量结构体中读取 ECB 指针的操作置于人为添加的临界区中, 所读取的值赋给专门引入的局部变量 *pevt*. 由于 Mutex_struct 本身的值不会被任何任务所修改, 形参 p 为局部变量, 因而这一做法不会改变程序的语义. 而在验证中, 添加临界区允许在对读取互斥量结构体中 ECB 指针的操作进行推理时, 获取 Mux_struct[p] 的良构性条件. 类似地, 互斥量模块代码中其他在临界区外访问全局状态的操作亦在形式化过程中使用小临界区技术进行了处理.

需要指出上述形式化代码片段为在 Coq 中关于 C 语言代码的建模, 使用 Coq 中所定义的 C 语言子集语法 (以及相应定义的 notation) 写成, 其外观与 C 语言语法略有出入, 但不难由形式化代码读出其所对应的 C 语言代码.

5.2 接口函数 pthread_mutex_lock 的形式化规范

以下以接口函数 pthread_mutex_lock 其中一种成功情形和失败情形为例, 介绍为其建立的形式化规范.

(1) 定义 $\omega_{\text{muxlk\_null}}$ $\gamma_{\text{muxacs\_null}}$ 表示失败情形: 如果指定互斥量的指针为空指针, 返回 ERR.

$$\gamma_{\text{muxlk\_getptr}}(vl) \stackrel{\text{def}}{=} \lambda\Sigma,(v,\Sigma').\ \exists mls,p,ow,pip,tp,cnt,pevent.$$
$$vl=(p::pevent::nil)\wedge\Sigma(mls)=\rho\wedge\rho(p)=(pevent,ow,pip,tp,cnt)\wedge\Sigma=\Sigma'.$$

$$\gamma_{\text{muxlk\_eqnull}}(vl) \stackrel{\text{def}}{=} \lambda\Sigma,(v,\Sigma').\ \exists pevent.$$
$$vl=(pevent::nil)\wedge pevent=V\text{null}\wedge v=\text{ERR}\wedge\Sigma=\Sigma'.$$

$$\omega_{\text{muxlk\_null}}(p::pevent::nil) \stackrel{\text{def}}{=} \lambda_{\text{muxlk\_getptr}}(p::pevent::nil);;\lambda_{\text{muxlk\_eqnull}}(pevent::nil).$$

其中, 原子操作 $\gamma_{\text{muxlk\_getptr}}$ 描述在小临界区内获得互斥量抽象数据结构中的互斥量事件控制块指针. Σ 表示该操作开始执行时的抽象状态, (v, Σ') 中的 v 表示函数的返回值, Σ'表示执行该操作后得到的抽象状态. vl 表示抽象操作的参数表. 参数 p 表示互斥量结构体数组下标, $\Sigma(mls)$ 表示在抽象状态中获得互斥量状态 ρ. $\rho(p)$ 表示通过互斥量映射关系获得 p 对应的数据结构信息. 此情形中参数 *pevent* 在 $\gamma_{\text{muxlk\_getptr}}$ 中作为输出参数, 并在 $\gamma_{\text{muxlk\_eqnull}}$ 中作为输入参数, 以判断其是否为空指针.

以上, ;; 表示抽象程序语句的顺序组合. 获取互斥量结构体中 ECB 指针并判断指针为空的操作分为顺序执行的两个原子操作进行表示. 这反映了任务执行 pthread_mutex_lock 函数代码过程中, 在获取互斥量结构体中 ECB 指针后, 判断该指针为空前, 可能被抢占, 而抢占任务可对状态进行修改. 因而抽象程序如实反映并发环境中程序的执行方式. 为方便理解, 以下参数均表示同一含义, *pevent* 表示获取的互斥量事件控制块指针, *ow* 表示互斥量占有者, *pip* 表示互斥量置顶优先级, *tp* 用以表示互斥量是否支持嵌套, *cnt* 表示互斥量嵌套数.

(2) 定义 $\omega_{\text{muxlk\_norc\_pendok}}$ 表示成功情形: 如果指向互斥量控制块的指针非空, 互斥量不支持嵌套, 调用函数 OSMutexPend, z 的值为 PEND_OK, 返回 NO_ERR, 申请互斥量成功.

$$\gamma_{\text{muxlk\_gettp}}(vl) \overset{\text{def}}{=} \lambda\Sigma,(v,\Sigma').\ \exists mls,p,pevent,ow,pip,tp,cnt,eid.$$
$$vl=(p::pevent::tp::nil)\wedge\Sigma(mls)=\rho\wedge$$
$$pevent\neq V\text{null}\wedge$$
$$\rho(p)=(eid,ow,pip,tp,cnt)\wedge\Sigma=\Sigma'.$$

$$\gamma_{\text{muxlk\_norc}}(vl) \overset{\text{def}}{=} \lambda\Sigma,(v,\Sigma').\exists tp.$$
$$vl=(tp::nil)\wedge tp\neq recursv\wedge\Sigma=\Sigma'.$$

$$\gamma_{\text{afterpend\_ok}}(vl) \overset{\text{def}}{=} \lambda\Sigma,(v,\Sigma').\exists pend\_ret.$$
$$vl=(pend\_ret::nil)\wedge pend\_ret=\text{PEND\_OK}\wedge v=\text{NO\_ERR}\wedge\Sigma=\Sigma'.$$

$$\omega_{\text{muxlk\_norc\_pendok}}(p::pevent::tp::pevent1::rv::nil) \overset{\text{def}}{=} \lambda_{\text{muxlk\_gettp}}(p::pevent::tp::nil);;\gamma_{\text{muxlk\_norc}}(tp::nil);;$$
$$\gamma_{\text{muxlk\_getptr}}(p::pevent1::nil);;$$
$$\gamma_{\text{mutexpend}}(pevent1::0::rv::nil);;\gamma_{\text{afterpend\_ok}}(rv::nil).$$

其中, $\gamma_{\text{mutexpend}}$ 表示函数 OSMutexPend 在 μC/OS-II 中定义的形式化规范, 其在接口函数 pthread_mutex_lock 形式化规范中复用. 在调用函数 OSMutexPend 之前, 先在小临界区中获取参数互斥量事件控制块指针的值 pevent1, 然后将其获取的值传入函数 OSMutexPend 中. 此情形首在小临界区中用 $\gamma_{\text{muxlk\_gettp}}$ 获取互斥量抽象数据结构中的类型 *tp*, 其中的 *tp* 作为输出参数. 而后, 将 *tp* 作为输入参数传入 $\gamma_{\text{muxlk\_norc}}$ 中, *tp* 的值不是嵌套型 recursv. 之后, $\gamma_{\text{muxlk\_getptr}}$ 在小临界区中获取互斥量事件控制块指针 pevent1, 这里定义 $\gamma_{\text{muxlk\_getptr}}$ 可以复用. 在调用函数 OSMutexPend 后, 将函数 OSMutexPend 的返回值作为 $\gamma_{\text{afterpend\_ok}}$ 的输入参数, 此时返回值为 PEND_OK , 接口函数返回 NO_ERR.

5.3 接口函数 pthread_mutex_lock 的形式化证明

本节介绍接口函数 pthread_mutex_lock 形式化证明的关键内容. 该函数正确性定理的表述中, 基于其抽象程序 mutex_lock 构造该函数的前断言 muxlkpre 和后断言 muxlkpost.

$$\text{muxlkpre} \overset{\text{def}}{=} \exists rv,v,vl.\text{OS}[\bar{0},1,nil,nil]*z\overset{\text{Tint8}}{\mapsto}{}_1rv*p\overset{\text{Tint32}}{\mapsto}v*<\|\text{mutex\_lock}(vl)\|>.$$
$$\text{muxlkpost} \overset{\text{def}}{=} \lambda\hat{v}.\exists rv,v.\text{OS}[\bar{0},1,nil,nil]*z\overset{\text{Tint8}}{\mapsto}{}_1rv*p\overset{\text{Tint32}}{\mapsto}{}_1v*<\|\text{END}\hat{v}\|>.$$

表达式 $\text{OS}[\bar{0},1,nil,nil]$ 表示没有中断正在被服务、中断使能开关处于打开状态、当前程序不在任何临界区内执行、当前程序不在任何中断处理程序中执行, 这反映接口函数 pthread_mutex_lock 由用户程序调用执行. 表达式 $z\overset{\text{Tint8}}{\mapsto}{}_1rv$ 表示局部变量 z 类型为 int8, 在内存中的值为 *rv*. 表达式 $p\overset{\text{Tint32}}{\rightarrow}{}_1v$ 表示局部变量 p 类型为 int32, 在内存中的值为 v. 后断言中表达式 $\text{END}\hat{v}$ 表示系统 API 函数执行结束, 返回值为 $\hat{v}$.

函数 pthread_mutex_lock 的证明过程中用到顺序组合语句、分支语句、进出临界区语句、赋值语句、函数调用语句、函数返回语句等的推理规则, 以及具体层、抽象层的包容规则 (subsumption rule). 其中具体层包容规

则允许强化语句的前条件、弱化后条件. 抽象层包容规则将断言中抽象程序的执行视为对断言的弱化.

关于函数调用 OSMutexPend (Mux_struct[p].pevent, 0) 进行推理时, 用到函数 OSMutexPend 的形式化规范——证明调用前程序点的逻辑断言蕴含 OSMutexPend 的前条件, 并在调用后程序点处得到该函数的后条件, 以之作为该位置处的逻辑断言. 函数 OSMutexPend 为 μC/OS-II 中的 API 函数, 而在微内核操作系统的验证中, 其地位变为内部函数. 在验证工作中需对 OSMutexPend 此前作为 API 函数的形式化规范进行一定调整, 使其符合 CSL-R 验证框架中内部函数规范的形式, 因而对原 OSMutexPend 函数的证明代码同样需要进行一定调整. 总体上, 对于类似 OSMutexPend, 被微内核操作系统内部调用的 μC/OS-II 中 API 函数, 其形式化规范的主体内容保持不变, 且其证明代码的总体框架和许多关键环节仍可沿用原证明代码中对应要素. 形式化规范、证明代码中可被复用的成分在相当程度上缩短了完成微内核操作系统互斥量模块形式化验证所需的时间.

6 验证代码量、发现的问题、经验与教训

某微内核操作系统互斥量模块接口函数包括 pthread_mutex_trylock、pthread_mutex_timedlok、pthread_mutex_lock 等 12 个函数, 本文依次对这些接口函数的功能正确性进行验证, 并统计了各函数的待验证代码行数和 Coq 验证代码行数, 具体如表 1 所示.

表 1 Coq 中的验证代码行数

函数名	被验证代码行数	验证代码行数
pthread_mutex_trylock	96	2457
pthread_mutex_timedlock	106	2765
pthread_mutex_lock	99	2514
pthread_mutex_unlock	97	2563
pthread_mutex_init	76	1 986
pthread_mutex_destroy	33	605
pthread_mutexattr_init	60	1150
pthread_mutexattr_destroy	21	506
pthread_mutexattr_gettype	36	654
pthread_mutexattr_settype	38	705
pthread_mutexattr_getprioceiling	38	726
pthread_mutexattr_setprioceiling	39	732

互斥量模块接口函数在 Coq 中证明总行数达到大约 17000 行. 平均而言, 一行程序代码大约需要 24 行 Coq 代码进行验证, 这与 seL4、μC/OS-II 等操作系统验证工作中的情况相当.

在验证互斥量接口函数过程中, 我们发现了代码中存在的两类问题.

(1) 多任务访问全局资源导致事件对象状态不一致的问题: 当存在两个任务针对同一互斥量执行删除函数, 且允许并发任务在此过程中创建其他种类事件 (例如, 消息队列) 时, 可能创建出包含其他种类 ECB (如消息队列 ECB) 指针的互斥量数据结构.

(2) 由于未在同一临界区中对事件指针进行是否为空的检查, 导致空指针异常: 互斥量的某些接口函数中, 在某个临界区中检查互斥量结构体中指向 μC/OS-II 互斥量 ECB 的指针不为 NULL, 而在其后的另一个临界区中再次获取互斥量结构体中 ECB 指针, 并通过该指针访问 ECB 中的成员, 但并未再次检查该 ECB 指针不为 NULL. 在两个临界区之间, 抢占任务可能将互斥量结构体中的 ECB 指针置为 NULL.

针对上述问题, 对微内核操作系统的代码进行了修改, 并且完成了修改后代码的验证. 具体修改如下.

1) 将 C 代码中两操作位置互换, 即将对互斥量事件指针的置空操作与删除互斥量事件操作位置调换, 以确保上述问题 (1) 的情形不会出现.

2) 在问题 (2) 中描述的后一个临界区中添加互斥量事件指针是否为空的判断, 以排除在此临界区互斥量事件指针为空指针的情况.

此外, 在本工作开展过程中采取了一系列手段以减轻证明工作的负担, 缩短证明代码编写所需的时间.

● 在验证互斥量模块接口函数的过程中, 常常需要对已编写的证明代码进行修改, 包括前提名称或前提中变量名称的修改. 有时在证明开始位置利用重命名策略 rename 对名称进行预先调整, 使其与一部分已有证明中所使用的名称相一致, 从而避免在该部分已有证明内部大量调整前提与变量的名称.

● 推理过程中时常需要通过位置序号引用某个分离逻辑断言, 如在形如 $p*q*r$ 的断言中通过序号 2 引用 q. 但在某些基础定义被调整后, 证明中所需引用的断言序号常发生变化, 导致在验证工作中频繁需要修改证明中使用的分离逻辑断言序号. 使用 Coq 的 Ltac 命令定义了一系列自动化证明策略, 能够根据断言中构造子或其他子结构的名称, 而非断言序号, 对断言进行引用. 这些策略的使用在一定程度上减轻了证明对于基本定义调整的敏感度.

● 在使用 destruct 策略对一些构造子进行讨论时通常会在各分支遇到相同的目标, 通常将这些目标的证明提前到 destruct 之前以减少 Coq 证明行数.

● 在使用封装进出临界区推理规则的自动化证明策略前, 考虑手动对一些断言进行展开和合并以缩短 Coq 执行证明策略所需的处理时间.

若要采取本文的定理证明技术手段验证目标操作系统进一步的模块, 或者其他同类操作系统, 仍然需要投入较多人力和时间. 验证团队认识到在进一步验证工作中能够帮助节省时间精力的几个要素. 首先, 在本工作中, C 语言代码在 Coq 中的表示是通过手工建模得到的. 虽然在 Coq 中定义了 C 语言语法的记号 (notation), 使得 Coq 中的 C 程序形式化代码与原本的 C 代码非常接近, 但手工形式化建模仍会导致一定时间精力消耗, 并且不利于充分保障准确性. 在 Coq 中, 通过 CompCert 可将 C 代码自动转为形式化模型, 但该模型所依托的 C 语言子集及其形式化不同于 CSL-R 框架中的 C 语言子集及其形式化, 故这个转化过程在本工作中无法直接利用. 在未来工作中我们考虑实现自动化工具, 将操作系统内核的 C 代码翻译为 CSL-R 框架中 C 语言的形式化表示. 其次, 各种内核数据结构 (如互斥量数据结构) 内存断言的生成、抽象状态的定义、具体与抽象状态间关联的表达有许多共通之处, 可以考虑对这些任务的部分或全部进行自动化. 最后, 验证团队的技能和经验是此类验证任务中非常宝贵的资源, 对减少新验证任务所造成的时间消耗具有重要意义.

7 总 结

本文介绍某抢占式微内核操作系统互斥量模块功能正确性的形式化验证工作. 该操作系统通过对 μC/OS-II 的封装和扩展进行实现. 本工作使用 Coq 定理证明器, 依托 CSL-R 分离逻辑验证框架, 对目标操作系统的互斥量接口函数进行了代码级形式化描述, 给出了其形式化规范, 并证明了各个函数的代码是其形式化规范的精化. 为表达用户传入互斥量模块的数据良构这一假设条件, 对验证框架进行了扩展. 此外, 通过 CSL-R 中抽象状态的定义描述了目标操作系统中互斥量事件对象的状态, 并通过不变式条件的定义表达了目标操作系统中互斥量数据结构与 μC/OS-II 中事件控制块的关联、互斥量事件对象与任务的关联、互斥量优先级所满足的性质. 在验证中发现了互斥量模块代码中的两类问题, 并完成了修改版代码的验证. 通过本工作, 为某微内核操作系统的互斥量模块提供了高度可信的正确性保障, 并在一定程度上提升了该模块的代码质量.

致谢 感谢冯新宇老师关于 CSL-R 验证框架所提供的指导.

References:

[1] Wang J, Zhan NJ, Feng XY, Liu ZM. Overview of formal methods. Ruan Jian Xue Bao/Journal of Software, 2019, 30(1): 33–61 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5652.htm [doi: 10.13328/j.cnki.jos.005652]

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

[3] Cohen E, Dahlweid M, Hillebrand M, Leinenbach D, Moskal M, Santen T, Schulte W, Tobies S. VCC: A practical system for verifying concurrent C. In: Berghofer S, Nipkow T, Urban C, Wenzel M, eds. Proc. of the 22nd Int'l Conf. on Theorem Proving in Higher Order

Logics. Munich: Springer, 2009. 23–42. [doi: 10.1007/978-3-642-03359-9_2]

[4] Baumann C, Beckert B, Blasum H, Bormer T. Formal verification of a microkernel used in dependable software systems. In: Buth B, Rabe G, Seyfarth T, eds. Proc. of the 28th Int'l Conf. on Computer Safety, Reliability, and Security. Hamburg: Springer, 2009. 187–200. [doi: 10.1007/978-3-642-04468-7_16]

[5] Leinenbach D, Santen T. Verifying the Microsoft Hyper-V hypervisor with VCC. In: Cavalcanti A, Dams DR, eds. Proc. of the 2nd World Congress on Formal Methods. Eindhoven: Springer, 2009. 806–809. [doi: 10.1007/978-3-642-05089-3_51]

[6] Klein G, Huuck R, Schlich B. Operating system verification. Journal of Automated Reasoning, 2009, 42(2–4): 123–124. [doi: 10.1007/s10817-009-9126-9]

[7] Klein G, Andronick J, Elphinstone K, Murray T, Sewell T, Kolanski R, Heiser G. Comprehensive formal verification of an OS microkernel. ACM Trans. on Computer Systems, 2014, 32(1): 2. [doi: 10.1145/2560537]

[8] 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]

[9] Gu RH, Koenig J, Ramananandro T, Shao Z, Wu XN, Wang 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: Association for Computing Machinery, 2015. 595–608. [doi: 10.1145/2676726.2676975]

[10] Feng XY, Shao Z. Modular verification of concurrent assembly code with dynamic thread creation and termination. In: Proc. of the 10th ACM SIGPLAN Int'l Conf. on Functional Programming. Tallinn: Association for Computing Machinery, 2005. 254–267. [doi: 10.1145/1086365.1086399]

[11] Feng XY, Shao Z, Vaynberg A, Xiang S, Ni ZZ. Modular verification of assembly code with stack-based control abstractions. In: Proc. of the 27th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Ottawa: Association for Computing Machinery, 2006. 401–414. [doi: 10.1145/1133981.1134028]

[12] Gu RH, Shao Z, Chen H, Wu XN, Kim J, Sjoberg V, Costanzo D. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In: Proc. of the 12th USENIX Conf. on Operating Systems Design and Implementation. Savannah: USENIX Association, 2016. 653–669.

[13] Chen H, Wu XN, Shao Z, Lockerman J, Gu RH. Toward compositional verification of interruptible OS kernels and device drivers. Journal of Automated Reasoning, 2018, 61(1–4): 141–189. [doi: 10.1007/s10817-017-9446-0]

[14] Liu MQ, Rieg L, Shao Z, Gu RH, Costanzo D, Kim J, Yoon M. Virtual timeline: A formal abstraction for verifying preemptive schedulers with temporal isolation. Proc. of the ACM on Programming Languages, 2020, 4: 20. [doi: 10.1145/3371088]

[15] Andronick J, Lewis C, Morgan C. Controlled owicki-gries concurrency: Reasoning about the preemptible eChronos embedded operating system. arXiv:1511.04170, 2015.

[16] Gu HB. Formalization and verification of operating system's global properties [MS. Thesis]. Hefei: University of Science and Technology of China, 2018 (in Chinese with English abstract).

[17] Zhang XR. Freedom of unbounded priority inversion and its verification [MS. Thesis]. Hefei: University of Science and Technology of China, 2021 (in Chinese with English abstract). [doi: 10.27517/d.cnki.gzkju.2021.000955]

[18] Zhao YW, Sanán D, Zhang FY, Liu Y. Reasoning about information flow security of separation kernels with channel-based communication. In: Chechik M, Raskin JF, eds. Proc. of the 22nd Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Eindhoven: Springer, 2016. 791–810. [doi: 10.1007/978-3-662-49674-9_50]

[19] Sun JW, Long X, Zhao YW. A verified capability-based model for information flow security with dynamic policies. IEEE Access, 2018, 6: 16395–16407. [doi: 10.1109/ACCESS.2018.2815766]

[20] Zhao YW, Sanán D, Zhang FY, Liu Y. Refinement-based specification and security analysis of separation kernels. IEEE Trans. on Dependable and Secure Computing, 2019, 16(1): 127–141. [doi: 10.1109/TDSC.2017.2672983]

[21] Ma YW, Zhang QY, Zhao SJ, Wang GH, Li XM, Shi ZP. Formal verification of memory isolation for the TrustZone-based TEE. In: Proc. of the 27th Asia-Pacific Software Engineering Conf. (APSEC). Singapore: IEEE, 2020. 149–158. [doi: 10.1109/APSEC51365.2020.00023]

[22] Bertot Y, Castéran P. Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Berlin: Springer, 2004. 1–472. [doi: 10.1007/978-3-662-07964-5]

[23] Gonthier G. Formal proof—The four-color theorem. Notices of the American Mathematical Society, 2008, 55(11): 1382–1393.

[24] Leroy X. Formal verification of a realistic compiler. Communications of the ACM, 2009, 52(7): 107–115. [doi: 10.1145/1538788.1538814]

附中文参考文献:

[1] 王戟, 詹乃军, 冯新宇, 刘志明. 形式化方法概貌. 软件学报, 2019, 30(1): 33–61. http://www.jos.org.cn/1000-9825/5652.htm [doi: 10.13328/j.cnki.jos.005652]

[16] 顾海博. 操作系统全局性质的形式化描述和验证 [硕士学位论文]. 合肥: 中国科学技术大学, 2018.

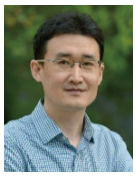
[17] 张啸然. 无上界优先级反转问题的避免及其证明 [硕士学位论文]. 合肥: 中国科学技术大学, 2021. [doi: 10.27517/d.cnki.gzkju.2021.000955]

张林雁(1998－), 女, 硕士生, CCF 学生会员, 主要研究领域为形式化方法, 软件验证.

关永(1966－), 男, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为形式化验证, 高可靠嵌入式系统, 机器人.

李希萌(1987－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为形式化方法, 软件验证.

曹钦翔(1990－), 男, 博士, 副教授, 博士生导师, CCF 专业会员, 主要研究领域为交互式定理证明, 分离逻辑, 程序验证.



施智平(1974－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为形式化方法, 人工智能.

张倩颖(1986－), 女, 博士, 副教授, CCF 专业会员, 主要研究领域为形式化验证, 系统安全, 操作系统.