中国科学技术大学

博士学位论文

抢占式操作系统内核验证框架的设计和实现

作者姓名：许峰唯

学科专业：计算机软件与理论

导师姓名：冯新宇　教授

付明　副研究员

完成时间：

University of Science and Technology of China

A dissertation for doctor's degree

Design and Implementation of A Verification Framework for Preemptive OS Kernels

Author : Fengwei Xu

Speciality : Computer Software and Theory

Supervisor : Prof. Xinyu Feng

A/Prof. Ming Fu

Finished Time :

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文,是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：_____　　签字日期：_____

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

☐ 公开　☐ 保密 _____ 年

作者签名：_____　　导师签名：_____

签字日期：_____　　签字日期：_____

摘 要

计算机系统被广泛应用于国防、通讯、金融等关键领域，构建高可信的计算机系统已成为世界范围的重要课题。操作系统内核作为计算机系统中的底层核心软件，其安全可靠性是构建高可信计算机系统的关键。

防崩溃代码（crash proof code），即用形式化验证技术严格保证底层操作系统的正确性，被 2011 年 MIT 出版的《技术评论》评选为十大新兴技术之一，成为了一个新的研究热点。而操作系统本身的一些特征使得形式化验证变得困难，譬如很多操作系统通常由 C 语言和内嵌汇编实现、代码规模较大、支持多任务并发、中断和抢占等，在这些特征中由于抢占和中断导致内核代码的高度并发使得操作系统的验证变得尤其困难。

操作系统的正确性通常使用应用程序编程接口（API）的具体实现与其高层抽象之间的精化关系来刻画，这使得操作系统正确性验证需要基于程序精化验证技术，而并发系统内核的验证需要同时结合并发验证和精化验证，这使得并发内核的验证变得十分困难。一方面，由于抢占和中断请求导致任务执行的不确定性使得并发内核的验证需要可组合的并发精化验证，而相关理论问题直到最近几年才被解决，因此之前的操作系统内核验证项目都通过避免抢占和中断来简化操作系统内核验证，它们验证的内核代码是串行的。另一方面，现有的可组合的并发精化验证技术和中断验证方法都是基于简化的理论模型，不能直接将其应用于商业化的抢占式操作系统内核验证工作中，需要进行调整和扩展。本文探索如何将可组合的并发精化验证技术和中断验证方法应用于验证商业化的抢占式内核，并做出了如下贡献：

- 本文设计并实现了第一个实用的抢占式操作系统内核验证框架。验证框架以验证操作系统内核功能正确性为目标，这里操作系统内核的功能正确性被定义为应用程序编程接口（API）的实现及其抽象规约之间的上下文精化关系。整个验证框架由三个部分构成：（1）操作系统内核的形式化建模；（2）一个支持多级中断的并发精化程序逻辑 CSL-R 用于验证内核代码的正确性；（3）以及一些自动证明策略来提高验证效率。

- 本文首次完成了一个商业抢占式实时操作系统内核 μC/OS-II [1] 的关键功能模块的正确性验证，包括任务调度程序，时钟中断程序，时钟管理，以及四种同步通信机制：消息队列、互斥锁、消息邮箱和信号量，总计约 3450 行 C 代码，覆盖了 68% 左右的常用 API。同时在高层模型上证明了互斥锁满足无优先级反转的性质 (priority-inversion-freedom， PIF)。值得一提的是，本文验证的是第三方开发的商用内核，而不是为了验证自己开发的内核。

- 本文所描述的操作系统内核验证框架和 μC/OS-II 的验证都已在定理证明工具 Coq [2] 中实现，生成了机器可检查的证明。证明脚本的代码量总计约 225000 行 [3]，这是第一个商业化实时操作系统内核的核心功能模块的机器可检查证明。

关键词： 并发，抢占，操作系统内核，中断，上下文精化，优先级反转

ABSTRACT

Computer systems have been widely used in a lot of key areas, like national defense, financial industry and communication systems. How to build high confidence systems has become an important research topic. As the core underlying software of computer system, the security and reliability of operating system (OS) kernels are very important.

Formal verification is an important way to guarantee that a OS kernel is free of programming errors and has been widely used in a lot of OS kernel verification projects. There are many problems in OS verification, like C inline assembly, large code base, interrupt, preemption and so on. Preemption, on the one hand, is crucial for real-time embedded systems, but on the other hand requires careful implementation of interrupt handlers, synchronization primitives and task scheduling, resulting in highly concurrent and complex kernel code.

Most of the OS kernel verification projects are based on refinement verification, while compositional proof technique for concurrent program refinement remained unproposed until recent years. So concurrency and preemption are avoided in all the projects. Existing compositional proof technique for concurrent program refinement and interrupt verification are based on simplified models, they cannot be used for OS kernel verification directly. This dissertation explores how to apply existing compositional refinement verification technique and interrupt verification technique in preemptive OS kernel verification and makes several contributions.

First, we models the correctness of API implementations in OS kernels as contextual refinement of their abstract specifications. In order to verify the refinement relation, we adapt existing theories on interrupt verification and contextual refinement of concurrent programs, and integrate them into a framework for real-world preemptive OS kernel verification. The framework consists of three parts: (1) modeling of OS kernels; (2) a program logic CSL-R for refinement verification of concurrent kernel code with multi-level hardware interrupts; (3) automated tactics for developing mechanized proofs.

Second, we have successfully applied the framework to verify key modules of a commercial preemptive OS μC/OS-II [1](around 3450 lines of C code with comments and empty lines), including the scheduler, interrupt handlers, message queues, and mutexes *etc.* We also verify the priority-inversion-freedom (PIF) in μC/OS-II. Our work is the first to verify the functional correctness of a practical preemptive OS kernel with machine-checkable proofs. It is worth nothing that, unlike existing works that are all

focused on systems newly developed with verification in mind, we take a commercial system developed by an independent third-party and verify the code with minimum modification, which demonstrates the generality and applicability of our framework.

Finally, all the proofs are mechanized in Coq [2]. The whole proof package is about 225,000 lines of proof script. This is the first mechanized proof of the core modules for a commercial real time OS kernel.

Keywords: Concurrency, Preemptive, OS Kernel, Interrupt, Contextual Refinement, Priority Inversion

目 录

表格索引

插图索引

主要符号对照表

API	应用程序编程接口 (Application-Programming-Interface)
TCB	任务控制块 (Task-Control-Block)
CSL	并发分离逻辑 (Concurrent-Separation-Logic)
RGSim	依赖保证模拟 (Rely-Guarantee Based Simulation)
PI	优先级反转 (Priority-Inversion)
UPI	无边界优先级反转 (Unbounded-Priority-Inversion)
PIF	无优先级反转 (Priority-Inversion-Freedom)
CSL-R	并发精化分离逻辑 (Relational-Concurrent-Separation-Logic)
SMT	特定理论可满足性问题 (Satisfiability-Modulo-Theories)

第一章　绪论

作为信息产业的核心技术，软件系统被广泛地应用于铁路交通、航空航天、核电能源、保健医疗、军事指挥控制等安全攸关的国家战略基础设施和人们的日常生活中。高可信的安全攸关软件系统成了保障国家安全、保持经济可持续发展、维护社会稳定和保护人们生命财产安全的必要条件。不幸的是，软件在很多时候也是我们最缺乏信任感的工程产品。

国内外由于软件缺陷而导致严重的灾难、事故和损失屡见不鲜。例如，1997 年的美国的火星探测器探路者号 (Mars Pathfinder) 的操作系统 Vxworks [4] 由于优先级反转问题引起的故障，导致系统崩溃。2003 年 5 月，由于飞船的导航软件设计缺陷，俄罗斯“联盟－ TMA1”载人飞船返回途中偏离了预定降落地点约 460 公里。2006 年，我国中航信离港系统发生三次软件系统故障，造成近百个机场的登机系统瘫痪。2010 年，“超级蠕虫病毒”Stuxnet [5] 通过攻击西门子 WinCC/PCS 7 SCADA 控制软件，瘫痪了包括伊朗布什尔核电厂在内的大量工业设施。2014 年 4 月发现的 OpenSSL 的 HeartBleed 漏洞 [6] 会导致全球各大主流网站的私有密钥泄露，而导致这个 bug 的主要原因则是缓冲区下标越界的读操作。

如今计算机系统已在国防、通讯、金融、能源、交通、医疗等关键领域中得到广泛应用，构建高可信系统已成为世界范围的重要课题。操作系统内核的安全可靠性是构建高可信计算机系统的关键，因为任何一个微小的内核错误都有可能导致整个计算机系统崩溃。防崩溃代码（crash proof code），即用形式化验证技术严格保证底层操作系统的正确性，被 2011 年 MIT 出版的《技术评论》评选为十大新兴技术之一，成为了一个新的研究热点。此外随着系统软件内核的并行化、复杂化与追求高性能的发展趋势，内核的可靠性显得更加重要。然而现实情况却不容乐观，全球范围内不断发现和公布的内核漏洞与设计缺陷时刻威胁着众多安全攸关的计算机系统。特别是对于实时性要求较高的嵌入式软件系统，其中使用的操作系统大多支持抢占，也即允许高优先级的任务打断低优先级任务的执行，这导致内核高度并发和复杂，其正确性也更难验证。

为了构建安全可靠的操作系统内核，本文基于近年来新提出的并发精化验证理论和技术 [7–12]，设计并实现了一个支持抢占和多级中断的操作系统内核的精化验证框架，并成功应用该框架验证了一个在工业界被广泛应用的商业实时嵌入式操作系统内核 μC/OS-II [1] 的核心模块。

下面本文将在第1.1节介绍操作系统内核验证工作的主要困难和挑战；接着在第1.2节介绍目前操作系统验证工作的研究现状和操作系统验证所基于的并发精化验证理论的研究现状；然后在第1.3节介绍本文研究的主要问题和贡献；最后在第1.4介绍本文的组织结构。

1.1　操作系统验证工作的困难和挑战

作为计算机系统的核心底层软件，操作系统的安全性非常重要，但是操作系统验证面临如下困难和挑战：

操作系统正确性定义　验证操作系统首先需要形式化定义操作系统的正确性。操作系统可以理解为提供给应用层程序员的一些 API 的集合，应用层程序员通过调用这些 API 来控制机器运行。应用层程序员在使用这些 API 的时候并不需要查看和理解具体的实现代码，他们只需要理解这些 API 的规范。操作系统正确性需要刻画出这些 API 的实现和规范之间的一致性，该一致性需要保证任意的应用程序在运行时不会产生程序员无法预料的行为，因此如何定义操作系统正确性来保证系统 API 的实现和其规范之间的一致性是一个首先要解决的难点。

C 语言内嵌汇编　大多数操作系统都是用 C 语言内嵌汇编实现的，C 语言和汇编本身是两种处在不同抽象层次上的编程语言，它们执行所依赖的机器状态不同，如汇编指令需要访问寄存器，而 C 代码则不需要。特别是当操作系统进行任务切换时会改变程序指针寄存器 pc 的值，如何将 pc 寄存器改动体现在 C 代码上是一个非常困难的问题。因此如何对 C 语言内嵌汇编的行为进行形式化建模和验证相应代码是一个需要解决的难点。

丰富的语言特性和较大的代码规模　和很多理论工作不同，操作系统验证针对的是实际的程序语言和机器模型，需要考虑支持很多的语言特性的代码验证（例如：C 语言的函数调用，位运算，强制类型转换等）。同时操作系统内核代码一般规模较大，如何支持模块化验证将巨大的验证任务分解、同时开发自动验证工具提高验证效率使得验证工作能在合理时间内完成也是亟待解决的问题。

中断和抢占导致的复杂并发　操作系统的正确性通常使用底层 API 的具体实现与其高层抽象之间的精化关系来刻画，这使得操作系统正确性验证需要基于程序精化验证技术，而并发系统内核的验证需要同时结合并发验证和精化验证，这使得并发内核的验证变得十分困难。一方面，由于抢占和中断请求导致任务执行的不确定性使得并发内核的验证需要可组合的并发精化验证，而相关理论问题直到最近几年才被解决，因此之前的操作系统内核验证项目都通过避免抢占和中断来简化操作系统内核验证，它们验证的内核代码是串行的。另一方面，现有的可组合的并发精化验证技术和中断验证方法都是基于简化的理论模型，不能直接将其应用于商业化的抢占式操作系统内核验证工作中，需要进行调整和扩展，因此如何支持带抢占和中断的并发内核的精化验证是操作系统内核验证工作的一个难点。

系统级性质的定义和验证　很多时候用户关心操作系统的一些系统级性质，这些性质不是某个 API 能够保证的性质，而是整个系统在运行过程中一直保持的性质，例如，系统不会发生死锁、系统不会发生优先级反转等等。如果直接在代码层定义这些性质，不仅会使得定义复杂难懂，同时也暴露了不必要的实现细节，而且直接在代码层验证性质会比较复杂和困难。如何清楚正确的定义并验证这些系统级性质也是操作系统验证中的一个难点。

这些难点中，支持内核级中断和抢占的多任务实时系统的验证尤为困难，这是之前的操作系统内核验证项目没有解决的问题，也是本文重点解决的问题。

1.2　研究现状

本节主要介绍相关研究工作的现状，第1.2.1节介绍操作系统验证工作的研究现状，第1.2.2节操作系统验证基于的并发精化验证理论的研究现状。

1.2.1　操作系统验证现状

近十年来，国际上众多大学与研究机构纷纷开展操作系统内核的形式化验证工作：例如澳大利亚国家通讯技术研究中心的 seL4 项目 [13, 14]；耶鲁大学的 CertiKOS 项目 [15–17]。；澳大利亚联邦科学与工业研究组织的 eChronos 项目 [18, 19]；德国德累斯顿工业大学的 VFiasco 项目 [20]、德国的 Verisoft 项目 [21–23] 与后续的 Verisoft-XT 项目 [24]、以及微软研究院的众多项目，如 Singularity[25]、VCC[26]、HAVOC[27] 和 Verve[28]。

seL4 项目组在 2009 年率先宣布成功验证了一个可以实际应用的操作系统内核 seL4[29]，其相关论文被当年计算机领域的顶级学术会议“The ACM Symposium on Operating Systems Principles (SOSP)”评为最佳论文，引起各方关注。seL4 内核代码多达 8000 行，编写的 Isabelle 定义和证明脚本超过 200,000 行，验证共耗费了 25 人年的工作量。为了兼顾操作系统内核效率和验证的复杂度，seL4 项目组先在一个函数式语言 Haskell 中实现了内核的原型，同时使用一个工具自动的将 Haskell 代码转换到定理证明工具 Isabelle/HOL 工具中形成可执行规范 (executable specification)，在可执行规范之上通过抽象规范 (abstract specification) 来描述操作系统的主要性质。为了避免庞大的 Haskell 运行时环境 (Haskell runtime) 的验证，他们手动地用 C 重新实现了操作系统代码。最后通过验证由 C 实现的操作系统代码和可执行规范之间的精化关系以及可执行规范和抽象规范之间的精化关系来保证操作系统的正确性。关于 C 代码编译过程的正确性，seL4 通过检验 (validation) 的方法验证了在 gcc 4.5.1 上编译出来的二进制代码的正确性 [30]。seL4 目前正在试图通过隔离（isolation）的方法在一个安全的操作系统内核的基础上构建一个安全的软件系统，包括 TCP/IP 协议栈，文件系统等等。由于 seL4 内核并不是实时的响应中断请求，而是通过轮询的方式在操作系统由内核态切换至用户态时检查是否有外部中断请求。所以 seL4 内核并

不是严格的并发操作系统内核,它在内核态时不及时响应外部中断请求从而避免了内核代码与外部环境间的细粒度并发，大大降低了精化验证的难度，然而很多实时嵌入式操作系统，如本文所验证的内核 μC/OS-II 则允许在内核代码中及时响应中断，内核代码的执行可以被中断打断并与其它任务并发执行，所以不能简单的把 μC/OS-II 的内核代码当做串行程序来验证，而需要将其视为并发程序来验证，这大大提高了验证的难度。

美国耶鲁大学的 Flint 研究组多年来在汇编语言级验证取得了较多成果[31–33]。早在 2008 年，就对操作系统中的中断带来的并发问题进行了相关研究工作 [34]，该工作验证的是汇编代码，首次将占有权转移 (ownership-transfer) 思想应用于处理中断带来的非对称并发问题，但该研究工作基于的并发分离逻辑只能证明程序的部分正确性（partial correctness），而部分正确性不足以描述复杂操作系统内核的正确性，而且该工作针对的是一个实验用的简单操作系统内核，与工业界中实际使用的操作系统内核差距较大。

CertiKOS 是美国耶鲁大学 Flint 研究组自行开发的一个虚拟机（hypervisor），支持对 Linux 的加载。他们的验证工作是针对 CertiKOS 的一个简化版本 mCertiKOS[16]。mCertiKOS 通过扩展可信编译器 CompCert[35] 来支持对带抽象原语的 C 程序的可信编译，并采用最强功能性规范（deep specification）和分层抽象方法，通过大约 40 层的分层抽象完成了目标机器码对程序抽象规范精化关系的完整验证。但 mCertiKOS 为了完成证明在代码实现上做了大量的妥协，引入了大量的冗余函数调用，这导致 mCertiKOS 编译生成的目标代码性能较差很难被实际使用。另外，实际已有的操作系统内核代码很难进行如此细粒度的分层，所以这种验证方法很难直接用于一个已有操作系统内核的验证。而且 mCertiKOS 整个编译过程的正确性是由 CompCert 保证的，他们对 CompCert 作了扩展来支持对抽象状态和抽象原语的编译和证明链接，但通过 CompCert 编译产生的目标机器码的性能问题也导致该方法很难被工业界所使用。与 seL4 类似，mCertiKOS 内核代码也是在关中断的环境下运行的，所以 mCertiKOS 内核的验证也是在串行执行的假设下完成的。最近他们在 mCertiKOS 的基础上验证了一些中断处理程序和设备驱动程序 [17]，但是他们要求这些中断处理程序之间没有共享资源，同时他们还对中断处理程序的代码作了大量的要求，不支持中断处理程序中发生上下文切换，所以无法验证抢占式操作系统内核。

eChronos 项目 [18] 希望实现并验证一个小的多用途的实时嵌入式操作系统（RTOS）eChronos。该项目只验证操作系统的功能正确性，不去验证程序最坏执行时间相关的性质。目前该项目提出了一个支持中断导致的并发操作系统验证框架，但只验证了操作系统的调度相关的性质 [19]。

Verisoft 项目组提出了一个内核验证框架 CVM[21]，该框架不仅包含了处理器、内存、外部设备的形式化语义，还包含了一个经过验证的 C0 语言编译器。这样，由 C0 语言编写的内核代码与应用代码就可以通过 Hoare 逻辑在源语言

级进行验证，通过编译器产生可靠的可执行代码。Verisoft 项目也在证明工具 Isabelle/HOL 中验证了一个操作系统 [36]，然而该框架也不支持在内核中响应中断。他们的验证项目 Verisoft XT [24]，通过使用自动证明工具 VCC [26] 来验证一个商用的 Hyper-V 虚拟机。VCC 支持通过添加辅助代码（auxiliary code）和辅助状态（ghost states）来验证并发的 C 程序，但是 VCC 并不保证上下文精化关系，而且目前没有相关论文阐述如何使用 VCC 来完成该虚拟机内多级嵌套中断的验证。

微软的 Verve 项目 [37] 结合了一个类型安全的操作系统和一个极小的硬件抽象层，该操作系统内核是并发的，但是他们验证的性质大多是类型安全相关的性质，比本文验证的上下文精化关系要弱得多，而且 Verve 将系统简化为只有一级中断。VCC/VerisoftXT 和 Verve 都使用了 SMT 求解工具 Z3 [38]，而本文的验证工作基于 Coq，Coq 会生成机器可检查的证明过程，这也使得本文的验证工作的可信计算基础（Trusted-Computing-Base）更小。

微软研究院开发了一个研究型安全操作系统——Singularity[25]。其大部分代码通过 Sing# 语言的类型系统来保证安全性，但是有关任务调度、Sing# 语言类型安全机制与垃圾收集的内核代码则需要另外进行验证。

目前国内系统软件内核验证的研究尚处于起步阶段。清华大学的朱允敏等人提出了一个基于多核的并行程序验证框架 [39]。长沙理工大学的肖增良等人研究了通过依赖图与依赖集解决程序验证中的并行调度问题 [40]。

综上所述，现有的主要操作系统验证项目都不支持抢占式内核的接口功能正确性的验证，表1.1是本文工作和已有主要操作系统验证工作的比较，本文工作是第一个支持抢占和嵌套中断的第三方并发内核的接口功能正确性验证。

系统内核 验证项目	支持 抢占	内核 并发	支持 嵌套中断	接口功能 正确性	第三方 内核	证明 工具
seL4 [13, 14]	×	×	×	✓	×	Isabelle/HOL
mCertiKOS [15–17]	×	×	×	✓	×	Coq
VCC/VerisoftXT [36]	×	✓	×	✓	✓	Z3
Verve [37]	✓	✓	×	×	×	Z3
本文工作	✓	✓	✓	✓	✓	Coq

表 1.1　本文工作和主要操作系统验证项目的比较

1.2.2　并发精化验证理论的现状

操作系统的正确性通常使用底层 API 的具体实现与其高层抽象之间的精化关系来刻画，这使得操作系统正确性验证需要基于程序精化验证技术。所谓两个程序 C 和 $\mathbb{C}$ 满足精化关系，或者说程序 C 是对程序 $\mathbb{C}$ 的精化，记作 $C \subseteq \mathbb{C}$，

那么程序 C 产生的行为是程序 $\mathbb{C}$ 行为的子集。由于 C 不会产生 $\mathbb{C}$ 不具有的行为，因此在任何使用程序 $\mathbb{C}$ 的地方，都可以使用程序 C 来替代。例如，下面这个程序可以看作原子执行的程序 $\langle x++\rangle$ 的一种实现：

$$\mathtt{local}\ r;\ r := x;\ x := r + 1;$$

它先将 x 的值读入局部变量 r，再将计算结果写入内存变量 x。容易看出，若不关心局部变量 r 的值，则这个程序是对 $x++$ 的精化。

但是由于不考虑并发环境对程序行为的影响，所以串行环境下建立的精化关系在并发环境下缺少可组合性。例如，图1.1中下方的两个线程各自精化上方的对应线程，但下方的整个程序却不是对上方程序的精化。假设 x 的初始值为 0，可知上方程序执行结束后 x 的值只能是 2（这里 $\langle x++\rangle$ 是原子执行的），而下方程序则会产生两种可能的结果：1 和 2。

$$\langle x++\rangle; \quad \| \quad \langle x++\rangle;$$

$$vs.$$

$$\begin{array}{lcl} \mathtt{local}\ r_1; & & \mathtt{local}\ r_2; \\ r_1 := x; & \| & r_2 := x; \\ x := r_1 + 1; & & x := r_2 + 1; \end{array}$$

图 1.1　并发导致精化关系丢失

如前所述，已有操作系统内核验证工作都不考虑内核级中断和抢占，这些工作都是基于串行环境下的精化验证技术，无法解决本文关心的内核级中断和抢占引发的并发精化验证问题。近年并发精化验证技术获得了长足发展 [7–12]。

中科大 -耶鲁高可信软件联合研究中心在并发精化验证技术的发展过程中作出了重要贡献，他们的工作也是本文主要的理论基础。2012 年该中心提出了一种基于依赖/保证的模拟关系 RGSim（Rely-Guarantee-based Simulation）[7]，作为并发程序精化的通用验证技术。RGSim 以依赖/保证条件 [41] 为参数，描述任务和并发环境之间的影响。RGSim 关系具有并发组合下的可组合性，RGSim 将多任务程序的精化证明分解为单个任务上的精化证明。2013 年他们接着提出了一个用以模块化地、高效地验证并发对象的线性一致性的程序逻辑 [8]。该逻辑基于一元并发程序逻辑 LRG [42]，支持可线性化点不固定的并发对象的验证。为了证明程序逻辑的可靠性，该工作通过扩展了 RGSim 并提出了一个新的程序模拟关系，而该模拟关系可以保证一种上下文精化关系，该上下文精化关系与线性一致性等价。之后的工作 [9, 12] 进一步研究了如何通过并发精化技术验证程序的终止性和一些常见的进展性性质，包括无等待性、无锁性、无阻碍性、无饥饿性和无死锁性。

马克思 -普朗克软件系统研究中心（Max Planck Institute for Software Systems）在并发精化关系上也做了大量研究，[10] 研究了如何验证细粒度并发

数据结构和对应的粗粒度规范之间的上下文精化关系。[11] 研究了高阶函数（high-order function）及其带来的并发问题。

1.3　本文贡献

针对第1.1节提出的操作系统验证的困难，本文工作有如下贡献：

首先，本文最主要的贡献是扩展已有的关于中断处理程序验证的工作 [34]，调整已有的并发程序上下文精化关系验证的相关理论 [7–11]，并将它们整合成一个实用的抢占式操作系统内核的验证框架。该验证框架有如下特点：

- 验证框架将操作系统内核的正确性定义为一种上下文精化关系。该上下文精化关系要求对于任意的应用程序，其运行在底层机器模型上产生的行为不会比在高层抽象机器模型上产生的行为多。这里底层机器模型是实际程序运行的机器，对于一般的操作系统内核就是 C 语言内嵌汇编的机器模型；高层机器模型是指应用层程序员理解中的抽象机器模型，该模型会抽象掉很多实现细节而只保留理解系统 API 功能所必须的内容。上下文精化关系不仅能很好的刻画系统 API 的功能正确性，还能将抽象机器上证明的一些性质带入到底层实际机器运行的程序中，这样就可以在高层的抽象机器模型上进行性质证明，这比直接在底层实际机器模型上进行性质证明要简单得多。

- 验证框架通过将汇编代码封装成一些功能完整的原语，并将 C 语言机器模型扩展，完成了底层内核代码的建模。验证框架还提供了一个简单的建模语言用于定义高层规范。一方面，希望规范能够尽量抽象掉调度实现中的不重要的细节，但同时又希望规范能够保留关键的信息来验证抢占式内核关于任务调度的一些重要的性质。本文的规范语言同时兼顾了关于任务调度的抽象和表达力。

- 验证框架包含了一个新的并发精化程序逻辑 CSL-R 来验证并发操作系统内核的上下文精化关系。该程序逻辑支持多级中断和可配置的调度程序，将并发分离逻辑（Concurrent Separation Logic, CSL）[43] 的断言语言扩展为关系型断言，并将占有权转移思想应用于一个实际的中断模型（x86 中断模型）下的多级中断的验证。同时支持可组合验证，其将两层机器模型的上下文精化关系的验证转化为对内核的每个函数以及中断处理程序的验证。

- 实际操作系统验证中，规范和推理规则都相对繁琐。如何开发出合适的自动证明策略来减轻验证人员的负担也是一个关键问题。框架验证中包含了一些实用的自动证明策略来辅助证明，能大大提高证明效率。

其次，本文利用上述验证框架首次成功验证了一个商用抢占式操作系统内核 μC/OS-II [1] 的核心模块，其中包括中断处理程序、任务调度、时间管理模块以及消息队列、邮箱、信号量、互斥锁等四个通信模块 (为了体现对多级中断的支持，还引入了一个简单的计数器中断)。已有工作 [16, 18, 19, 29, 36, 37] 都是针对自己开发的操作系统，在操作系统开发的过程中已经考虑了验证的问题，而本文工作针对的是第三方开发的商用操作系统内核，而且在验证过程中只对源码做了很小的改动，这更能说明本文验证框架的一般性和实用性。此外，本文还在高层抽象模型上证明了 μC/OS-II 中的互斥锁不会发生优先级反转（PIF）这也说明了高层规范语言的实用性。

最后，本文所描述的操作系统内核验证框架和 μC/OS-II 的验证都已在定理证明工具 Coq [2] 中实现，生成了机器可检查的证明。证明脚本的代码量总计约 225000 行 [3]，这是第一个商业化实时操作系统内核的核心功能模块的机器可检查证明。

1.4　本文组织结构

- 第二章介绍操作系统内核验证的相关技术背景；
- 第三章总体地介绍操作系统内核验证框架及其各个组成部分，包括内核正确性定义、内核的形式化建模、并发精化程序逻辑以及自动证明策略，然后在第四章、第五章和第六章详细介绍各部分内容；
- 第四章具体给出内核底层和高层机器模型的形式化定义，以及操作系统内核正确性的形式化定义；
- 第五章具体介绍证明操作系统内核正确性的并发精化程序逻辑 CSL-R，包括断言语言、推理规则以及推理规则的可靠性（soundness）证明等；
- 第六章具体介绍对验证框架在分数权限（fractional permission）和局部不变式方面的扩展；
- 第七章介绍基于推理规则设计的自动证明策略，用于提高验证效率；
- 第八章介绍应用验证框架验证 μC/OS-II 核心模块的功能正确性，并在高层机器模型上完成了 PIF 性质的证明。

第二章　相关技术背景介绍

本章将简要介绍本文工作中涉及到的相关技术、工具和待验证的目标内核。首先在2.1节介绍操作系统内核中的抢占和硬件中断导致内核的并发行为；然后在2.2 节介绍基于并发分离逻 [43] 的占有权转移思想验证中断处理程序 [34]；接着在2.3 节介绍并发程序的精化验证；最后在第2.4节和第2.5节分别介绍使用的定理证明工具 Coq [2] 和待验证的目标内核——商业抢占式操作系统内核 μC/OS-II [1, 44]。

2.1　抢占和硬件中断

中断是指当某个事件发生时，CPU 停止运行正在执行的程序, 而转去执行处理该事件的程序, 处理完该事件后, 返回原程序继续执行下去。嵌套中断是指允许中断处理程序在执行的过程中开中断并被别的中断请求打断。在抢占式操作系统中，正在执行的任务 T_1 在任何程序点都可能会被中断打断（除非 CPU 处于关中断状态），而中断处理程序在执行的过程中又可能切换到别的任务 T_2 执行。这种情况称任务 T_1 被任务 T_2 抢占，因为任务 T_1 并不是主动执行调度程序来放弃 CPU，而是因为环境的变化导致 CPU 资源被其他任务抢占。内核级抢占的意思是指任务在执行内核代码的时候被其他任务抢占。为了支持内核级抢占，操作系统内核实现需要满足下面两个条件：（1）内核代码执行过程中允许开中断，（2）中断处理程序执行过程中会发生调度和上下文切换。在嵌入式实时操作系统内核中，中断和抢占在内核中被大量使用来保证系统的实时性。

图2.1给出了一个结合了嵌套中断、抢占以及任务主动执行调度的例子，其执行过程如下：

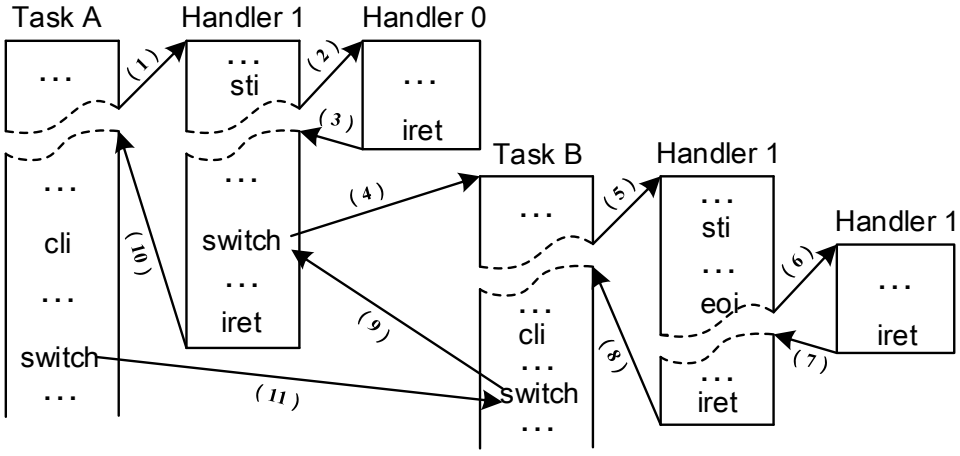

图 2.1　抢占和嵌套中断

(1) 任务 A 被中断同时程序的控制流被切换到 1 号中断处理程序

(2) 1 号中断处理程序在执行过程中开中断，并被更高优先级的 0 号中断打断
(3) 0 号中断执行结束返回 1 号中断继续执行
(4) 1 号中断中执行上下文切换，切换到任务 B 执行，这里任务 A 被任务 B 抢占
(5) 任务 B 执行的过程中被 1 号中断打断
(6) 1 号中断在执行过程中执行 eoi 指令，表示告诉中断处理器 1 号中断执行结束，1 号中断发生重入
(7) 嵌套的 1 号中断返回
(8) 1 号中断返回任务 B
(9) 任务 B 执行上下文切换并切换回 1 号中断
(10) 1 号中断执行结束返回任务 A
(11) 任务 A 主动执行调度程序又切换到任务 B

从上例可以看出，由于抢占式内核的控制流相当复杂，中断处理程序和 API 的正确实现变得相对困难。同时，在验证系统代码时需要考虑中断处理程序和其他任务代码的不确定行为，如何对环境不确定的干扰进行抽象来支持内核代码的可组合验证变得十分困难。

x86 的硬件中断模型　本文对操作系统内核的形式化建模是针对 x86 硬件中断模型（基于 8259A 中断控制器），这里先简单的介绍一下 x86 硬件中断模型的行为。在 x86 机器上，通过中断控制器 8259A 芯片和 CPU 中的 EFLAGS 寄存器完成。为了简化机器模型，本文对 8259A 和 CPU 作了一些抽象，首先 x86 中的 EFLAGS 寄存器只保留中断状态 IF 位，其值为 1 表示允许中断，为 0 表示屏蔽中断。指令 **sti** 将 IF 置为 1，用来允许中断，指令 **cli** 用来将 IF 置为 0。在 8259A 中，有一个 8 位寄存器 *isr*，其每一位对应一个硬件中断，如果某位为 1 表示其对应的中断正在被处理。除了 *isr*寄存器外，8259A 中断控制器中还有其他寄存器来完成中断控制，例如 IMR 和 IRR，这些寄存器要么是在初始化后就确定了的，要么是对 CPU 不可见，这里只保留 *isr*寄存器就可以刻画所有的中断处理过程中会遇到的情况，同时假定 *isr*中越低的位所对应的中断优先级越高，0 位对应的中断优先级最高。当 n 号中断请求到达时，检查 IF 位和 *isr*寄存器，如果允许中断（IF=1）而且没有优先级高于或等于 n 的中断正在处理，则进入该中断请求被响应。中断被响应后，会将 IF 位置为 0，表示中断被屏蔽，同时将 *isr*寄存器的 n 位置为 1，表示 n 号中断正在被处理。**eoi** k 指令用于将 *isr* 中的 k 位清空，表示 k 号中断执行结束。

2.2　基于占有权转移的中断验证

并发分离逻辑 [43] 通过使用全局资源不变式将系统资源按照每个并发执行的任务划分为隔离的各个部分，然后通过要求每个任务只能访问自己占有的那

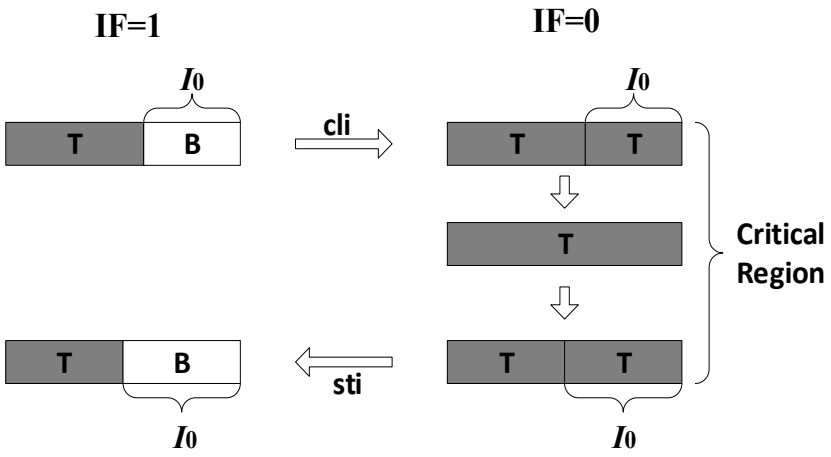

图 2.2　单个中断占有权转移的语义

部分资源来阻止任务间的数据竞争（Data Race），但是为了支持资源在并发任务间的共享，这种划分不能是静态的，资源的占有权必须能够在并发执行的任务间动态的转移，而同步操作（例如获得释放锁、内核中的开关中断操作等）的语义就可以建模成为引发共享资源的占有权转移的操作。

基于上述资源占有权转移的思想，[34] 通过扩展并发分离逻辑并赋予中断相关的操作占有权转移的语义完成了一个简单模型下单级中断的验证。因为中断处理程序具有较高优先级，可以随时打断任务执行，而任务无法打断中断执行，所以和传统的对称并发模型不一样，中断和任务之间的并发是非对称的。如图2.2所示，让中断处理程序先选择其要访问的资源（图中的 **B**），当中断打开时，因为中断随时可能发生，所以需要保证资源 **B** 是可用的，并且是良构的（well-formed），即满足资源不变式 $\mathbf{I_0}$ 的要求，良构的资源块 **B**可以保证中断处理程序能够安全的执行。例如，如果 **B**对应操作系统内核中的一个链表，那么需要保证此时该链表是完整可用的。

图中灰色的部分表示任务的局部资源。当任务执行 **cli** 指令来关闭中断时会发生占有权转移，**B** 会变成任务的局部资源。因为此时不会有中断处理程序来访问 **B**，所以中断关闭后不再要求资源 **B** 是可用的和良构的，此时任务可以任意的访问资源 **B**。在执行 **sti** 之前，需要保证资源 **B**良构，这是因为一旦中断打开，随时中断都可能发生并访问 **B**。执行 **sti** 也会发生占有权转移，**B**从任务的局部资源变成共享资源。类似的，当中断发生时，**B**会自动变为中断处理程序的局部资源，**iret** 后会重新变为全局资源。

因为 **B**被中断处理程序和任务共享，所以当 **B**处于全局资源中时，必须要求其是良构的。**cli** 和 **sti** 的占有权转移语义可以简单得表示为下面的两个逻辑规则：

$$\frac{}{I_0 \vdash \{p_t\}\,\mathbf{cli}\,\{p_t * I_0\}} \qquad\qquad \frac{}{I_0 \vdash \{p_t * I_0\}\,\mathbf{sti}\,\{p_t\}}$$

其中 p_t 是当前任务的局部资源，对应图2.2中的 **T**。任务执行 **cli** 后会获得

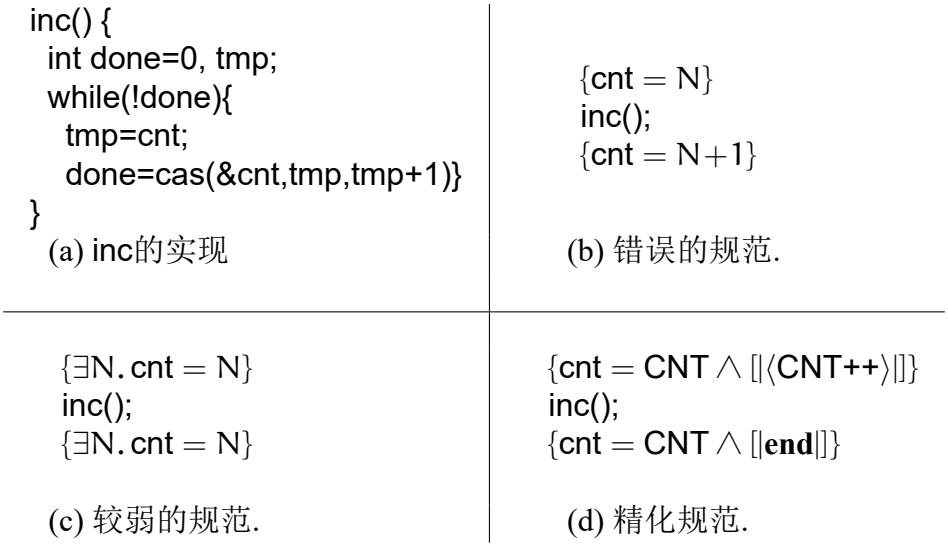

图 2.3　并发程序规范

I_0 描述的良构的共享资源 **B** 的占有权。任务执行 **sti** 之前需要保证占有的共享资源是良构的，在执行 **sti** 之后会释放占有的共享资源。这里要注意 **B**和 **T** 的划分是一个通过分离逻辑中的分离连接“ * ”符 [45] 进行的逻辑划分，并不要求在程序状态模型上进行物理划分。而且这个划分是根据 I_0 的定义来动态确定的，**B**包含的资源会根据情况发生变化。

2.3　并发程序的精化验证

对于并发程序来说，精化关系能够比传统的霍尔三元组提供更强的功能正确性。图2.3给出的简单的例子很好的说明了这一点。图2.3(a) 中给出的函数 inc是一个可以多个任务同时调用的加一计数器。这里 cas 表示比较和交换（compare-and-swap），它带着三个参数，如果第一个参数和第二个参数相等，则将第三个参数的值赋值给第一个参数，否则就什么也不做。cas 是原子执行的。图2.3(b) 给出的规范在串行执行的情况下是正确的并且很好的描述了该函数的功能。但是在并发的情况下其他任务可能也在执行 inc并修改 cnt 的值，所以该规范是错误的。为了保证正确，传统的霍尔逻辑只能采用图2.3(c) 所示的规范，但是这个规范太弱了，根本无法刻画出程序的功能。图2.3(d) 给出了一个关系型规范表示 inc精化了一个抽象的原子操作 ⟨CNT++⟩ [9]，关系型断言描述了三个重要的部分，底层实际的机器状态（cnt），高层抽象的机器状态（CNT），和底层程序精化的高层规范代码（⟨CNT++⟩）（高层规范操作可能不是原子的 [9]）。所以可以看出，传统的霍尔风格的程序逻辑所描述的是程序执行后机器状态的变化，而关系型逻辑通过对应的规范代码不仅仅可以表达程序执行结束时的状态，还可以更细粒度的记录程序执行的过程中的行为。前断言要求一开始时底层机器状态 cnt 和高层机器状态 CNT 是一致的，并且这段代码对应一个高层的原子操作，该原子操作的语义是将高层状态 CNT 加 1。后断言表示，在程序执

行结束后底层和高层的机器状态仍然是一致的，并且高层的原子操作也执行结束。这里 inc是对 ⟨CNT++⟩ 的精化。

本文用于验证操作系统内核的并发精化程序逻辑采用上述类似的思路，将传统的霍尔风格的并发分离逻辑扩展为验证程序精化关系的程序逻辑，同时借鉴 [34] 中关于中断处理程序验证的工作，并对其进行扩展并支持多级中断和中断嵌套。第五章会对验证操作系统内核正确性的并发精化程序逻辑做详细的介绍，并给出程序逻辑的可靠性（soundness）证明。

2.4　证明工具 Coq 简介

Coq[2] 是法国国家信息与自动化研究所研发的一个交互式的定理证明工具，它提供一个具有很强表达能力的语言，可以用来形式化定义数学运算，程序语言，算法等等，并可以基于这些定义给出性质和定理并证明。最近这几年它也被广泛应用在系统软件的形式化验证中 [16, 17, 35, 46]。本文所描述的操作系统内核的验证工作全部在 Coq 实现。

2.4.1　在 Coq 中定义 C 表达式

```
Inductive expr :=
  | evar : var → expr
  | ederef : expr → expr
  ...
end.
```

```
Notation " x' " := (evar x).
Notation " * A " := (ederef A).
```

图 2.4　Coq 中定义 C 表达式和 Coq 记号

下面通过在 Coq 中定义 C 语言的表达式来简单了解下 Coq。C 语言的表达式可以是变量表达式，解引用表达式，结构体成员表达式等等。图2.4归纳定义了表达式，其中evar 和ederef 分别是变量表达式、解引用表达式的构造子，var 表示变量类型。例如evar x 表示一个变量表达式；ederef (evar x) 表示一个解引用表达式。Coq 还允许定义记号（Notation），这样使得 Coq 中表达式的语法和 C 语言本身的语法非常类似，例如在定义了图2.4中的记号后，ederef (evar x) 就可以写为 * x'。

2.4.2　自动证明策略编程语言 Ltac

Coq 自带了策略编程语言 Ltac。Ltac 可以让用户自己实现一些决策过程来完成证明的自动搜索。对于很多特定类型的问题，证明过程都是遵循某些固定模式，这种情况下自动证明策略能够大大提高证明效率。Ltac 语言为在 Coq 中

实现自动证明和交互式手动证明相结合提供了保障。本文中的自动证明策略都是通过 Ltac 语言来实现的。

2.5　μC/OS-II 简介

μC/OS-II [1, 44] 是一个被商用的嵌入式操作系统，已经被广泛移植到各种嵌入式处理器上，如 Cirrus Logic 公司生产的 EP7312 芯片，Samsung 的 S3C44B0X 微处理器，德州仪器公司 (TI) 的 TMS320F2812DSP 处理器等；并已经应用于飞机电器系统测控终端、配电终端电子仪表、短信息电话机等各种嵌入式设备中。

μC/OS-II 主体是一个基于优先级调度的抢占式的实时内核，提供最基本的系统服务，包括任务管理、信箱、内存管理、互斥锁、消息队列、信号量、时钟管理等，内核代码一共 6600 多行，其中常用代码约 5289 行，图2.5是 μC/OS-II 的代码结构，可以看出系统的层次划分本身已经比较清晰，处理器相关部分集中在 OS_CPU.H，OS_CPU_A.ASM，OS_CPU_C.C 三个文件，用户需要编写的配置代码集中在 OS_CFG.H 和 INCLUDES.H 两个文件，剩下的文件是内核的主体代码，这些代码独立于硬件，可以方便的被移植到不同的处理器平台上。本文选择 μC/OS-II 作为验证对象，主要由于其具有以下特点：

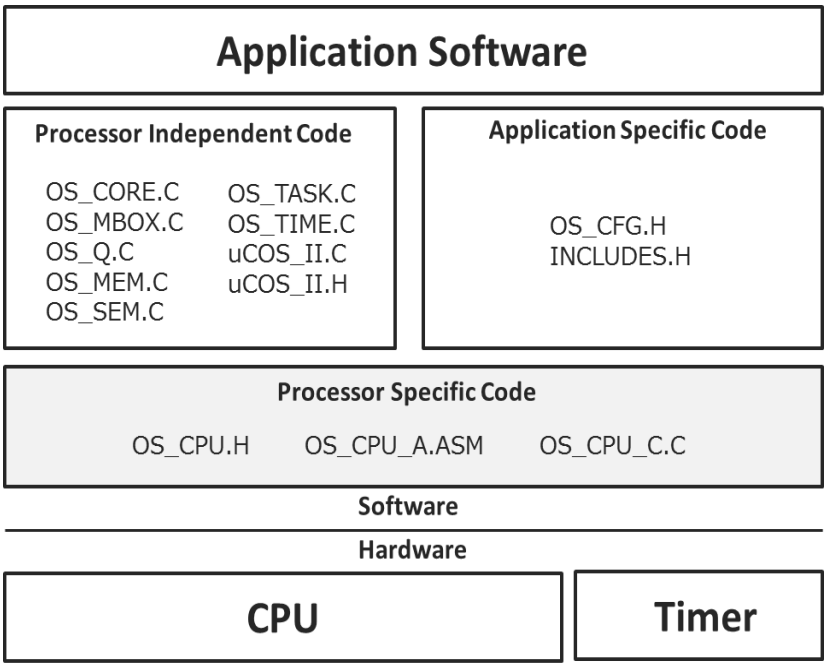

图 2.5　μC/OS-II 代码结构

多任务，抢占式内核　为了满足实时性，μC/OS-II 设计为多任务抢占式内核，多任务抢占导致的高度并发和复杂控制流等问题也是本文提出的验证框架想要解决的主要问题。

允许中断嵌套　μC/OS-II 支持最多 255 层中断嵌套，多级中断嵌套问题也是本文提出的验证框架想要解决的一个重要问题。

文档规范完整　其源码中含有大量规范的注释，并有大量的书籍和资料。这对于验证工作非常重要，因为这让验证人员充分了解操作系统开发人员对代码的理解。如果各个系统 API 的注释和规范不完整，那只能凭验证人员自己的理解来写规范（specification），这不仅容易出现理解错误，也使得验证的真实性大打折扣。

可移植性　μC/OS-II 中的源码大部分是用 C 语言写的，与处理器相关的部分是用汇编写的，汇编编写的内容已经被压缩到最低限度，以方便 μC/OS-II 移植到其他微处理器上。这和本文验证框架的设计思路是一致的，本文验证框架需要将汇编代码部分封装成原语，如果内核中有太多的汇编代码会将验证工作变得相当繁琐。

基于上述特点，本文选择 μC/OS-II 作为待验证的目标操作系统内核。

第三章　操作系统内核验证框架概述

本章主要从整体上对操作系统验证框架的结构做简要的介绍。图3.1给出了验证框架的主体结构，它包括三个部分：**A** ——操作系统内核建模；**B** ——并发精化程序逻辑 CSL-R；**C** ——自动证明策略。下面在第3.1节简要介绍操作系统内核正确性定义，在第3.2节简单介绍操作系统内核的形式化建模方法，在第3.3节简单介绍并发精化程序逻辑的主要设计思想，最后在3.4节简单介绍自动证明策略。

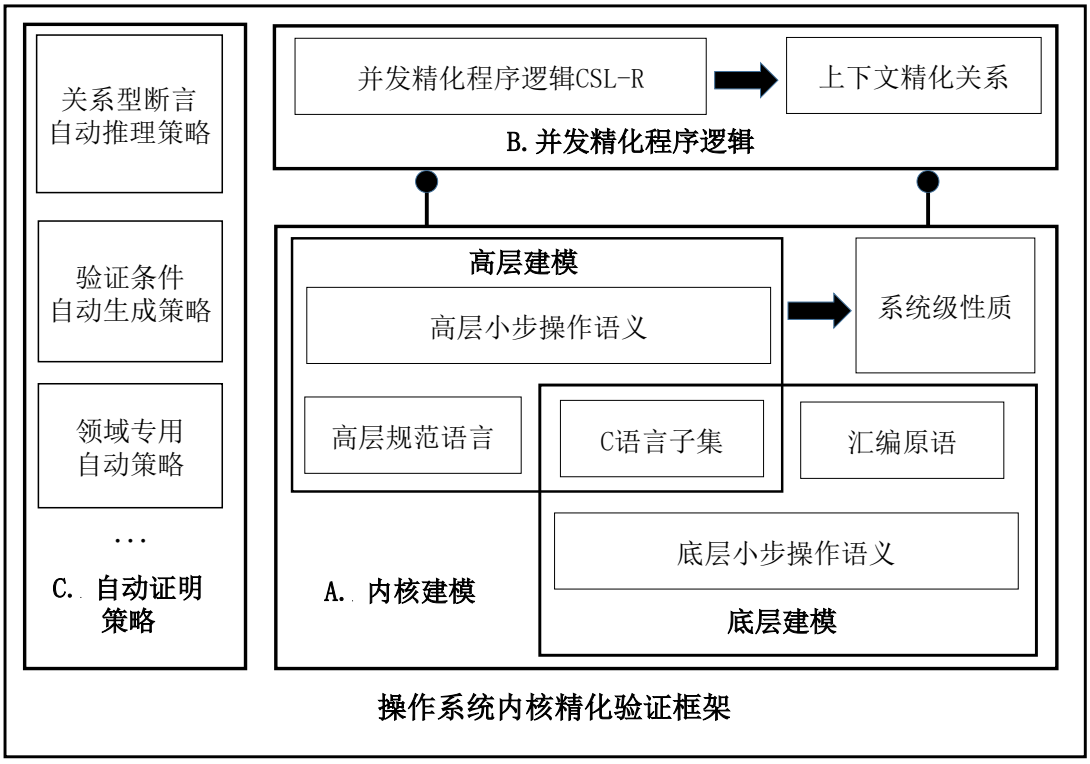

图 3.1　操作系统内核验证框架结构

3.1　操作系统内核的正确性

操作系统内核通过隐藏机器硬件实现细节给程序员提供了一个抽象编程模型，使得程序员能够使用系统 API 方便的编写应用程序，正确的操作系统内核应该保证系统底层的具体实现与其高层抽象之间的一致性 [47]。因此操作系统内核的正确性可以理解为系统底层具体实现和高层抽象之间满足精化关系。具体而言，这里可以认为存在三种程序实体，应用程序 A，系统应用和中断的抽象规范 Ⓞ 及其底层实现 O。假定应用程序 A 用 C 语言实现。当发生系统调用或者中断处理程序被响应时，Ⓞ 中的程序会在抽象层被调用，O 中的程序会在实现层被调用。这样操作系统内核的正确性可以定义为在任意的上下文 A 下，系

统实现 O 精化其抽象规范 𝕆：

$$\forall A.[\![A[O]]\!] \subseteq [\![A[\mathbb{O}]]\!]$$

这里的定义是非形式化的，$[\![A[O]]\!]$ 和 $[\![A[\mathbb{O}]]\!]$ 分别表示底层和高层可见行为的集合（如：I/O 输出等）。注意这里实现应用程序 A 的语言和实现内核 O 的语言可以是无关的，为了方便验证，假定应用程序和内核程序共用一个相同的 C 语言子集。

3.2　操作系统内核建模

验证操作系统内核的正确性，首先需要对操作系统内核形式化建模。内核建模包含两个部分：（1）操作系统内核实际运行的机器建模（底层机器模型）；（2）应用层程序员眼中的机器建模（高层机器模型）。

3.2.1　底层机器模型

如前所述，底层程序包含应用程序代码 A 和内核代码 O。本文假定内核代码不会访问应用程序状态，应用程序代码只能通过系统调用访问系统内核状态，所以在底层机器状态上物理地将应用程序状态和内核程序状态分离。

内核语言用于实现操作系统内核 O。大多数操作系统内核都是用 C 语言内嵌汇编实现。然而直接给出 C 语言内嵌汇编的语义，需要暴露寄存器和栈，这会导致语义非常复杂。为了避免这个问题，可以将汇编代码封装为一些汇编原语，同时在底层机器状态中扩展出一些相关的抽象状态，这些汇编原语在底层机器模型中是原子执行的。汇编原语主要用来实现中断控制（例如：开关中断和发送中断结束信号 EOI 等等）和上下文切换。这里底层机器模型将汇编代码实现的上下文切换操作封装为一个抽象的汇编原语，来描述底层机器模型中的上下文切换和多任务并发。这里值得注意的是，本文并没有验证具体汇编代码实现的正确性，而是通过汇编抽象原语来刻画汇编代码的抽象行为。汇编代码和抽象汇编原语之间的一致性验证作为我们将来的工作，不在本文工作范围内。

3.2.2　高层机器模型

从应用层程序员的角度，本文将操作系统内核抽象为一个带系统调用的、支持多任务的 C 语言。如前所述，C 语言用来实现应用程序 A，系统调用会去调用 𝕆 中的抽象规范。验证框架提供了一个规范语言来实现 𝕆 中的抽象规范。同时将底层内核程序状态抽象为一些抽象内核状态，这些抽象内核状态会隐藏很多底层实现细节而只保留应用层程序员理解内核所必须的内容，例如，就绪任务表在程序员的理解中就是所有就绪任务的集合，但在底层可能是用链表、数组或位图实现的，这些数据结构相关的信息是应用层程序员不关心的。抽象

规范只会访问抽象内核状态，高层应用程序无法通过一般的 C 语言指令访问抽象内核状态，只能通过系统调用对其访问。

应用层程序员无法控制中断（例如：开关中断），中断在抽象层被理解为在任意程序点都可能到来的非确定性事件，$\mathcal{O}$ 中同样包含这些中断的规范。

高层语言不是传统的多任务交错执行的并发模型，而是通过一个抽象调度器来实现多任务并发执行，该调度器可以被实例化为各种不同的调度策略。高层规范语言中包含一个抽象指令**sched**，该指令会调用抽象任务调度器。在高层暴露调度相关的细节可以用来描述和验证与任务调度相关的性质，例如互斥所满足无优先级反转的性质（PIF）。

由于高层机器模型具有足够的表达力，可以用来支持验证调度相关的系统级性质，例如，系统是否保证当前任务是就绪任务中优先级最高的（抢占），PIF 等等。此外，在高层机器模型上验证这些系统级的性质带来了两方面的好处，一方面在高层定义这些性质更加直观清晰（不需要暴露很多不相关的底层细节）；另一方面由于高层规范语言和高层模型相对简单，在高层验证这些性质会更加简单。

小步操作语义　为了反映中断可以发生在任意两个机器指令之间，程序语句的执行和表达式计算都必须是和机器指令相同的粒度。底层机器模型采用和 CompCertTSO [48] 类似的小步操作语义，其中表达式计算也被分解为小步执行，但由于目前只考虑单核处理器上的执行，所以并没有使用 CompCertTSO 中的 TSO(Total Store Order) 的弱内存模型。

3.3　并发精化程序逻辑 CSL-R

具有可组合性的精化关系证明一直是一个非常困难的问题，直到近几年大量的技术被提出用于解决该问题。本文提出一个并发分离逻辑 [43] 风格的并发精化程序逻辑 CSL-R 用于验证刻画操作系统内核正确性的上下文精化关系，该程序逻辑的设计思想基于已有的可组合的并发程序模拟关系 RGSim [7] 和并发精化验证的相关理论 [8, 9]，但作了如下的简化和扩展：

- 由于在内核中对于共享数据的访问都是发生在临界区中，不存在细粒度的无锁并发，所以 CSL-R 抛弃了用于验证细粒度并发的相关机制（例如：投机和帮助机制）。同时将 RGSim 中的描述状态转换关系的依赖/保证条件替换成了更为简单的描述共享状态的全局不变式，借鉴 CSL 的思想，通过全局不变式来刻画共享资源的良构性（well-formedness）并赋予同步原语资源占有权转移语义的方式来实现并发内核的局部推理。

- 基于已有的工作 [34] 中的推理单个中断的思想，同时结合并发精化验证技术来支持多级中断的推理。给出了中断控制原语（包括 **cli**/**sti**, **eoi** 和**iext**等

等）基于占有权转移的公理语义。验证框架的中断模型采用的是实际的 x86 中断模型，并做了少量的抽象和简化。

- 和已有理论工作中的并发模型不同，操作系统底层任务调度和上下文切换都是通过显示地调用相关代码来实现。为了保证精化程序逻辑的可靠性和可组合性，要求在底层执行上下文切换的时候，高层也必须执行抽象调度程序并切换到相同任务。

参考 [49] 中关于程序逻辑可靠性的证明，通过构建多层具有可组合性的程序模拟关系来证明程序逻辑中推理规则的可靠性，用于证明程序逻辑可靠性的程序模拟关系的定义与 RGSim [7] 类似，采用类似 CSL 中描述共享资源的全局不变式替换 RGSim 中的依赖保证条件来描述程序和环境的交互，由于操作系统内核中对共享资源的访问都发生在临界区内，同步访问控制的方式相对比较简单, 全局不变式提供了足够的表达力描述程序和环境的交互，而且它本身比依赖保证条件简单，可以简化内核的证明。

3.4　自动证明策略

自动证明策略的设计对于像操作系统内核验证这样的代码量较大的验证项目来说是至关重要的。一方面自动证明策略可以将那些简单而冗长的证明自动化，减少大量的 Coq 证明脚本；另一方面也可以帮助隐藏很多底层的逻辑细节，省去了代码验证人员的学习负担。特别是对于操作系统内核的验证来说，要支持一个表达力足够的 C 语言子集和封装的汇编原语，会使得程序语言和机器状态变得相当复杂。根据以往的代码验证经验 [34, 50], 在没有自动证明策略的支持下，一行 C 程序的验证平均需要 100 多行的 Coq 证明脚本，这代价是非常大的。同时由于机器指令和程序状态的复杂，程序逻辑提供的推理规则（包括辅助推理规则）一共有 100 多条，还有大量的封装的数据结构的断言（例如结构体和链表），如果要理解这些细节，代码验证人员的学习负担会非常大。本文通过借鉴已有的在霍尔逻辑上的自动化验证的工作 [51, 52]，并将其扩展到并发精化程序逻辑上，大大的提高了验证的效率。自动证明策略主要包含以下几个部分：

(1) 自动选择合适的推理规则并应用该规则。由于 CSL-R 建立在一个真实的 C 语言子集上的程序逻辑，C 语言丰富的语言特征导致会出现很多种情况。例如，对于赋值语句就可能有结构体成员变量的赋值，对数组成员的赋值等等情况。对于不同的情况 CSL-R 都为用户提供了对应的辅助推理规则，而自动证明策略“hoare forward”会根据当前上下文选择自动合适的推理规则，这样验证人员就不需要再去记住大量的推理规则，从而减轻了验证人员的负担。

(2) 关系分离逻辑断言之间蕴含关系的自动证明。因为在使用 CSL-R 验证内核代码过程中，会产生大量的关系分离逻辑断言之间的蕴含关系的证明，而这些证明中有很大一部分是相当机械的。自动证明策略“sep auto”通过扩展标准分离逻辑断言的蕴含关系的证明策略 [51, 52] 来支持关系分离逻辑断言蕴含关系的自动证明。

(3) 内核功能相关的数学性质的自动证明。在验证内核代码的过程中同样会产生大量描述内核功能正确的数学关系需要证明，例如，内核代码中经常会使用位运算，这就导致产生大量带有位运算的 32 位机器整数的数学性质。这些性质的证明也相当繁杂，领域专用的证明策略“mauto”用来自动证明这类数学性质。

这些自动证明策略大大提高了证明的效率，μC/OS-II 的验证工作中一行 C 程序的验证代价已经被压缩到 27 行 Coq 证明脚本。具体的自动证明策略的实现细节在第七章。

3.5 本章小结

本章介绍了验证框架的总体结构并简单了解了各个组成部分，验证框架包括底层机器建模、高层机器建模、基于精化关系的操作系统正确性定义、用于验证操作系统正确性并发精化程序逻辑 CSL-R 以及基于该程序逻辑的自动证明策略。下一章将具体介绍底层机器模型和高层机器模型以及操作系统正确性定义，第五 章将介绍并发精化程序逻辑 CSL-R，第七章将介绍自动证明策略。

第四章　操作系统建模及操作系统正确性定义

第三章简单介绍了验证框架的各个组成部分。本章首先给出底层机器模型和高层机器模型的形式化定义，然后给出操作系统正确性的定义。

4.1　底层机器模型

如前所述，底层机器上实际运行两种程序语言，一种是用来实现内核的 C 语言子集内嵌汇编（内核语言），另一种是应用层程序员使用的程序语言（应用语言）。为了方便，本文验证框架中假定应用语言是和内核相同的 C 语言子集。本节先介绍应用语言（C 语言子集），然后再介绍内核语言。

4.1.1　应用语言

(*Ident*)	id	$\in$	$\mathbb{Z}$	(*DeclList*)	$\mathcal{D}$	$\in$	$\mathtt{nil} \mid (\mathsf{id}, \tau) :: \mathcal{D}$
(*FName*)	f	$\in$	$\mathbb{Z}$	(*OpVal*)	$\hat{v}$	$::=$	$\bot \mid v$
(*ValList*)	$\bar{v}$	$::=$	$\mathtt{nil} \mid v :: \bar{v}$	(*ExprList*)	$\bar{e}$	$::=$	$\mathtt{nil} \mid e :: \bar{e}$
(*TypeList*)	$\mathcal{T}$	$::=$	$\mathtt{nil} \mid \tau :: \mathcal{T}$	(*DeclList*)	$\mathcal{D}$	$::=$	$\mathtt{nil} \mid (\mathsf{id}, \tau) :: \mathcal{D}$
(*UOP*)	$\mathtt{uop}$	$::=$	$\sim \mid ! \mid \ldots$	(*BOP*)	$\mathtt{bop}$	$::=$	$+ \mid - \mid \gg \mid \& \mid \ldots$
(*Type*)	τ	$::=$	$\mathtt{Tnull} \mid \mathtt{Tvoid} \mid \mathtt{Tint8} \mid \mathtt{Tint16} \mid \mathtt{Tint32} \mid \mathtt{Tptr}(\tau)$ $\mid \mathtt{Tcomptr}(\mathsf{id}) \mid \mathtt{Tarray}(\tau, \mathsf{k}) \mid \mathtt{Tstruct}(\mathsf{id}, \mathcal{D})$				
(*CExpr*)	e	$::=$	$\mathsf{x} \mid \mathsf{k} \mid *e \mid \&e \mid e.\mathsf{id} \mid e[e] \mid (\tau)e \mid \mathtt{uop}\ e \mid e\ \mathtt{bop}\ e$				
(*CltStmts*)	d	$::=$	$e = e \mid \mathsf{f}(\bar{e}) \mid e = \mathsf{f}(\bar{e}) \mid \mathsf{d}; \mathsf{d} \mid \textbf{if}\ (e)\ \textbf{then}\ \mathsf{d}\ \textbf{else}\ \mathsf{d} \mid \textbf{while}\ (e)\ \mathsf{d}$				
		$\mid$	$\textbf{return}\ e \mid \textbf{print}\ e \mid \ldots$				
(*CltFDef*)	*cfd*	$::=$	$(\tau, \mathcal{D}_1, \mathcal{D}_2, \mathsf{d})$				
(*CltCode*)	A	$::=$	$\{\mathsf{f}_1 \rightsquigarrow \mathit{cfd}_1, \ldots, \mathsf{f}_n \rightsquigarrow \mathit{cfd}_n\}$				

图 4.1　应用语言

图4.1给出了应用语言的定义，应用程序 A 可以理解为应用层程序员写的函数的集合，它将函数名 f 映射到函数定义 *cfd*。函数定义 *cfd* 包含四个部分，分别是返回值类型 τ，参数声明列表 $\mathcal{D}_1$，局部变量声明列表 $\mathcal{D}_2$ 以及函数体 d。

τ 一共包含 8 种类型，包括 NULL 指针的类型 `Tnull`；void 类型 `Tvoid`；字符类型 `Tint8`；短整数类型 `Tint16`；整数类型 `Tint32`；指针类型 $\mathtt{Tptr}(\tau)$，这里 τ 是被指向的数据的类型；数组类型 $\mathtt{Tarray}(\tau, \mathsf{k})$，这里 τ 是数组元素的类型，k 是数组的长度；结构体类型 $\mathtt{Tstruct}(\mathsf{id}, \mathcal{D})$，这里 id 是结构体的名字，$\mathcal{D}$ 包含结构体所有成员变量的名字和类型。为了定义链表之类的数据结构，C

$$
\begin{array}{lll}
(\mathit{Cont}) & \mathsf{K} ::= (\kappa_e, \kappa_s) & (\mathit{ExprCont})\ \kappa_e ::= \circ \mid \ldots \\
(\mathit{StmtCont}) & \kappa_s ::= \bullet \mid \mathsf{d} \cdot \kappa_s \mid (\mathsf{c}, \kappa_e, \mathsf{E}) \cdot \kappa_s \mid (\mathsf{f}, \bar{v}, \bar{e}) \cdot \kappa_s \mid (\mathsf{f}\ \mathsf{s}\ \mathsf{E}) \cdot \kappa_s \mid \ldots \\
(\mathit{CurEval}) & \mathsf{c} ::= e \mid \mathsf{d} \mid \textbf{\textit{fexec}}(\mathsf{f}, \bar{v}) \mid \textbf{\textit{alloc}}(\bar{v}, \mathcal{D}) \mid \textbf{skip} \mid v \mid \ldots \\
(\mathit{TaskCode}) & \mathsf{C} ::= (\mathsf{c}, \mathsf{K}) & (\mathit{CMem}) \quad \mathfrak{m} ::= (\mathsf{G}, \mathsf{E}, \mathsf{M})
\end{array}
$$

图 4.2　C 程序状态

语言的很多结构体定义时包含了指向自己的指针，为了支持这种特性，本文采用和 CompCert 相同的处理方法，定义了一个特殊的类型 `Tcomptr(id)`，用于定义结构体 id 时表示指向 id 自己的指针。

$\mathcal{D}$ 是一个列表其包含了一些变量或参数的声明，每个声明包含变量的名称和类型。验证框架支持大多数的 C 语言的语句，d 可以是赋值语句 $e = e$，函数调用语句 $\mathsf{f}(\bar{e})$、$e = \mathsf{f}(\bar{e})$，条件分支语句 **if** (e) **then** d **else** d，循环语句 **while** (e) d 等等。注意这里还引入了逻辑语句 **print** e，这是为了构建可见事件精化关系，后面会具体介绍。应用程序在执行函数调用时，其可能调用 A 中的函数，也可能是调用内核代码，对应在高层机器上还可能是调用规范代码。

表达式 e 可以是一个程序变量 x，也可能是常量表达式 k，解引用表达式 $*e$，取地址表达式 $\&e$，结构体成员变量表达式 $e.\mathtt{id}$，数组成员表达式 $e[e]$，一元运算表达式 `uop` e 和二元运算表达式 e `bop` e。其中一元运算包括取反，按位取反等等，二元运算包括加、减、与、或、按位与等等。

目前验证框架中的 C 语言子集不包含 **break** 语句，**for** 循环语句，**do while** 语句。一方面是因为验证的内核代码中目前没有使用这些 C 语言语句，另一方面这些代码都可以简单的改写为 while 循环。因为系统内核中一般没有浮点运算，所以验证框架目前不考虑浮点运算。

C 语言程序状态　图4.2给出了这些 C 程序状态的定义。底层机器模型和高层机器模型共用一个相同的 C 语言子集，C 语言子集相关的程序状态在底层和高层是相同的。本文采用后继（continuation）风格的操作语义，任务代码 C 包含两个部分，当前正在执行的部分 c 和程序后继 K。由于采用细粒度的小步操作语义，每个语句和表达式的计算都是分成多步完成的。程序后继 K 用来记录目前执行到语句或表达式的哪个位置同时保存还未执行的代码。K 分为两个部分，分别是语句后继 κ_s 和表达式后继 κ_e。$\mathfrak{m}$ 是 C 语言代码需要访问的程序状态，包括全局符号表、局部符号表和内存。

图 4.3给出了内存和符号表的形式化定义。内存 M 是一个从地址 l 到内存值 v 的部分函数。地址是由内存块号 b 和块内偏移 i 组成的二元组。这里用正整数来表示内存块号，用整数来表示偏移。本文中的内存模型和 CompCert [35] 类似，但是为了支持可组合验证，内存分配操作是非确定的，后面会具体介绍。

$$
\begin{array}{llll}
(Nat) & \mathtt{k} & \in & \mathtt{nat} \quad (Offset)\ \mathtt{i} \in \mathbb{Z} \\
(Byte) & \mathtt{bt} & \in & \mathtt{byte} \quad (Int32)\ \mathfrak{n} \in \mathtt{int32} \\
(Block) & \mathfrak{b} & \in & \mathtt{positive} \\
(Addr) & \mathfrak{l} & \in & Block \times Offset \\
(MemVal) & \mathfrak{v} & ::= & \mathtt{undef} \mid \mathtt{bt} \mid \mathsf{PtrByte}(\mathfrak{b}, \mathfrak{i}, \mathtt{k}) \mid \mathsf{null} \\
(MValList) & \mathbb{V} & ::= & \mathtt{nil} \mid \mathfrak{v} :: \mathbb{V} \\
(Var) & \mathfrak{x}, \mathfrak{y}, \ldots & \in & \mathbb{Z} \\
(Memory) & \mathcal{M} & \in & Addr \rightharpoonup MemVal \\
(SymTable) & \mathcal{G}, \mathcal{E} & \in & Var \rightharpoonup Block \times Type
\end{array}
$$

图 4.3　内存和符号表定义

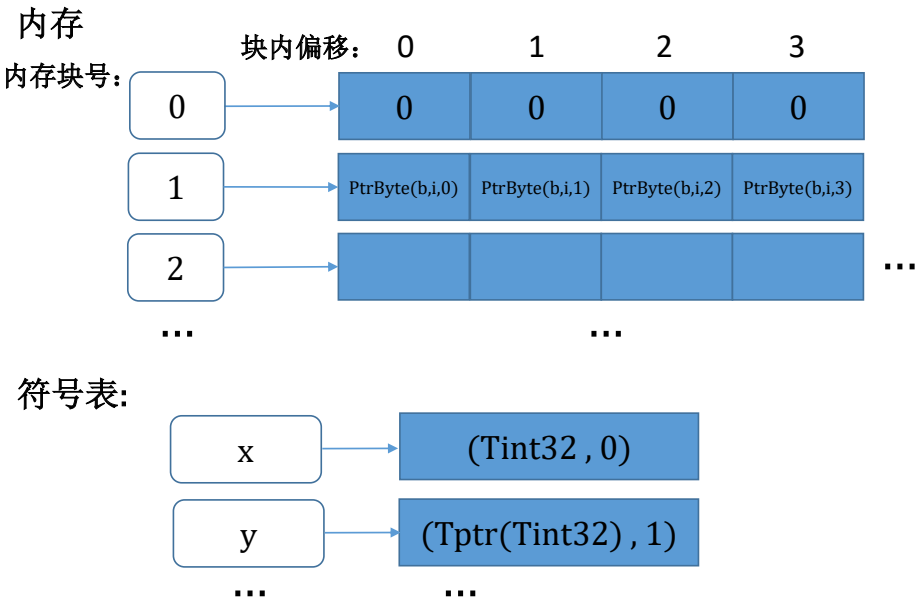

图 4.4　内存模型

内存值 $\mathfrak{v}$ 表示保存在每个内存单元（Byte）内的值。$\mathfrak{v}$ 可能是一个 8 位的整数；可能是指针的某个部分 $\mathsf{PtrByte}(\mathfrak{b}, \mathfrak{i}, \mathtt{k})$，指针类型一共消耗 4 个字节，(b,i) 是该指针指向的地址，$\mathtt{k}$ 表示该字节是指针的第几部分；可能是一个特殊的常数值 null 表示 NULL 指针，NULL 指针一共消耗 4 字节，每个字节的内存值都是 null；还可能是 `undef` 表示未定义的内存。$\mathbb{V}$ 表示一组内存值。

符号表保存变量到其类型和内存块号的映射，由于每个程序变量都单独占有一块内存，所以这里只要记录内存块号 $\mathfrak{b}$，变量在内存中的起始地址就是 $(\mathfrak{b}, 0)$。符号表分为全局符号表 $\mathcal{G}$ 和局部符号表 $\mathcal{E}$，分别保存全局变量和局部变量的信息，这里用整数来表示变量名。

$$\mathsf{fresh}(M,b) \stackrel{\text{def}}{=} \forall l,i.\ l=(b,i) \Longrightarrow l \notin \mathrm{dom}(M)$$

$$\mathsf{FullMemory}(M) \stackrel{\text{def}}{=} \forall b.\ \exists i.\ (b,i) \in \mathrm{dom}(M)$$

$$\mathsf{allocb}(M,b,i,k) \stackrel{\text{def}}{=} \begin{cases} M & \text{if} \quad k=0 \\ M'\{(b,i)\mapsto \mathtt{undef}\} & \text{if} \quad M'=\mathsf{allocb}(M,b,i{+}1,k')\wedge \\ & \qquad k=k'{+}1 \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{freeb}(M,b,i,k) \stackrel{\text{def}}{=} \begin{cases} M & \text{if} \quad k=0 \\ M'\setminus\{(b,i)\mapsto v\} & \text{if} \quad M(b,i)=v\wedge k=k'{+}1\wedge \\ & \qquad M'=\mathsf{freeb}(M,b,i{+}1,k') \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{storebytes}(M,l,\mathbb{V}) \stackrel{\text{def}}{=} \begin{cases} M & \text{if} \quad \mathbb{V}=\mathtt{nil} \\ M'\{l\mapsto v\} & \text{if} \quad l\in\mathrm{dom}(M)\wedge l=(b,i)\wedge \mathbb{V}=v::\mathbb{V}'\wedge \\ & \qquad M'=\mathsf{storebytes}(M,(b,i{+}1),\mathbb{V}') \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{loadbytes}(M,l,k) \stackrel{\text{def}}{=} \begin{cases} \mathtt{nil} & \text{if} \quad k=0 \\ v::\mathbb{V}' & \text{if} \quad M(l)=v\wedge l=(b,i)\wedge k=k'{+}1\wedge \\ & \qquad \mathbb{V}'=\mathsf{loadbytes}(M,(b,i{+}1),k') \\ \bot & \text{otherwise} \end{cases}$$

$$M_1 \perp M_2 \stackrel{\text{def}}{=} \mathrm{dom}(M_1)\cap\mathrm{dom}(M_2)=\emptyset$$

图 4.5　内存上的基本操作和性质定义

为了方便理解，图4.4 给出了内存结构的示意图，图中有两个变量 x 和 y，x 的类型是 `Tint32`，保存在 0 号内存块中，其占用 4 个字节的空间，每个字节的内容都是 0。变量 y 是一个指向地址 (b,i) 的指针，其值保存在 1 号内存块中，其同样占有 4 个字节的空间。

图4.5给出了内存上的基本操作和一些性质定义。$\mathsf{fresh}(M,b)$ 表示 M 中的内存块 b 是未被分配的。$\mathsf{FullMemory}(M)$ 表示内存满了，也即 M 中所有的内存块都被分配了。如果内存满了分配操作会失败，本文假设内存是足够的，内存分配不会失败。$\mathsf{allocb}(M,b,i,k)$ 用来在 M 中分配一块 k 个字节的内存，起始地址为 (b,i)，并返回分配结束后的新的内存。$\mathsf{freeb}(M,b,i,k)$ 用来释放 M 中 k 个字节的内存，起始地址为 (b,i)，并返回释放后的新的内存，这里要求每个字节都是分配过的，否则报错。$\mathsf{storebytes}(M,l,\mathbb{V})$ 用来更新内存 M 中的从起始地址 l 开始的内存值（MemVal），直到 $\mathbb{V}$ 内的所有内容都被写入。$\mathsf{loadbytes}(M,l,k)$ 用来读取 M 中 l 开始的 k 个字节，返回读取出来的一组内存值 $\mathbb{V}$，如果读取了未分配的内存则出错。

$$
|\tau| \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if} \quad \tau = \texttt{Tint8} \\ 2 & \text{if} \quad \tau = \texttt{Tint16} \\ 4 & \text{if} \quad \tau \in \{\texttt{Tnull}, \texttt{Tint32}, \texttt{Tptr}(\tau'), \\ & \qquad \texttt{Tcomptr}(\mathsf{id}), \texttt{Tvoid}\} \\ |\tau'| \times k & \text{if} \quad \tau = \texttt{Tarray}(\tau', k) \\ \mathsf{szstruct}(\mathcal{D}) & \text{if} \quad \tau = \texttt{Tstruct}(\_, \mathcal{D}) \end{cases}
$$

$$
\mathsf{szstruct}(\mathcal{D}) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if} \quad \mathcal{D} = \mathtt{nil} \\ |\tau| + \mathsf{szstruct}(\mathcal{D}') & \text{if} \quad \mathcal{D} = (\mathsf{id}, \tau) :: \mathcal{D}' \end{cases}
$$

图 4.6　计算类型长度

$$
(\mathit{Val}) \quad v \ ::= \ \texttt{Vundef} \mid \texttt{Vnull} \mid \texttt{Vint}(n) \mid \texttt{Vptr}(a)
$$

$$
\begin{aligned}
&\mathsf{alloc}(x, \tau, E, M, E', M') \stackrel{\text{def}}{=} \exists b.\, \mathsf{fresh}(M, b) \wedge M' = \mathsf{allocb}(M, b, 0, |\tau|) \wedge \\
&\qquad\qquad x \notin \mathrm{dom}(E) \wedge E' = E\{x \mapsto (b, \tau)\} \\
&\mathsf{free}(\tau, b, M) \stackrel{\text{def}}{=} \mathsf{freeb}(M, b, 0, |\tau|) \\
&M\{l \overset{\tau}{\rightsquigarrow} v\} \stackrel{\text{def}}{=} \mathsf{storebytes}(M, l, \mathsf{encode}(\tau, v)) \\
&\mathsf{load}(\tau, M, l) \stackrel{\text{def}}{=} \begin{cases} \mathsf{decode}(\tau, \mathbb{V}) & \text{if} \quad \mathbb{V} = \mathsf{loadbytes}(M, l, |\tau|) \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

图 4.7　结合类型的内存操作

C 语言中表达式计算出的值（*Val*）分为四种（定义在图4.7中），分别是整数值 $\texttt{Vint}(n)$，指针值 $\texttt{Vptr}(a)$，未定义内存值 Vundef，还有空指针值 Vnull。

因为表达式的值（*Val*）和内存值（*MemVal*）不同，内存值是以字节为单位的，而表达式的值（*Val*）占用多少字节是取决于表达式类型的。和 CompCert 类似，本文定义了两个函数 encode 和 decode 用于实现 *Val* 和 *MemVal* 之间的转换。例如，图4.4中变量 x，其值（*Val*）为 $\texttt{Vint}(0)$，类型为 Tint32，encode 函数会将其转化为一组内存值（*MemVal*）$(0 :: 0 :: 0 :: 0 :: \mathtt{nil})$；如果 x 的类型是 Tint16，那么 encode 函数会将其转化为 $(0 :: 0 :: \mathtt{nil})$。对于变量 y，其值（*Val*）为 $\texttt{Vptr}(b, i)$，类型为 $\texttt{Tptr}(\texttt{Tint32})$，encode 函数会将其转化为一组内存值（$\mathsf{PtrByte}(b, i, 0) :: \mathsf{PtrByte}(b, i, 1) :: \mathsf{PtrByte}(b, i, 2) :: \mathsf{PtrByte}(b, i, 3) :: \mathtt{nil}$）。decode 是 encode 的逆过程。encode 和 decode 的具体定义这里不再展开介绍，具体可以看 Coq 代码 [3] 中的“CertiOS/framework/machine/memory/memdata.v”文件。

为了方便定义操作语义，图4.7还给出了一些封装后的内存操作。其中

$$\boxed{A \vdash (C, m) \longmapsto (C', m')}$$

$$\frac{}{A \vdash ((\mathtt{f}(\mathtt{nil}), (\circ, \kappa_s)), m) \longmapsto ((\textbf{\textit{fexec}}(\mathtt{f}, \mathtt{nil}), (\circ, \kappa_s)), m)}\ (\textsc{Fn})$$

$$\frac{}{A \vdash ((\mathtt{f}(e :: \bar{e}), (\circ, \kappa_s)), m) \longmapsto ((e, (\circ, (\mathtt{f}, \mathtt{nil}, \bar{e}) \cdot \kappa_s)), m)}\ (\textsc{Fa})$$

$$\frac{}{A \vdash ((v, (\circ, (\mathtt{f}, \bar{v}, (e :: \bar{e})) \cdot \kappa_s)), m) \longmapsto ((e, (\circ, (\mathtt{f}, (v :: \bar{v}), \bar{e}) \cdot \kappa_s)), m)}\ (\textsc{Feval})$$

$$\frac{}{A \vdash ((v, (\circ, (\mathtt{f}, \bar{v}, \mathtt{nil}) \cdot \kappa_s)), m) \longmapsto ((\textbf{\textit{fexec}}(\mathtt{f}, v :: \bar{v}), (\circ, \kappa_s)), m)}\ (\textsc{FEnter})$$

$$\frac{\begin{array}{c} A(\mathtt{f}) = (\tau, \mathcal{D}_1, \mathcal{D}_2, s) \qquad m = (G, E, M) \\ m' = (G, \emptyset, M) \qquad \mathcal{D}' = \mathsf{rev}(\mathcal{D}_1) +\!+ \mathcal{D}_2 \end{array}}{A \vdash ((\textbf{\textit{fexec}}(\mathtt{f}, \bar{v}), (\circ, \kappa_s)), m) \longmapsto ((\textbf{\textit{alloc}}(\bar{v}, \mathcal{D}'), (\circ, (\mathtt{f}, s, E) \cdot \kappa_s)), m')}\ (\textsc{FAlloc})$$

$$\frac{m = (G, E, M) \quad \mathsf{alloc}(x, E, M, E', M', \tau) \quad m' = (G, E', M')}{A \vdash (\textbf{\textit{alloc}}(\mathtt{nil}, (x, \tau) :: \mathcal{D}), \kappa_s, m) \longmapsto (\textbf{\textit{alloc}}(\mathtt{nil}, \mathcal{D}), \kappa_s, m')}\ (\textsc{FAllocL})$$

$$\frac{\begin{array}{c} m = (G, E, M) \quad \mathsf{alloc}(x, E, M, E', M', \tau) \quad E'(x) = (b, \tau) \\ M'\{(b, 0) \overset{\tau}{\rightsquigarrow} v\} = M'' \quad m' = (G, E', M'') \end{array}}{A \vdash (\textbf{\textit{alloc}}(v :: \bar{v}, (x, \tau) :: \mathcal{D}), \kappa_s, m) \longmapsto (\textbf{\textit{alloc}}(\bar{v}, \mathcal{D}), \kappa_s, m')}\ (\textsc{FAllocP})$$

$$\frac{}{A \vdash ((\textbf{\textit{alloc}}(\mathtt{nil}, \mathtt{nil}), (\circ, (\mathtt{f}, s, E) \cdot \kappa_s)), m) \longmapsto ((s, (\circ, (\mathtt{f}, s, E) \cdot \kappa_s)), m)}\ (\textsc{FBody})$$

图 4.8　C 操作语义

$\mathsf{alloc}(x, \tau, E, M, E', M')$ 表示给变量 x 分配内存，其类型为 τ，当前的符号表和内存分别为 E 和 M，分配结束后符号表和内存分别变为 E' 和 M'。为了满足分离逻辑对操作语义局部性（locality）的要求，内存分配操作是非确定的（non-deterministic），分配时会找任意一块空闲的内存块。$\mathsf{free}(\tau, b, M)$ 表示释放 M 中内存块 b 上长度为 $|\tau|$ 的空间。$M\{l \overset{\tau}{\rightsquigarrow} v\}$ 表示将值 v 经过 encode 转化后写入内存 M，起始地址为 l，并返回写入过后的新的内存。$\mathsf{load}(\tau, M, l)$ 表示读内存 M 中起始地址为 l，长度为 $|\tau|$ 的内存，并返回一个经过 decode 转化后的值。$|\tau|$ 表示类型 τ 的值占用的内存字节数，其定义在图4.6中。

操作语义　如图4.8 所示，“$A \vdash (C, m) \longmapsto (C', m')$” 用来定义 C 语言子集的操作语义。这里用函数调用相关的规则 Fn、Fa、Feval、FEnter、FAlloc 、FAllocL 、FAllocP 和 FBody 为例来介绍。当调用一个函数 f 时，首先通过 Fa 规则取出实参表达式并计算它的值；计算完成后 Feval 规则用于将计算好的值保存起来，并取出下一个实参继续计算；完成所有实参的值的计算后 FEnter 规则会保存调用函数（caller）的一些状态上下文（包括语句后继和局部符号表）并进入被调用函数（callee）f 执行，这里用一个动态语句 $\textbf{\textit{fexec}}(\mathtt{f}, \bar{v})$ 来作为调

用函数和被调用函数的边界。当进入被调用函数后，FAllOC 规则表示开始给参数和局部变量分配内存和初始化；FAllOCP规则用于分配形参的内存，并将实参的值写入形参。FAllOCL规则用于局部变量的内存分配；在完成了参数和局部变量的初始化后，会应用 FBODY 规则进入被调用函数的函数体执行。FN 是被调用函数 f 没有参数时使用的规则。完整的 C 语言部分操作语义在附录A。

4.1.2　内核语言

$$
\begin{array}{llll}
(\mathit{LPrim}) & \iota & ::= & \textbf{switch}\ x \mid \textbf{encrt} \mid \textbf{excrt} \mid \textbf{eoi}\ k \mid \textbf{iext} \mid \textbf{cli} \mid \textbf{sti} \\
(\mathit{LStmts}) & s & ::= & d \mid \iota \mid s; s \mid \textbf{while}\ (e)\ s \mid \textbf{if}\ (e)\ \textbf{then}\ s\ \textbf{else}\ s \\
(\mathit{ItrpCode}) & \theta & ::= & [s_0, \ldots, s_{N-1}] \qquad (\mathit{LFunDef})\quad \mathit{ofd} ::= (\tau, \mathcal{D}_1, \mathcal{D}_2, s) \\
(\mathit{ProgUnit}) & \eta & ::= & \{f_1 \rightsquigarrow \mathit{ofd}_1, \ldots, f_n \rightsquigarrow \mathit{ofd}_n\} \\
(\mathit{LOSCode}) & O & ::= & (\eta_a, \eta_i, \theta) \qquad (\mathit{LProg}) \quad P ::= (A, O)
\end{array}
$$

图 4.9　内核语言

内核语言　图4.9给出了内核语言的定义。运行在底层机器上的程序 P 一共由两部分组成，分别是应用语言实现的应用代码 A 和由内核语言实现的系统内核代码 O。内核代码 O 一共包含三个部分：一个是提供给用户使用的系统 API 代码 η_a，一个是中断处理程序的集合 θ，还有用户不可见的内部函数 η_i。θ 是从中断向量号到中断处理程序的映射，系统中断个数是系统初始化时确定的，记为 N，并且中断向量号越低，其对应的中断优先级越高。η 是从函数名到函数实现 *ofd* 的映射，和应用语言类似，函数定义 *ofd* 包括函数返回值类型，函数参数声明列表，函数局部变量列表，以及函数体。这里函数体 s 可能是和应用语言一样的 C 语言子集中的语句 d，也可能是汇编原语 ι，或者是一些其他的组合语句。在本文 Coq 实现中其实并没有区分 *ofd* 和 *cfd* 的定义，函数定义都是使用 *ofd*，然后通过语法检查要求应用代码中不能使用汇编原语，这样可以省去很多操作语义上类似性质的证明。

汇编原语 **switch** x 表示任务切换，x 是一个程序变量，保存了要切换到的任务 ID。**encrt** 会关闭中断，表示进入临界区，为了支持嵌套临界区，**encrt** 还会将关中断之前 IF 位保存到栈上，因为在底层机器模型中抽象掉了栈，所以这里单独用一个状态栈 *cs* 来保存进入临界区的历史信息（*cs* 的定义在图4.10中，后面会具体解释）。注意在底层机器模型中，用 *ie* 来表示 IF 位，并抽象掉 EFLAGS 寄存器的其他位。**excrt** 会弹栈并赋值给 *ie*, 表示退出当前临界区。**eoi** k 会将 *isr* 的第 k 位清 0，表示 k 号中断不再被服务。**iext** 表示中断退出，回到被中断的程序继续执行并开中断。**sti** 和 **cli** 用于开关中断，但不会进行 IF 的保存和恢复，只是简单的修改 *ie* 的值。

$$
\begin{array}{llll}
(\mathit{BitVal}) & \mathtt{b}, \mathit{ie} \in \{0,1\} & (\mathit{ISRReg}) & \mathit{isr} ::= [\mathtt{b}_0, \ldots, \mathtt{b}_{N-1}] \\
(\mathit{CrtStk}) & \mathit{cs} ::= \mathtt{nil} \mid \mathit{ie} :: \mathit{cs} & (\mathit{ItrpStk}) & \mathit{is} ::= \mathtt{nil} \mid \mathtt{k} :: \mathit{is} \\
(\mathit{ItrpTaskSt}) & \delta ::= (\mathit{ie}, \mathit{is}, \mathit{cs}) & (\mathit{ItrpSt}) & \pi ::= \{\mathtt{t}_1 \rightsquigarrow \delta_1, \ldots, \mathtt{t}_n \rightsquigarrow \delta_n\} \\
(\mathit{CState}) & \Delta ::= (G, \Pi, M) & (\mathit{SymTblSet}) & \Pi ::= \{\mathtt{t}_1 \rightsquigarrow E_1, \ldots, \mathtt{t}_n \rightsquigarrow E_n\} \\
(\mathit{LOsFullSt}) & \Lambda ::= (\Delta, \mathit{isr}, \pi) & (\mathit{TaskLocalSt}) & \sigma ::= (\mathtt{m}, \mathit{isr}, \delta) \\
(\mathit{LWorld}) & W ::= (T, \Delta, \Lambda, \mathtt{t}) & (\mathit{TaskPool}) & T ::= \{\mathtt{t}_1 \rightsquigarrow C_1, \ldots, \mathtt{t}_n \rightsquigarrow C_n\}
\end{array}
$$

图 4.10　底层程序状态

底层机器状态　图4.10给出了完整的底层机器的定义。底层机器 W 一共包含四个部分，分别是任务池 T，用户程序状态 Δ，内核程序状态 Λ 和当前任务号 $\mathtt{t}$。底层机器模型支持多任务，任务池 T 是从任务号到任务代码的映射。

因为每个任务都需要有自己的局部符号表，所以用户程序状态，包含三个部分：所有任务共享的全局符号表；每个任务的任务号到自己局部符号表的映射；以及所有任务共享的一块内存。

内核程序状态包含三个部分，C 语言程序状态 Δ（和用户程序状态类型相同），所有任务共享的 *isr* 寄存器状态，中断状态集合 π，本文假定应用代码和内核代码访问的程序状态是隔离的，应用程序只能通过调用 O 中的系统 API 来访问系统状态，而系统代码 O 无法访问用户程序状态。

isr 是一个全局的寄存器，如果 *isr* 的某一位 $\mathtt{i}$ 为 1，则表示 $\mathtt{i}$ 号中断正在被服务，如果 $\mathtt{i}$ 位为 0 表示该中断并没有被服务。π 是从任务 ID 到该任务的中断状态的映射。任务中断状态 δ 是一个三元组。*ie* 记录了该任务当前是否允许中断，如果 $\mathit{ie} = 1$ 表示允许中断，$\mathit{ie} = 0$ 表示不允许中断。*cs* 如前所述是为了支持嵌套临界区而保存的临界区历史信息的栈。*is* 是记录中断嵌套路径的栈，这里要注意 *is* 并没有真实的机器状态对应，而是为了验证引入的辅助状态。

由于中断的发生和 CPU 执行是并发的，中断请求可以在任意时刻发生，如果中断打开（$\mathtt{ie} = 1$），并且没有更高优先级的中断请求，则 CPU 在当前指令结束后会响应中断（假设为 $\mathtt{i}$ 号中断）。中断被响应后，*isr* 的 $\mathtt{i}$ 位会被置 1，表示该中断正被服务，*ie* 会被置 0，同时会将 $\mathtt{i}$ 压入栈 *is* 来记录中断路径，同时保存局部符号表和代码上下文以便中断退出时恢复，最后跳转到中断处理程序执行。

为了方便，这里还定义了任务内核状态 σ，它由三个部分组成，分别是该任务的内核 C 程序状态 $\mathtt{m}$，*isr* 寄存器以及该任务的中断状态 δ。

4.1.3　底层操作语义

为了保证中断和并发的语义的正确性，程序语句的执行和表达式计算都必须是和机器指令相同的粒度，这里采用和 CompCertTSO [48, 53] 类似的小步操作语义，但由于只考虑单核，所以并没有使用 TSO 的内存模型。为了结构清晰，

$$\lfloor \kappa_s \rfloor_c \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if} \quad \kappa_s = \bullet \\ \kappa_s & \text{if} \quad \kappa_s = (f, s, E) \cdot \kappa_s' \\ \lfloor \kappa_s' \rfloor_c & \text{otherwise} \end{cases} \qquad \lfloor \kappa_s \rfloor \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if} \quad \kappa_s = \bullet \\ \kappa_s & \text{if} \quad \kappa_s = (c, \kappa_e, E) \cdot \kappa_s' \\ \lfloor \kappa_s' \rfloor & \text{otherwise} \end{cases}$$

$$f \perp g \stackrel{\text{def}}{=} dom(f) \cap dom(g) = \emptyset \qquad f \uplus g \stackrel{\text{def}}{=} \begin{cases} f \cup g & \text{if} \quad f \perp g \\ \mathit{undef} & \text{otherwise} \end{cases}$$

$$(G, \Pi, M)|_t = (G, E, M) \stackrel{\text{def}}{=} \Pi(t) = E$$

$$(\Delta, \mathit{isr}, \pi)|_t = (m, \mathit{isr}, \delta) \stackrel{\text{def}}{=} \Delta|_t = m \wedge \pi(t) = \delta$$

$$\Lambda' = \textsc{updTS}(\Lambda, t, \sigma') \stackrel{\text{def}}{=} \Lambda'|_t = \sigma' \wedge \forall t' \neq t. \Lambda'|_{t'} = \Lambda|_{t'}$$

$$\Delta' = \textsc{updCS}(\Delta, t, m') \stackrel{\text{def}}{=} \Delta'|_t = m' \wedge \forall t' \neq t. \Delta'|_{t'} = \Delta|_{t'}$$

$$[\![x]\!]_{(G,E,M)} = t' \stackrel{\text{def}}{=} \exists a.\ (E(x) = a \wedge M(a) = t') \vee (G(x) = a \wedge x \notin dom(E) \wedge M(a) = t')$$

$$[\![x]\!]_{(t,((G,\Pi,M),\mathit{isr},\pi))} = t' \stackrel{\text{def}}{=} [\![x]\!]_{(G,\Pi(t),M)} = t'$$

$$((G, \Pi, M'), \mathit{isr}, \pi) = \textsc{updG}(((G, \Pi, M), \mathit{isr}, \pi), \texttt{OSTCBCur}, t') \stackrel{\text{def}}{=} \exists b, \tau.\ G(\texttt{OSTCBCur}) = (b, \tau) \wedge M' = M\{(b, 0) \overset{\tau}{\rightsquigarrow} t'\}$$

$$\mathsf{InOS}(C, (A, O)) \stackrel{\text{def}}{=} \exists c, \kappa_s.\ C = (c, (\_, \kappa_s)) \wedge ((\exists f. f \in dom(O.\eta_a \uplus O.\eta_i) \wedge (c = \textbf{\textit{fexec}}(f, \_) \vee \lfloor \kappa_s \rfloor_c = (f, \_, \_) \cdot \_)) \vee \lfloor \kappa_s \rfloor \neq \bot)$$

图 4.11　底层操作语义相关的辅助定义

$\boxed{P \vdash W =\!\!\text{L}\Rightarrow W'}$

$$\frac{\begin{array}{c} \Lambda|_t = \sigma \quad P \vdash (C, \sigma, m) -\!\!\text{L}\!\!\rightarrow (C', \sigma', m') \\ T(t) = C \quad T' = T\{t \rightsquigarrow C'\} \quad \Lambda' = \textsc{updTS}(\Lambda, t, \sigma') \\ \Delta|_t = m \quad \Delta' = \textsc{updCS}(\Delta, t, m') \end{array}}{P \vdash (T, \Delta, \Lambda, t) =\!\!\text{L}\Rightarrow (T', \Delta', \Lambda', t)} \ (\textsc{Ptask})$$

$$\frac{T(t) = (\textbf{switch}\ x, K) \quad [\![x]\!]_{(t,\Lambda)} = t' \quad \Lambda' = \textsc{updG}(\Lambda, \texttt{OSTCBCur}, t')}{P \vdash (T, \Delta, \Lambda, t) =\!\!\text{L}\Rightarrow (T\{t \rightsquigarrow (\textbf{skip}, K)\}, \Delta, \Lambda', t')} \ (\textsc{Psw})$$

$$\frac{\begin{array}{c} P = (A, (\eta_a, \eta_i, \theta)) \quad T(t) = (c, (\kappa_e, \kappa_s)) \\ \Lambda|_t = ((G, E, M), \mathit{isr}, (1, \mathit{is}, \texttt{nil})) \quad \forall k'. k' \leqslant k \rightarrow \mathit{isr}(k') = 0 \\ C' = (\theta(k), (\circ, (c, \kappa_e, E) \cdot \kappa_s)) \quad T' = T\{t \rightsquigarrow C'\} \\ \sigma' = ((G, \emptyset, M), \mathit{isr}\{k \rightsquigarrow 1\}, (0, k :: \mathit{is}, \texttt{nil})) \quad \Lambda' = \textsc{updTS}(\Lambda, t, \sigma') \end{array}}{P \vdash (T, \Delta, \Lambda, t) =\!\!\text{L}\Rightarrow (T', \Delta, \Lambda', t)} \ (\textsc{Pitrp})$$

图 4.12　底层全局操作语义

底层机器的操作语义被分为四层，分别是全局操作语义、任务操作语义、内核操作语义和 C 操作语义。C 操作语义在图4.8已经介绍过了，这里不再赘述。

图4.12给出了底层全局操作语义的定义，全局操作语义执行一步 $P \vdash W =\!\!\Rightarrow W'$ 可能是当前任务执行一步（PTASK），也可能是执行任务切换（PSW），或者被中断打断（PITRP）。本文不考虑自修改代码，程序 P 在执行的过程中不会发生变化，自修改代码是指程序在执行的过程中可能改变正在执行的代码本身，经常被用于防止被逆向工程获得源码、实时代码生成等等。

PTASK 规则给出了当前任务执行一步对全局状态的改变，$\Lambda|_t$ 表示从系统内核状态 Λ 中获取当前任务 t 的内核状态，$\text{UPDTS}(\Lambda, t, \sigma')$ 用当前任务执行一步后的状态 σ' 来更新 Λ 中对应的部分。$\Delta|_t$ 表示从全局用户状态 Δ 中获取当前任务的用户状态 m，$\text{UPDCS}(\Delta, t, m')$ 用当前任务执行一步后的用户状态 m' 来更新 Δ 中对应的部分。这些辅助符号的定义在图4.11中。

用于实现上下文切换的汇编代码被抽象为一个汇编原语 **switch** x，x 中保存将要切换到的任务号，`OSTCBCur` 是保存正在运行的任务的全局变量，$\text{UPDG}(\Lambda, \texttt{OSTCBCur}, t')$ 表示用新的任务号 t' 来更新全局变量 `OSTCBCur`（定义在图4.11中）。PSW规则简单来说就是通过变量 x 的内容来更新保存当前任务的程序状态以及保存当前任务的全局变量 `OSTCBCur`。

PITRP规则给出了当前任务被中断 k 打断的语义，规则要求被打断之前 *ie* 为 1（允许中断）且 *cs* 为 `nil`（不在临界区中），而且没有更高优先级的中断正被服务（$\forall k'.k' \leqslant k \rightarrow \mathit{isr}(k') = 0$）。当前任务被中断后 *ie* 会被设置为 0，表示中断处理程序开始默认中断关闭，并将中断号 k 压入 *is* 栈（*is* 栈是用来保存中断路径的逻辑状态），同时更新 *isr* 中的第 k 位，表示 k 号中断正被服务。由于中断改变了程序的执行路径，中断发生时需要保存上下文，包括当前正在执行的内容 c、表达式后继 κ_e 和局部符号表 E，这些信息被保存在语句后继中。

$$\boxed{P \vdash (C, m, \sigma) \text{—}\text{L}\!\!\rightarrow (C', m', \sigma')}$$

$$\frac{\mathsf{InOS}(C, P) \quad P \vdash (C, \sigma) \bullet\!\text{—}\text{L}\!\!\triangleright (C', \sigma')}{P \vdash (C, m, \sigma) \text{—}\text{L}\!\!\rightarrow (C', m, \sigma')} \ (\text{TKERNEL})$$

$$\frac{\neg\mathsf{InOS}(C, (A, (\eta_a, \eta_i, \theta))) \qquad A \uplus \eta_a \vdash (C, m) \longmapsto (C', m')}{(A, (\eta_a, \eta_i, \theta)) \vdash (C, m, \Lambda) \text{—}\text{L}\!\!\rightarrow (C', m', \Lambda)} \ (\text{TCLT})$$

图 4.13　底层任务操作语义

图4.13给出了底层机器任务操作语义。每个任务在执行时都存在两种情况，可能是在执行应用代码，也可能正在执行内核代码。当执行应用代码时访问的是该任务的用户状态 m，执行的语句是 C 语言的子集；当执行内核代码时访问的是该任务的内核状态 σ，执行的是 C 语言的子集和汇编原语。TKERNEL 是

$$\boxed{\mathsf{P} \vdash (\mathsf{C}, \sigma) \bullet\!\!-\!\!\mathrm{L}\!\!\triangleright (\mathsf{C}', \sigma')}$$

$$\frac{\sigma = (m, isr, (ie, is, cs)) \quad \sigma' = (m, isr, (0, is, ie :: cs))}{\mathsf{P} \vdash ((\mathbf{encrt}, \mathsf{K}), \sigma) \bullet\!\!-\!\!\mathrm{L}\!\!\triangleright ((\mathbf{skip}, \mathsf{K}), \sigma')} \ (\textsc{EnterCrt})$$

$$\frac{\sigma = (m, isr, (ie, is, ie' :: cs)) \quad \sigma' = (m, isr, (ie', is, cs))}{\mathsf{P} \vdash ((\mathbf{excrt}, \mathsf{K}), \sigma) \bullet\!\!-\!\!\mathrm{L}\!\!\triangleright ((\mathbf{skip}, \mathsf{K}), \sigma')} \ (\textsc{ExitCrt})$$

$$\frac{0 \leqslant k < N \quad \sigma = (m, isr, (ie, is, cs)) \quad \sigma' = (m, isr\{k \rightsquigarrow 0\}, (ie, is, cs))}{\mathsf{P} \vdash ((\mathbf{eoi}\ k, \mathsf{K}), \sigma) \bullet\!\!-\!\!\mathrm{L}\!\!\triangleright ((\mathbf{skip}, \mathsf{K}), \sigma')} \ (\textsc{Eoi})$$

$$\frac{\begin{array}{c}\sigma = ((\mathsf{G}, \mathsf{E}, \mathsf{M}), isr, (ie, k :: is, cs)) \\ \sigma' = ((\mathsf{G}, \mathsf{E}', \mathsf{M}), isr, (1, is, cs)) \quad \lfloor \kappa_s \rfloor = (c, \kappa_e, \mathsf{E}') \cdot \kappa_s'\end{array}}{\mathsf{P} \vdash ((\mathbf{iext}, (\circ, \kappa_s)), \sigma) \bullet\!\!-\!\!\mathrm{L}\!\!\triangleright ((c, (\kappa_e, \kappa_s')), \sigma')} \ (\textsc{IRet})$$

$$\frac{\sigma = (m, isr, (ie, is, cs)) \quad \sigma' = (m, isr, (0, is, cs))}{\mathsf{P} \vdash ((\mathbf{cli}, \mathsf{K}), \sigma) \bullet\!\!-\!\!\mathrm{L}\!\!\triangleright ((\mathbf{skip}, \mathsf{K}), \sigma')} \ (\textsc{Cli})$$

$$\frac{\sigma = (m, isr, (ie, is, cs)) \quad \sigma' = (m, isr, (1, is, cs))}{\mathsf{P} \vdash ((\mathbf{sti}, \mathsf{K}), \sigma) \bullet\!\!-\!\!\mathrm{L}\!\!\triangleright ((\mathbf{skip}, \mathsf{K}), \sigma')} \ (\textsc{Sti})$$

$$\frac{\eta_i \vdash (\mathsf{C}, m) \longmapsto (\mathsf{C}', m')}{(\mathsf{A}, (\eta_a, \eta_i, \theta)) \vdash (\mathsf{C}, (m, isr, \delta)) \bullet\!\!-\!\!\mathrm{L}\!\!\triangleright (\mathsf{C}', (m', isr, \delta))} \ (\textsc{KCstep})$$

图 4.14　底层内核操作语义

执行内核代码的规则，$\mathsf{InOS}(\mathsf{C}, \mathsf{P})$（定义在图4.11中）用来判断当前是否在执行内核代码，其通过程序后继来判断当前任务处于哪个函数（中断处理程序）中，如果该函数属于 O，则任务正在执行内核代码。当应用程序调用内核 API 函数 f 时，从 ***fexec***$(\mathsf{f}, \_)$ 开始到 f 返回之前都认为执行的是内核代码，可以将 ***fexec***$(\mathsf{f}, \_)$ 看作是应用代码和内核代码的分割点，关于函数调用的操作语义后面会具体介绍。对应的 TCLT 是执行应用代码的规则，注意这里应用程序只能调用自己定义的函数和内核 API，而无法调用内核内部函数和中断处理程序。

图4.14给出了底层机器内核操作语义。当内核代码执行时，其可能是在执行汇编原语也可能是在执行一行 C 语言指令。汇编原语的语义已经介绍过了，图4.14给出了形式化定义，这里不再一一介绍。需要注意的是，由于在进中断时保存了上下文，执行中断返回语句（IRET）时用 $\lfloor \kappa_s \rfloor$ 取出被保存的上下文，其具体定义在图4.11中。

本文不考虑钩子函数（hook）机制，内核代码执行时只能调用内部函数 η_i，而不允许调用应用代码。钩子函数是指系统 API 调用的一些只有声明但并未定义的函数接口，这些函数留给用户来根据实际需求实现。

```
Inductive expr:=
| enull : expr
| evar : var -> expr
| econst32 : int32 -> expr
| eunop : uop -> expr -> expr
| ebinop : bop -> expr -> expr -> expr
| ederef : expr -> expr
| eaddrof : expr -> expr
| efield : expr -> ident -> expr
| ecast : expr -> type -> expr
| earrayelem : expr -> expr -> expr.
```

图 4.15　在 Coq 中归纳定义 C 语言表达式

```
Inductive stmts :=
| sskip : option val -> stmts
| sassign : expr -> expr -> stmts
| sif : expr -> stmts -> stmts -> stmts
| sifthen:expr->stmts->stmts
| swhile : expr -> stmts -> stmts
| sret : stmts
| srete : expr -> stmts
| scall : fid -> exprlist -> stmts
| scalle : expr -> fid -> exprlist -> stmts
| sseq : stmts -> stmts -> stmts
```

图 4.16　在 Coq 中归纳定义 C 语言程序语句

4.1.4　在 Coq 中编码

为了在 Coq 中完成系统内核验证，需要在 Coq 中归纳定义 C 语言子集的抽象语法树，然后在 Coq 中编码内核程序。图4.15给出了在 Coq 中对 C 语言子集表达式的归纳定义。其中冒号左边的是归纳定义构造子，例如：**enull** 用于构造 C 语言中的 NULL 表达式，**evar** 用于构造变量表达式。图4.16给出了在 Coq 中对 C 语言部分程序语句的归纳定义。直接使用上述归纳定义在 Coq 中定义 C 程序会非常复杂难看，在 Coq 中符号机制（notation）的帮助下，本文将 Coq 代码和 C 语言代码的结构定义得非常相似。本文中的 C 程序都是手动在 Coq 中进行编码的，实际上可以实现一个工具来自动将 C 语言程序转化为 Coq 中对应的编码。图4.17(b) 展示了在 Coq 中对 μC/OS-II 中的调度函数（图4.17 (a)）的编码。对如何在 Coq 中定义 C 语言语法感兴趣的读者可以看本文的 Coq 实现 [3] 中的“/CertiOS/framework/machine/language.v ”文件。同样可以在 Coq 中归纳定义 C 语言的小步操作语义，这里不再展开介绍，具体实现在“/CertiOS/framework/machine/opsem.v ”文件中。

```
void  OS_Sched (void)
{

  INT8U      y;

  OS_ENTER_CRITICAL();
  y             = OSUnMapTbl[OSRdyGrp];
  OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
  if (OSPrioHighRdy != OSPrioCur) {
    OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
    OSCtxSwCtr++;
    OSCtxSw();
  }
  OS_EXIT_CRITICAL();
  return;
}
```

(a) 调度程序的 C 语言实现

```
Definition OS_Sched_impl :=
Void ·OS_Sched·(␣ ␣)··{
       ␣y @ Int8u␣;

        ENTER_CRITICAL;ₛ
        y' =ₑ OSUnMapTbl'[OSRdyGrp'];ₛ
        OSPrioHighRdy' =ₑ  ((y' ≪ '3) +ₑ OSUnMapTbl'[OSRdyTbl'[y']]);ₛ
        If (OSPrioHighRdy' !=ₑ OSPrioCur') {
            OSTCBHighRdy' =ₑ OSTCBPrioTbl'[OSPrioHighRdy'];ₛ
            ++ OSCtxSwCtr';ₛ
            SWITCH
        };ₛ
        EXIT_CRITICAL;ₛ
        RETURN
}·.
```

(b) 调度程序在 Coq 中的编码

图 4.17　C 代码和 Coq 编码

4.2　高层机器模型

从应用层程序员的角度来看，操作系统内核可以理解为，支持多任务和系统调用的扩展的 C 语言。系统调用的实现对于应用层程序员来说是个黑盒，他们只知道执行系统调用产生的效果。如前所述，C 语言用来实现应用程序 A，系统调用会去执行 $\mathcal{O}$ 中的抽象规范，而相应的底层内核数据结构会被抽象成应用层程序员理解中的抽象内核状态 Σ。本节将先介绍高层语言，接着介绍抽象内核状态，然后以 μC/OS-II 中 OSTimeDly 函数为例介绍如何定义 API 抽象规

范，并以 μC/OS-II 调度程序的规范 $\chi_{\mu C/OS\text{-}II}$ 为例看如何定义调度规范，最后介绍高层操作语义。

4.2.1　高层规范语言

$$
\begin{array}{lll}
(\mathit{HStmts}) & \mathbf{s} ::= \mathbf{sched} \mid \gamma(\bar{v}) \mid \mathbf{assert\ b} \mid \mathbf{end}\ \hat{v} \mid \mathbf{s}_1; \mathbf{s}_2 \mid \mathbf{s}_1 + \mathbf{s}_2 & \\
(\mathit{HExpr}) & \mathbf{b} \in \mathit{HAbsSt} \to \mathit{Bool} & (\mathit{HAPICd})\ \omega \in \mathit{ValList} \to \mathit{HStmts} \\
(\mathit{HAPISet}) & \varphi ::= \{f_1 \rightsquigarrow \omega_1, \ldots, f_n \rightsquigarrow \omega_n\} & (\mathit{HEvtSet})\ \varepsilon ::= [\mathbf{s}_0, \ldots, \mathbf{s}_{N-1}] \\
(\mathit{HSched}) & \chi \in \mathit{HAbsSt} \to \mathit{TaskId} \to \mathit{Prop} & (\mathit{TaskId})\ t \in \mathit{Nat} \\
(\mathit{HOSCode}) & \mathbb{O} ::= (\varphi, \varepsilon, \chi) & (\mathit{HProg})\ \mathbb{P} ::= (A, \mathbb{O})
\end{array}
$$

图 4.18　高层规范语言

如图4.18所示，完整的高层程序 $\mathbb{P}$ 由应用程序 A 和内核抽象规范 $\mathbb{O}$ 组成。这里应用程序和底层机器模型中的一样（图4.1）。系统规范代码 $\mathbb{O}$ 包含系统 API 的抽象规范 φ，中断处理程序的抽象规范 ε 以及抽象的调度程序 χ。高层机器上的程序员无法控制中断，中断始终是允许的。中断被抽象为一个可以在任何时刻发生的外部事件。当 k 号事件发生时，$\varepsilon(k)$ 会被执行，$\varepsilon(k)$ 可以理解为 k 号中断的规范。

系统 API 被抽象为从值表到抽象语句 **s** 的映射，**s** 相对于一般程序语言的语句来说比较简单，但是有很强的表达力。**sched** 用来表示执行调度程序，其语义取决于 χ。如图4.18所示，χ 是抽象内核状态 Σ（定义在图4.19 中）和任务 ID 的二元关系，也就是说给定一个抽象内核状态，和该抽象内核状态满足 χ 关系的任务都可以作为将要切换到的任务。注意这里 χ 是一个二元关系而不是一个根据程序状态返回任务 ID 的函数，因此抽象的调度程序可以是不确定的。因为 χ 是内核规范的一部分，所以在验证框架中 **sched** 的语义也是可配置的。和已有的很多操作系统内核验证工作不同，本文没有简单的将抽象调度程序定义为选取任意一个就绪任务，因此本文验证框架支持在抽象机器模型上验证调度相关的性质（例如 PIF）。

$\gamma(\bar{v})$ 用来定义任何高层机器状态的原子转换，$\bar{v}$ 是用来记录系统 API 的参数的值表。$\gamma(\bar{v})$ 是抽象内核状态之间的二元关系，如果两个抽象内核状态 Σ_1 和 Σ_2 满足 $\gamma(\bar{v})$，表示如果参数的值为 $\bar{v}$，当前抽象内核状态为 Σ_1 时, 执行 γ 后状态可能是 Σ_2。**end** $\hat{v}$ 表示系统 API 或者中断处理程序执行结束，返回值为 $\hat{v}$。$\mathbf{s}_1; \mathbf{s}_2$ 表示顺序组合，$\mathbf{s}_1 + \mathbf{s}_2$ 表示不确定选择，也即下一步可能执行 $\mathbf{s}_1$ 也可能执行 $\mathbf{s}_2$。**assert b** 表示当前状态如果满足 **b** 则继续执行，否则程序无法执行（stuck）。

中断处理程序和系统 API 的规范类似，但是由于中断处理程序没有参数，所以只是抽象为一个抽象语句。

$$
\begin{array}{llll}
(\mathit{HWorld}) & \mathbb{W} ::= (\mathsf{T}, \Delta, \Sigma) & (\mathit{HAbsSt}) & \Sigma ::= \{\mathbf{a}_1 \rightsquigarrow \Omega_1, \ldots, \mathbf{a}_n \rightsquigarrow \Omega_n\} \\
(\mathit{HDataNm}) & \mathbf{a} ::= \texttt{tcbls} \mid \texttt{ecbls} \mid \texttt{ctid} \mid \ldots & & \\
(\mathit{HData}) & \Omega ::= \alpha \mid \beta \mid \mathrm{t} \mid \ldots & (\mathit{HEvtId}) & \mathrm{eid} \in \mathit{Addr} \\
(\mathit{HTCBLs}) & \alpha ::= \{\mathrm{t}_1 \rightsquigarrow (\mathrm{pr}_1, ts_1), \ldots, \mathrm{t}_n \rightsquigarrow (\mathrm{pr}_n, ts_n)\} & & \\
(\mathit{HECBLs}) & \beta ::= \{\mathrm{eid}_1 \rightsquigarrow \mathrm{ed}_1, \ldots, \mathrm{eid}_n \rightsquigarrow \mathrm{ed}_n\} & & \\
(\mathit{Priority}) & \mathrm{pr} \in \mathit{int32} & (\mathit{WaitType}) & wt ::= \mathsf{mtx}(\mathrm{eid}) \mid \mathsf{tm} \mid \ldots \\
(\mathit{WaitQ}) & \mathrm{Q} \in \mathtt{nil} \mid \mathrm{t} :: \mathrm{Q} & (\mathit{HStatus}) & ts ::= \mathsf{rdy} \mid \mathsf{wait}(wt, \mathrm{k}) \\
(\mathit{MtxOwner}) & w ::= \bot \mid (\mathrm{t}, \mathrm{pr}) & (\mathit{HECBDt}) & \mathrm{ed} ::= \mathsf{mutex}(\mathrm{Q}, \mathrm{pr}, w) \mid \ldots
\end{array}
$$

图 4.19　高层程序状态

4.2.2　高层程序状态

高层全局程序状态 $\mathbb{W}$ 包含三个部分，分别是任务池 T、用户程序状态 Δ 和抽象内核状态 Σ。不同内核的抽象内核状态是不一样的。这里先假定抽象内核状态有三个部分，分别是任务列表 α、等待事件列表 β 和当前任务号。任务列表保存了每个任务的优先级和该任务的当前状态，任务状态 *ts* 分为两种，一种是就绪状态 rdy，表示该任务没有等待任何事件，随时可以被调度执行，当前正在运行的任务也是就绪状态；另一种是等待状态 $\mathsf{wait}(wt, \mathrm{k})$，$wt$ 表示等待的事件类型，k 表示要等待的时间。wt 分为两种，一种是单纯的等待一个时间（tm），只要等待时间结束该任务就变为就绪态；另一种状态是等待一个事件，例如 $\mathsf{mtx}(\mathrm{eid})$ 表示等待一个 ID 为 eid 的互斥信号量。等待事件列表 β 是一个从事件 ID 到该事件状态 ed 的映射，不同的事件的类型是不同的，例如 $\mathsf{mutex}(\mathrm{Q}, \mathrm{pr}, w)$ 表示一个互斥信号量的优先级为 pr，等待该互斥信号量的任务集合为 Q，占有该信号量的任务信息为 w，w 为 $\bot$ 表示该互斥信号量没有被某个任务占有，$(\mathrm{t}, \mathrm{pr})$ 表示任务 t 占有了该互斥信号量，且该任务的原始优先级为 pr，这里记录原始优先级信息是因为 μC/OS-II 中互斥信号量为了防止发生优先级反转而引入了优先级上限机制（priority-ceiling），导致任务在占有互斥信号量后优先级可能发生变化，在任务释放互斥信号量时需要恢复原始优先级。

下面以 μC/OS-II 中的 API "`void OSTimeDly(Int16u ticks)`" 的规范为例来看下如何使用规范语言。图4.20给出了 OSTimeDly 的代码和规范 $\mathbf{s}_{\mathtt{dly}}$，这里 OSTimeDly 只有一个参数，所以描述参数的值表被简写为一个值 ticks。OSTimeDly 的代码首先检查参数 ticks，如果 ticks 小于或者等于 0 表示参数错误，函数直接返回；如果 ticks 参数大于 0，程序会先调用 OS_ENTER_CRITICAL 进入临界区，μC/OS-II 中的 OS_ENTER_CRITICAL 是用汇编实现的，这里被抽象成了原语 **encrt**；3~5 行用来将当前任务从就绪任务表中删除，就绪任务表保存了所有的就绪任务号，为了快速的从就绪任务表中取出最高优先级的任务，μC/OS-II 中就绪任务表是用位图来实现的，这里具体的实现细节就不展开描述

void OSTimeDly (Int16u ticks) :

```
1  if (ticks > 0) {
2      OS_ENTER_CRITICAL();
3      OSRdyTbl[OSTCBCur->OSTCBY] =
                  OSRdyTbl[OSTCBCur->OSTCBY]& (OSTCBCur->OSTCBBitX);
4      if (OSRdyTbl[OSTCBCur->OSTCBY]==0) {
5          OSRdyGrp= OSRdyGrp&(OSTCBCur->OSTCBBitY)
       }
6      OSTCBCur->OSTCBDly=ticks;
7      OS_EXIT_CRITICAL();
8      OS_Sched();
   }
9  return;
```

$$\text{OSTimeDly: } \mathbf{s}_{\mathtt{dly}} \stackrel{\text{def}}{=} (\gamma_{\text{err}}(\mathsf{ticks}) + (\gamma_{\text{dly}}(\mathsf{ticks}); \mathbf{sched}))$$

$$\gamma_{\text{err}}(\mathsf{ticks}) \stackrel{\text{def}}{=} \lambda\Sigma, (\hat{v}, \Sigma').\ \Sigma = \Sigma' \wedge \mathtt{ticks} \leqslant 0$$

$$\gamma_{\text{dly}}(\mathsf{ticks}) \stackrel{\text{def}}{=} \lambda\Sigma, (\hat{v}, \Sigma').\ \mathtt{ticks} > 0 \wedge (\exists \mathrm{t}, \mathrm{pr}.\Sigma(\mathtt{ctid}) = \mathrm{t} \wedge \Sigma(\mathtt{tcbls})(\mathrm{t}) = (\mathrm{pr}, \mathsf{rdy}) \wedge \Sigma' = \Sigma\{\mathtt{tcbls} \rightsquigarrow \{\mathrm{t} \rightsquigarrow (\mathrm{pr}, \mathsf{wait}(\mathsf{tm}, \mathsf{ticks}))\}\})$$

图 4.20　OSTimeDly 的实现和规范

了。第 6 行将当前任务的任务控制块（Task-Control-Block）中的等待时间设置为 ticks，然后退出临界区并执行调度程序。

位图实现的就绪任务表和相应的位操作等对于应用层程序员来说是不可见的，应用层程序员不需要理解这些复杂的实现细节，他们只需要知道该函数要求参数 ticks 大于 0，如果参数正确当前任务会等待 ticks 时间并调度去其他任务执行，如果参数错误则系统状态不发生变化。$\mathbf{s}_{\mathtt{dly}}$ 很好的刻画了应用层程序员理解中的 OSTimeTick，$\mathbf{s}_{\mathtt{dly}}$ 根据 ticks 参数分为两种情况，当 tick 小于等于 0 时，抽象内核状态不发生变化。当 ticks 大于 0 时，会更新抽象任务表中当前任务的状态，将其从就绪态设置为等待时间 ticks，并执行一次抽象调度程序。

下面以 μC/OS-II 中的调度程序为例来看下如何定义 χ, μC/OS-II 中的调度策略是优先级调度，调度程序选取所有就绪态任务中优先级最高的任务继续执行。下面的定义是对 μC/OS-II 调度程序的规范，如果抽象任务状态 Σ 和 t 满足调度程序，要求 t 是抽象任务表 α 中的一个任务，且 t 当前为就绪状态，并且除了 t 外，α 中所有的就绪任务的优先级都低于 t 的优先级。可以看出该规范很好的刻画了 μC/OS-II 的调度程序的行为。

$$\chi_{\mu C/OS\text{-}II} \stackrel{\text{def}}{=} \lambda\Sigma, t.\ \exists\alpha, pr.\ \Sigma(\texttt{tcbls}) = \alpha \wedge \alpha(t) = (pr, \mathsf{rdy}) \wedge \forall t', pr'.\ (t \neq t' \wedge \alpha(t') = (pr', \mathsf{rdy})) \rightarrow pr' \prec pr$$

4.2.3　操作语义

和底层操作语义类似，高层操作语义也被分为四层，分别是高层全局操作语义、高层任务操作语义、规范操作语义和 C 语言子集操作语义。其中 C 语言操作语义和底层一样，这里不再赘述。

$$\boxed{\mathbb{P} \vdash \mathbb{W} =\text{H}\Rightarrow \mathbb{W}'}$$

$$\frac{\begin{array}{c}\Sigma(\texttt{ctid}) = t \quad T(t) = C \quad T' = T\{t \rightsquigarrow C'\} \\ \Delta|_t = m \quad \Delta' = \textsc{updCS}(\Delta, t, m') \quad \mathbb{P} \vdash (C, m, \Sigma) -\text{H}\rightarrow (C', m', \Sigma')\end{array}}{\mathbb{P} \vdash (T, \Delta, \Sigma) =\text{H}\Rightarrow (T', \Delta', \Sigma')}\ (\textsc{Htask})$$

$$\frac{\begin{array}{c}\mathbb{P} = (A, (\varphi, \varepsilon, \chi)) \quad \Sigma(\texttt{ctid}) = t \quad \chi\ \Sigma\ t' \\ T(t) = (\textbf{sched}; \textbf{s}, K) \quad T' = T\{t \rightsquigarrow (\textbf{s}, K)\}\end{array}}{\mathbb{P} \vdash (T, \Delta, \Sigma) =\text{H}\Rightarrow (T', \Delta, \Sigma\{\texttt{ctid} \rightsquigarrow t'\})}\ (\textsc{Schd})$$

$$\frac{\Sigma(\texttt{ctid}) = t \quad T(t) = (c, (\kappa_e, \kappa_s)) \quad T' = T\{t \rightsquigarrow (\varepsilon(k), (o, (c, \kappa_e, \emptyset) \cdot \kappa_s))\}}{(A, (\varphi, \varepsilon, \chi)) \vdash (T, \Delta, \Sigma) =\text{H}\Rightarrow (T', \Delta, \Sigma')}\ (\textsc{Pevent})$$

图 4.21　高层全局操作语义

图4.21给出了高层全局操作语义的定义，全局操作语义执行一步 $\mathbb{P} \vdash \mathbb{W} =\text{H}\Rightarrow \mathbb{W}'$ 可能是当前任务执行一步（Htask），也可能是执行抽象调度（Schd），或者被外部事件打断（Pevent）

Htask 规则给出了当前任务执行一步对全局状态的改变，和底层机器不同，高层机器的当前任务号保存在抽象内核状态中，$\Delta|_t$ 表示从全局用户状态 Δ 中获取当前任务的用户状态 m，$\textsc{updCS}(\Delta, t, m')$ 用当前任务执行一步后的用户状态 m' 来更新 Δ 中对应的部分。这些辅助符号的定义在图4.11中。抽象内核状态 Σ 是每个任务共享的。Schd 规则根据调度策略 χ 来执行抽象调度，并将新的任务号 t' 写入抽象内核状态。外部事件随时可能发生，Pevent 是外部事件发生时的规则。

$\boxed{(\mathbf{s},\Sigma)\bullet\!\!-\!\!\mathrm{H}\!\!\triangleright(\mathbf{s}',\Sigma')}$

$$\frac{\gamma\ (\bar{v},\Sigma)\ (\hat{v},\Sigma')\quad \mathrm{dom}(\Sigma(\mathtt{tcbls}))=\mathrm{dom}(\Sigma'(\mathtt{tcbls}))\quad \mathrm{dom}(\Sigma)=\mathrm{dom}(\Sigma')\quad \Sigma(\mathtt{ctid})=\Sigma'(\mathtt{ctid})}{(\gamma(\bar{v}),\Sigma)\bullet\!\!-\!\!\mathrm{H}\!\!\triangleright(\mathbf{end}\ \hat{v},\Sigma')}\ (\textsc{Prim})$$

$$\frac{\mathbf{b}\ \Sigma}{(\mathbf{assert}\ \mathbf{b},\Sigma)\bullet\!\!-\!\!\mathrm{H}\!\!\triangleright(\mathbf{end}\ \bot,\Sigma)}\ (\textsc{Assert})$$

$$\frac{}{(\mathbf{end}\ \hat{v};\mathbf{s},\Sigma)\bullet\!\!-\!\!\mathrm{H}\!\!\triangleright(\mathbf{s},\Sigma)}\ (\textsc{End})\qquad \frac{(\mathbf{s}_1,\Sigma)\bullet\!\!-\!\!\mathrm{H}\!\!\triangleright(\mathbf{s}_1',\Sigma')}{(\mathbf{s}_1;\mathbf{s}_2,\Sigma)\bullet\!\!-\!\!\mathrm{H}\!\!\triangleright(\mathbf{s}_1';\mathbf{s}_2,\Sigma')}\ (\textsc{Seq})$$

$$\frac{}{(\mathbf{s}_1+\mathbf{s}_2,\Sigma)\bullet\!\!-\!\!\mathrm{H}\!\!\triangleright(\mathbf{s}_1,\Sigma)}\ (\textsc{Choice1})\qquad \frac{}{(\mathbf{s}_1+\mathbf{s}_2,\Sigma)\bullet\!\!-\!\!\mathrm{H}\!\!\triangleright(\mathbf{s}_2,\Sigma)}\ (\textsc{Choice2})$$

图 4.23　规范操作语义

$\boxed{\mathbb{P}\vdash(C,m,\Sigma)-\!\!\mathrm{H}\!\!\rightarrow(C',m,\Sigma')}$

$$\frac{(\mathbf{s},\Sigma)\bullet\!\!-\!\!\mathrm{H}\!\!\triangleright(\mathbf{s}',\Sigma')}{\mathbb{P}\vdash((\mathbf{s},K),m,\Sigma)-\!\!\mathrm{H}\!\!\rightarrow((\mathbf{s},K),m,\Sigma')}\ (\textsc{InApi})$$

$$\frac{A\vdash(C,m)\longmapsto(C',m')}{(A,\mathbb{O})\vdash(C,m,\Sigma)-\!\!\mathrm{H}\!\!\rightarrow(C',m',\Sigma)}\ (\textsc{HTClt})$$

$$\frac{\varphi(f)=\omega\quad C=(\boldsymbol{fexec}(f,\bar{v}),K)}{(A,(\varphi,\varepsilon,\chi))\vdash(C,m,\Sigma)-\!\!\mathrm{H}\!\!\rightarrow((\omega\ \bar{v},K),m,\Sigma)}\ (\textsc{EnApi})$$

$$\frac{}{\mathbb{P}\vdash((\mathbf{end}\ v,K),m,\Sigma)-\!\!\mathrm{H}\!\!\rightarrow((v,K),m,\Sigma)}\ (\textsc{EndApi1})$$

$$\frac{}{\mathbb{P}\vdash((\mathbf{end}\ ,K),m,\Sigma)-\!\!\mathrm{H}\!\!\rightarrow((\mathbf{skip},K),m,\Sigma)}\ (\textsc{EndApi2})$$

$$\frac{C=(\mathbf{end}\ ,(\circ,(c,\kappa_e,\emptyset)\cdot\kappa_s))}{\mathbb{P}\vdash(C,m,\Sigma)-\!\!\mathrm{H}\!\!\rightarrow((c,(\kappa_e,\kappa_s)),m,\Sigma)}\ (\textsc{EndEvt})$$

图 4.22　高层任务操作语义

图4.22给出了高层规范操作语义。InApi是当前任务执行规范代码的规则，规范代码只会访问抽象内核状态 Σ。HTClt是高层任务执行应用代码的规则，和底层一样，应用代码只会访问用户程序状态 m。EnApi规则用于应用程序调用系统 API 进入内核规范代码执行。EndApi1和 EndApi2规则分别用于带返回值和不带返回值的系统 API 结束退出。EnvEvt 用于外部事件结束退出。

之前已经介绍过了规范语言，图4.23给出了规范语言的操作语义。PRIM 规则用于高层规范中的原子操作，这里需要注意 $\Sigma(\texttt{ctid}) = \Sigma'(\texttt{ctid})$ 表示除了 SCHD规则之外，普通的原子操作不能改变当前任务号，$\mathrm{dom}(\Sigma) = \mathrm{dom}(\Sigma')$ 表示内核模块不会被动态加载和卸载。这里不考虑任务的动态创建和删除（$\mathrm{dom}(\Sigma(\texttt{tcbls})) = \mathrm{dom}(\Sigma'(\texttt{tcbls}))$），为了模块化证明需要时刻保证底层和高层任务是一一对应的，如果要支持任务的动态创建和删除需要保证底层任务创建的同时高层任务也被创建，最新版本的验证框架中已经支持动态任务创建和删除，这部分不在本文的介绍范围内。其他的规则比较直观，这里不再一一介绍。

4.3　操作系统正确性定义

如前所述，操作系统正确性可以通过上下文精化关系来定义。下面给出操作系统正确性的形式化定义。

定义 4.3.1 (操作系统正确性)**.** $O \sqsubseteq_\psi \mathbb{O}$ 成立，当且仅当，对于任意的 A、W、$\mathbb{W}$，如果 $\mathsf{Match}(\psi, W, \mathbb{W})$ 成立，那么 $((A, O), W) \preccurlyeq ((A, \mathbb{O}), \mathbb{W})$ 成立。

这里 W 和 $\mathbb{W}$ 分别是底层和高层的全局机器状态，ψ 是对初始系统状态的要求。由定义4.3.1可以看出，在一个给定的全局初始状态要求 ψ 后，一个操作系统的实现 O 相对于其规范 $\mathbb{O}$，在一致性关系要求 ψ 下是正确的（$O \sqsubseteq_\psi \mathbb{O}$），当且仅当，对于任意的应用程序 A，底层初始程序状态 W 和高层初始程序状态 $\mathbb{W}$，如果满足 $\mathsf{Match}(\psi, W, \mathbb{W})$，则底层机器 $((A, O), W)$ 和高层机器 $((A, \mathbb{O}), \mathbb{W})$ 满足可见事件精化关系。这里任意的 A 就可以理解为上下文精化关系中的“上下文”。Match 是对初始状态的一些基本要求，定义如下：

$$
\begin{aligned}
\mathsf{Match}(\psi, (T, \Delta, \Lambda, t), (\mathbb{T}, \Delta, \Sigma)) \stackrel{\text{def}}{=}\ & (t \in \mathrm{dom}(T)) \wedge (\psi\ \Lambda\ \Sigma) \wedge (t = \Sigma(\texttt{ctid})) \wedge \\
& (\mathrm{dom}(T) = \mathrm{dom}(\Sigma(\texttt{tcbls}))) \wedge \\
& (\forall\, t.t \in \mathrm{dom}(T) \Longrightarrow \exists s.\mathbb{T}(t) = (s, (\circ, \bullet)))
\end{aligned}
$$

Match 包含如下 6 个要求：（1）初始状态时，底层和高层的任务池和用户状态是相同的；（2）底层当前任务 t 属于任务池；（3）底层内核状态 Λ 和抽象内核状态 Σ 满足一致性关系 ψ；（4）底层和高层的当前任务号是相同的；（5）任务池中包含的任务集合和抽象任务表中是相同的；（6）初始状态下任务池中每个任务的表达式后继和语句后继都为空。

ψ 用来描述系统内核初始状态。不同的操作系统，其初始化后的状态是不一样的，所以这里只给出 ψ 的类型，但是没有具体定义。其类型如下：

$$\psi \in \mathit{LOSFullSt} \to \mathit{HAbsSt} \to \mathit{Prop}$$

参考 [7] 的工作，图4.24给出了可见事件精化关系的形式化定义。它比较了低层程序与高层程序产生的事件路径（event trace）。如图所示，事件路径 ξ 是

$$(\mathit{EvtTrace})\quad \xi \ ::= \ \mathtt{nil} \mid \lightning \mid \vartheta :: \xi$$

$$\frac{}{\mathit{LETr}(\mathrm{P}, W, \mathtt{nil})} \qquad \frac{\mathrm{P} \vdash W =\!\!\Rightarrow_{\mathrm{L}} \textbf{abort}}{\mathit{LETr}(\mathrm{P}, W, \lightning)} \qquad \frac{\mathrm{P} \vdash W =\!\!\Rightarrow_{\mathrm{L}} W' \quad \mathit{LETr}(\mathrm{P}, W', \xi)}{\mathit{LETr}(\mathrm{P}, W, \xi)}$$

$$\frac{\mathrm{P} \vdash W =\!\!\Rightarrow_{\mathrm{L}, \vartheta} W' \quad \mathit{LETr}(\mathrm{P}, W', \xi)}{\mathit{LETr}(\mathrm{P}, W, \vartheta :: \xi)}$$

$$\frac{}{\mathit{HETr}(\mathbb{W}, \mathtt{nil})} \qquad \frac{\mathbb{P} \vdash \mathbb{W} =\!\!\Rightarrow_{\mathrm{H}} \textbf{abort}}{\mathit{HETr}(\mathbb{P}, \mathbb{W}, \lightning)} \qquad \frac{\mathbb{P} \vdash \mathbb{W} =\!\!\Rightarrow_{\mathrm{H}} \mathbb{W}' \quad \mathit{HETr}(\mathbb{P}, \mathbb{W}', \xi)}{\mathit{HETr}((\mathbb{P}, \mathbb{W}), \xi)}$$

$$\frac{\mathbb{P} \vdash \mathbb{W} =\!\!\Rightarrow_{\mathrm{H}, \vartheta} \mathbb{W}' \quad \mathit{HETr}(\mathbb{P}, \mathbb{W}', \xi)}{\mathit{HETr}(\mathbb{P}, \mathbb{W}, \vartheta :: \xi)}$$

$$(\mathrm{P}, W) \preccurlyeq (\mathbb{P}, \mathbb{W}) \stackrel{\text{def}}{=} \forall \xi. \mathit{LETr}(\mathrm{P}, W, \xi) \Longrightarrow \mathit{LETr}(\mathrm{P}, W, \xi)$$

图 4.24　可见事件精化关系

由事件构成的序列，本文中用 nil 表示空序列，出错记号 $\lightning$ 表示程序出错。在应用语言中，本文引入了 **print** e 语句可以产生一个外部可见的输出事件 v，v 是表达式 e 在当前程序状态下的值。实际上操作语义是 "$\_ \vdash \_ \longrightarrow_{\hat{\vartheta}} \_$" 这种格式的，在介绍操作语义时为了方便省去了 $\hat{\vartheta}$，$\hat{\vartheta}$ 定义如下：

$$\hat{\vartheta} \quad \stackrel{\text{def}}{=} \quad \bot \mid \vartheta$$

表示系统中某些指令会产生一个外部可见事件。$\mathit{LETr}(\mathrm{P}, W, \xi)$ 表示底层程序 P 在初始状态为 W 时，执行 0 步或多步可能产生一个事件序列 ξ，相应的 $\mathit{HETr}(\mathbb{P}, \mathbb{W}, \xi)$ 表示高层程序 $\mathbb{P}$ 在初始状态为 $\mathbb{W}$ 时，执行 0 步或多步可能产生一个事件序列 ξ。底层机器 (P, W) 和高层机器 $(\mathbb{P}, \mathbb{W})$ 满足可见事件精化关系 $(\mathrm{P}, W) \preccurlyeq (\mathbb{P}, \mathbb{W})$，是指底层机器能够产生的所有事件序列，都可以被高层机器产生。

4.4　本章小结

本章先介绍了底层机器模型和高层机器模型。底层机器模型被物理地划分为不相交的两个部分，一个是程序处于用户态时访问的应用程序状态 Δ，和程序处于内核态时访问的 Λ。在用户态时执行的应用程序语言是一个 C 语言子集，在内核态时执行的内核语言是 C 语言子集加上扩展的汇编原语，所以内核状态 Λ 是在 Δ 的基础上扩展出了汇编原语访问的中断相关的程序状态 π 和 *isr*。高层机器状态同样也被分为两个部分，分别是用户程序状态 Δ 和规范语言访问的抽象内核状态 Σ。在高层中断被抽象为随机发生的外部事件，当外部事件发生时，机器会去执行外部事件的抽象规范，系统 API 也被抽象为抽象规范。为了结构清晰，一共分为四层来定义底层机器的操作语义，包括 C 操作语

义、内核操作语义、底层任务操作语义以及底层全局操作语义。同样的，高层操作语义也分为四层，分别为 C 操作语义、规范语言操作语义、高层任务操作语义以及高层全局操作语义，其中 C 语言操作语义是共用的，完整的定义在附录 A中。最后给出了精化关系的形式化定义和基于精化关系的操作系统正确性定义。下一章将介绍用于验证操作系统正确性的程序逻辑 CSL-R，并证明其可靠性（soundness）。

第五章　精化程序逻辑

本章介绍用于验证操作系统正确性的精化程序逻辑 CSL-R，本章先介绍关系型断言语言，其次介绍如何用占有权转移思想验证中断，然后介绍推理规则，最后给出 CSL-R 的可靠性证明。

5.1　关系型断言语言

图5.1给出了关系型断言语言的语法, 图5.2给出了断言语义。

$$
\begin{aligned}
(\mathit{Asrt}) \quad & p, q ::= \mathtt{emp} \mid \mathsf{empE} \mid \langle P \rangle \mid a \overset{\tau}{\mapsto} v \mid x \overset{\tau}{\mapsto} v \mid x \overset{\tau}{\mapsto}_{l} v \mid e =_{\tau} v \mid x@(a,\tau) \\
& \mid x@_{l}(a,\tau) \mid \mathsf{ISR}(\mathit{isr}) \mid \mathsf{IE}(\mathit{ie}) \mid \mathsf{IS}(\mathit{is}) \mid \mathsf{CS}(\mathit{cs}) \mid \llcorner k \lrcorner \mid \chi \triangleright t \\
& \mid \mathbf{a} \rightarrowtail \Omega \mid [|\mathbf{s}|] \mid p * p \mid p \wedge p \mid p \vee p \mid \mathsf{true} \mid \mathsf{false} \mid \exists x.\, p \mid \neg p \\
(\mathit{InvAsrt}) \quad & I ::= [p_0, \ldots, p_N] \qquad (\mathit{RetAsrt}) \quad \rho \in \mathit{Option\ Value} \rightarrow \mathit{Asrt}
\end{aligned}
$$

图 5.1　关系型断言

断言用来描述关系型程序状态 Θ，Θ 包含三个部分，分别是底层任务内核状态 σ，高层抽象内核状态 Σ 以及底层代码精化的抽象规范语句 $\mathbf{s}$。σ 定义在图4.2中，其包含当前任务的程序变量和内存 m、*isr* 寄存器已经能够当前任务的中断相关状态 δ。m 包含全局符号表 G、局部符号表 E 和内存 M。

断言 emp 表示底层内存 M 和高层抽象内核状态 Σ 都为空。断言 empE 除了要求当前状态满足 emp 外还要求局部符号表 E 为空。$\langle P \rangle$ 要求当前状态满足 emp 并且命题 P 成立。$(\sigma, \Sigma, \mathbf{s}) \models a \overset{\tau}{\mapsto} v$ 描述了基地址为 a 的一串内存单元，其中保存了值 v，并且高层抽象内核状态为空。$x \overset{\tau}{\mapsto} v$ 表示全局变量 x 的类型是 τ，其在内存中的值为 v。$x \overset{\tau}{\mapsto}_{l} v$ 和 $x \overset{\tau}{\mapsto} v$ 类似，只不过这里 x 是局部变量。$e =_{\tau} v$ 表示表达式 e 在当前状态下的值是 v，类型为 τ，并且 v 的值不是未定义的（$v \neq \mathtt{Vundef}$），计算表达式类型和值的函数 $[\![]\!]^{t}$ 和 $[\![]\!]^{r}$ 的定义在附录A 中。$x@(a,\tau)$ 表示全局变量 x 所在的地址为 a，类型为 τ。$x@_{l}(a,\tau)$ 表示局部变量 x 所在的地址为 a，类型为 τ

ISR(*isr*)、IS(*is*)、IE(*ie*) 和 CS(*cs*) 用来描述对应的中断状态。$\llcorner k \lrcorner$ 表示当前正在运行中断处理程序的中断号为 k，如果 $k = N$ 则表示当前不在执行中断处理程序。$\chi \triangleright t$ 表示根据中断调度策略 χ，t 是可能被调度的任务（因为调度程序不是确定的，所以可能存在多个任务满足调度策略）。$\mathbf{a} \rightarrowtail \Omega$ 用来描述抽象内核状态中的一个抽象单元，该抽象单元的名字为 $\mathbf{a}$，其内容为 Ω。$[|\mathbf{s}|]$ 表示抽象规范语句还未执行的剩余代码为 $\mathbf{s}$。任何 Θ 都满足 true，任何 Θ 都不满足 false，存在断言 $\exists x.\, p$ 使断言语言支持存在量词。因为描述程序状态时一般不会使用全称量词，所以断言语言中没有加入全程量词断言。

$$(\mathit{RelState})\ \Theta ::= (\sigma, \Sigma, \mathbf{s})$$

$$
\begin{array}{ll}
(\sigma,\Sigma,\mathbf{s}) \models \mathtt{emp} & \text{iff } \sigma.\mathfrak{m}.\mathrm{M} = \emptyset \wedge \Sigma = \emptyset \\
(\sigma,\Sigma,\mathbf{s}) \models \mathsf{empE} & \text{iff } \sigma.\mathfrak{m}.\mathrm{E} = \emptyset \wedge (\sigma,\Sigma,\mathbf{s}) \models \mathtt{emp} \\
(\sigma,\Sigma,\mathbf{s}) \models \langle \mathrm{P} \rangle & \text{iff } ((\sigma,\Sigma,\mathbf{s}) \models \mathtt{emp}) \wedge \mathrm{P} \\
(\sigma,\Sigma,\mathbf{s}) \models \mathfrak{a} \overset{\tau}{\mapsto} \nu & \text{iff } \sigma.\mathfrak{m}.\mathrm{M} = \{\mathfrak{a} \overset{\tau}{\rightsquigarrow} \nu\} \wedge \Sigma = \emptyset \\
(\sigma,\Sigma,\mathbf{s}) \models \mathrm{x} \overset{\tau}{\mapsto} \nu & \text{iff } \exists \mathrm{b}.(\sigma.\mathfrak{m}.\mathrm{G})(\mathrm{x}) = (\mathrm{b},\tau) \wedge (\sigma,\Sigma,\mathbf{s}) \models (\mathrm{b},0) \overset{\tau}{\mapsto} \nu \\
(\sigma,\Sigma,\mathbf{s}) \models \mathrm{x} \overset{\tau}{\mapsto}_{\mathfrak{l}} \nu & \text{iff } \exists \mathrm{b}.(\sigma.\mathfrak{m}.\mathrm{E})(\mathrm{x}) = (\mathrm{b},\tau) \wedge (\sigma,\Sigma,\mathbf{s}) \models (\mathrm{b},0) \overset{\tau}{\mapsto} \nu \\
(\sigma,\Sigma,\mathbf{s}) \models e =_{\tau} \nu & \text{iff } [\![e]\!]^{\mathrm{r}}_{\sigma.\mathfrak{m}.\mathrm{M}} = \nu \wedge [\![e]\!]^{\mathrm{t}}_{\sigma.\mathfrak{m}.\mathrm{M}} = \tau \wedge \nu \neq \mathtt{Vundef} \\
(\sigma,\Sigma,\mathbf{s}) \models \mathrm{x}@(\mathfrak{a},\tau) & \text{iff } \exists\ \mathrm{b}.(\sigma.\mathfrak{m}.\mathrm{G})(\mathrm{x}) = (\mathrm{b},\tau) \wedge \mathfrak{a} = (\mathrm{b},0) \\
(\sigma,\Sigma,\mathbf{s}) \models \mathrm{x}@_{\mathfrak{l}}(\mathfrak{a},\tau) & \text{iff } \exists\ \mathrm{b}.(\sigma.\mathfrak{m}.\mathrm{E})(\mathrm{x}) = (\mathrm{b},\tau) \wedge \mathfrak{a} = (\mathrm{b},0) \\
(\sigma,\Sigma,\mathbf{s}) \models \mathsf{ISR}(\mathit{isr}') & \text{iff } \sigma.\mathit{isr} = \mathit{isr}' \wedge (\sigma,\Sigma,\mathbf{s}) \models \mathtt{emp} \\
(\sigma,\Sigma,\mathbf{s}) \models \mathsf{IE}(\mathit{ie}') & \text{iff } \sigma.\delta.\mathit{ie} = \mathit{ie}' \wedge (\sigma,\Sigma,\mathbf{s}) \models \mathtt{emp} \\
(\sigma,\Sigma,\mathbf{s}) \models \mathsf{IS}(\mathit{is}') & \text{iff } \sigma.\delta\mathit{is} = \mathit{is}' \wedge (\sigma,\Sigma,\mathbf{s}) \models \mathtt{emp} \\
(\sigma,\Sigma,\mathbf{s}) \models \mathsf{CS}(\mathit{cs}') & \text{iff } \sigma.\delta.\mathit{cs} = \mathit{cs}' \wedge (\sigma,\Sigma,\mathbf{s}) \models \mathtt{emp} \\
(\sigma,\Sigma,\mathbf{s}) \models \llcorner \mathrm{k} \lrcorner & \text{iff } ((\mathrm{k}{=}\mathrm{N} \wedge \mathit{is}{=}\mathtt{nil}) \vee \exists \mathit{is}'.(\sigma.\delta.\mathit{is}{=}\mathrm{k}{::}\mathit{is}')) \wedge \\
 & \quad (\sigma,\Sigma,\mathbf{s}) \models \mathtt{emp} \\
(\sigma,\Sigma,\mathbf{s}) \models \chi \triangleright \mathrm{t} & \text{iff } \chi\ \Sigma\ \mathrm{t} \\
(\sigma,\Sigma,\mathbf{s}) \models [|\mathbf{s}'|] & \text{iff } \mathbf{s} = \mathbf{s}' \wedge (\sigma,\Sigma,\mathbf{s}) \models \mathtt{emp} \\
(\sigma,\Sigma,\mathbf{s}) \models \mathbf{a} \rightarrowtail \Omega & \text{iff } \Sigma = \{\mathbf{a} \rightsquigarrow \Omega\} \wedge \sigma.\mathfrak{m}.\mathrm{M} = \emptyset \\
(\sigma,\Sigma,\mathbf{s}) \models \mathsf{true} & \text{iff } \mathrm{True} \\
(\sigma,\Sigma,\mathbf{s}) \models \mathsf{false} & \text{iff } \mathrm{False} \\
(\sigma,\Sigma,\mathbf{s}) \models \exists\, \mathrm{x}.\, \mathrm{p} & \text{iff } \exists\, \mathrm{x}.(\sigma,\Sigma,\mathbf{s}) \models \mathrm{p} \\
(\sigma,\Sigma,\mathbf{s}) \models \neg \mathrm{p} & \text{iff } \neg(\sigma,\Sigma,\mathbf{s}) \models \mathrm{p} \\
\mathrm{M} = \{(\mathrm{b},\mathrm{o}) \overset{\tau}{\rightsquigarrow} \nu\} & \overset{\text{def}}{=} \exists \nu_1, \cdots, \nu_{|\tau|}.\, \mathsf{encode}(\nu,\tau) = \{\nu_1, \cdots, \nu_{|\tau|}\} \wedge \\
 & \quad \mathrm{M} = \{(\mathrm{b},\mathrm{o}) \rightsquigarrow \nu_1, \cdots, (\mathrm{b},\mathrm{o}+|\tau|-1) \rightsquigarrow \nu_{|\tau|}\}
\end{array}
$$

$$
\mathrm{f} \perp \mathrm{g} \overset{\text{def}}{=} \mathrm{dom}(\mathrm{f}) \cap \mathrm{dom}(\mathrm{g}) = \emptyset \qquad \Sigma_1 \uplus \Sigma_2 \overset{\text{def}}{=} \begin{cases} \Sigma_1 \cup \Sigma_2 & \text{if } \Sigma_1 \perp \Sigma_2 \\ \mathit{undef} & \text{otherwise} \end{cases}
$$

$$
\sigma_1 \uplus \sigma_2 \overset{\text{def}}{=} \begin{cases} ((\mathrm{G},\mathrm{E},\mathrm{M}_1 \cup \mathrm{M}_2), \mathit{isr}, \delta) & \text{if } \mathrm{M}_1 \perp \mathrm{M}_2 \wedge \sigma_1 = ((\mathrm{G},\mathrm{E},\mathrm{M}_1), \mathit{isr}, \delta) \\ & \qquad \wedge \sigma_2 = ((\mathrm{G},\mathrm{E},\mathrm{M}_2), \mathit{isr}, \delta) \\ \mathit{undef} & \text{otherwise} \end{cases}
$$

$$
\Theta_1 \uplus \Theta_2 \overset{\text{def}}{=} (\sigma_1 \uplus \sigma_2, \Sigma_1 \uplus \Sigma_2, \mathbf{s}) \qquad \text{where } \Theta_1 = (\sigma_1,\Sigma_1,\mathbf{s}) \wedge \Theta_2 = (\sigma_2,\Sigma_2,\mathbf{s})
$$

$$
\Theta \models \mathrm{p}_1 * \mathrm{p}_2 \qquad \text{iff } \exists \Theta_1, \Theta_2.\Theta = \Theta_1 \uplus \Theta_2 \wedge \Theta_1 \models \mathrm{p}_1 \wedge \Theta_2 \models \mathrm{p}_2
$$

$$
\mathrm{p} \Rightarrow \mathrm{q} \qquad \text{iff } \forall \Theta.\Theta \models \mathrm{p} \Rightarrow \Theta \models \mathrm{q}
$$

图 5.2　关系型断言的语义

```
Inductive asrt : Type :=
    Aemp
  | Aconj  (p1: asrt)  (p2: asrt)
  ...
end.
```

```
Fixpoint sat (s:RelState) (p:asrt) :=
  match p with
    | Aemp ⇒ empst s
    | Aconj p q ⇒ sat s p ∧ sat s q
    ...
  end.
```

图 5.3　Coq 中断言的定义

Θ 满足分离连接断言 $p_1 * p_2$ 表示 Θ 可以分为两个不相交的关系型状态分别满足断言 p_1 和 p_2。这里只将底层机器模型中的内存和高层机器模型中的抽象内核状态看作资源，$\Theta = \Theta_1 \uplus \Theta_2$ 表示 Θ 中的内存 M 和抽象内核状态 Σ 可以划分为不相交的两块分别为 Θ_1 和 Θ_2 所有，Θ，Θ_1 和 Θ_2 中其余的程序状态都是相同的。

断言蕴含关系 $p \Rightarrow q$，表示对于任意的程序状态，如果满足断言 p，那么该程序状态同样满足断言 q。

I 是程序中用来描述共享资源满足的全局不变式断言的集合。ρ 用于描述函数返回前需要满足的条件，*Option Value*用于描述返回值，后面会具体介绍。

断言在 Coq 中的定义　图5.3是对上述断言语言在 Coq 中的定义，图中左半部分归纳定义的 asrt 是断言语言定义在 Coq 中的抽象语法树，右半部分的递归函数 sat 对应断言语言的语义（图5.2中的 _ ⊨ _），图中只给出了断言语言抽象语法树和断言语言语义定义的一部分。

在 Coq 中定义断言通常有两种风格，分别是深嵌入（deep embedding）和浅嵌入（shallow embedding）。图5.3中的定义方式为 deep embedding，deep embedding 要求在 Coq 中实现断言语言的抽象语法树，然后再定义断言语言的语义。shallow embedding 不需要定义断言语言的抽象语法树，如果采用 shallow embedding 的定义方法，Coq 中所有满足类型 *RelState* → Prop 的定义都是断言（Prop 是 Coq 中的命题的类型），例如：图5.3中的 Aemp 就可以定义为 $\lambda\Theta$.empst Θ。

deep embedding 和 shallow embedding 各有优劣。deep embedding 设计的好处在于可以根据断言的语法进行分析，并设计相应的自动证明策略；缺点在于不方便扩展，因为每次引入新的断言就会修改断言语言的语法结构，并导致相关的证明需要调整。shallow embedding 的好处在于可以方便的扩展，每次引入新的断言实际上只是多了一个新的定义而已，但是由于没有固定的断言语法，所以不方便设计自动证明策略。本文为了方便设计自动证明策略，采用的是 deep embedding 的设计风格。

$\bar{\mathtt{V}} \stackrel{\text{def}}{=} \mathtt{nil} \mid \bar{v} :: \bar{\mathtt{V}} \qquad\qquad \mathtt{Pf} \in \mathit{ValueList} \to \mathit{Option\ Val}$

$$\mathsf{Astruct}'((b,o),\mathcal{D},\bar{v}) \stackrel{\text{def}}{=}$$
$$\begin{cases} (b,o) \overset{\tau}{\mapsto} v * \mathsf{Astruct}'((b,o+|\tau|),\mathcal{D}',\bar{v}') & \text{if } \bar{v} = v :: \bar{v}' \wedge \mathcal{D} = (\mathrm{id},\tau) :: \mathcal{D}' \wedge \tau \neq \mathtt{Tstruct\_} \wedge \tau \neq \mathtt{Tarray\_} \\ \mathsf{Astruct}'((b,o),\mathcal{D}_1,\bar{v}_1) * \mathsf{Astruct}'((b,o+|\tau|),\mathcal{D}',\bar{v}') & \text{if } \bar{v} = \bar{v}_1 + +\bar{v}' \wedge \mathcal{D} = (\mathrm{id},\tau) :: \mathcal{D}' \wedge \tau = \mathtt{Tstruct}(\_,\mathcal{D}_1) \wedge |\bar{v}_1| = |\mathcal{D}_1| \\ \mathsf{Aarray}'((b,o),\tau',\bar{v}_1) * \mathsf{Astruct}'((b,o+|\tau|),\mathcal{D}',\bar{v}') & \text{if } \bar{v} = \bar{v}_1 + +\bar{v}' \wedge \mathcal{D} = (\mathrm{id},\tau) :: \mathcal{D}' \wedge \tau = \mathtt{Tarray}(\tau',n) \wedge |\bar{v}_1| = n \\ \mathtt{emp} & \text{if } \bar{v} = \mathtt{nil} \wedge \mathcal{D} = \mathtt{nil} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathsf{Aarray}'((b,o),\tau,n,\bar{v}) \stackrel{\text{def}}{=}$$
$$\begin{cases} (b,o) \overset{\tau}{\mapsto} v * \mathsf{Aarray}'((b,o+|\tau|),\tau,n-1,\bar{v}') & \text{if } \bar{v} = v :: \bar{v}' \wedge n > 0 \wedge \tau \neq \mathtt{tstruct\_} \wedge \tau \neq \mathtt{tarray\_} \\ \mathsf{Astruct}'((b,o),\mathcal{D}_1,\bar{v}_1) * \mathsf{Aarray}'((b,o+|\tau|),\tau,n-1,\bar{v}') & \text{if } \bar{v} = \bar{v}_1 + +\bar{v}' \wedge \tau = \mathtt{Tstruct}(\_,\mathcal{D}_1) \wedge |\bar{v}_1| = |\mathcal{D}_1| \\ \mathsf{Aarray}'((b,o),n_1,\tau',\bar{v}_1) * \mathsf{Aarray}'((b,o+|\tau|),\tau,n-1,\bar{v}') & \text{if } \bar{v} = \bar{v}_1 + +\bar{v}' \wedge \tau = \mathtt{Tarray}(\tau',n_1) \wedge |\bar{v}_1| = n \\ \mathtt{emp} & \text{if } \bar{v} = \mathtt{nil} \wedge n = 0 \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathsf{Astruct}((b,o),\tau,\bar{v}) \stackrel{\text{def}}{=} \exists \mathrm{id},\mathcal{D}.\tau = \mathtt{Tstruct}(\mathrm{id},\mathcal{D}) \wedge \mathsf{Astruct}'((b,o),\mathcal{D},\bar{v})$$

$$\mathsf{Aarray}((b,o),\tau,\bar{v}) \stackrel{\text{def}}{=} \exists \tau',n.\tau = \mathtt{Tarray}(\tau',n) \wedge \mathsf{Aarray}'((b,o),\tau',n,\bar{v})$$

$$\mathsf{slseg}(h,tn,\bar{\mathtt{V}},\tau,\mathtt{Pf}) \stackrel{\text{def}}{=}$$
$$\begin{cases} h = tn & \text{if } \bar{\mathtt{V}} = nil \\ \mathtt{Pf}(\bar{v}) = h' \wedge \mathsf{Astruct}(h,\tau,\bar{v}) * \mathsf{slseg}(h',tn,\bar{\mathtt{V}}',\tau,\mathtt{Pf}) & \text{if } \bar{\mathtt{V}} = \bar{v} :: \bar{\mathtt{V}}' \end{cases}$$

$$\mathsf{sl}(h,\bar{\mathtt{V}},\tau,\mathtt{Pf}) \stackrel{\text{def}}{=} \mathsf{slseg}(h,\mathtt{Vnull},\bar{\mathtt{V}},\tau,\mathtt{Pf})$$

$$\mathsf{dlseg}(h,hp,t,tn,\bar{\mathtt{V}},\tau,\mathtt{Pf}_n,\mathtt{Pf}_p) \stackrel{\text{def}}{=}$$
$$\begin{cases} h = tn \wedge t = hp & \text{if } \bar{\mathtt{V}} = nil \\ \mathtt{Pf}_n(\bar{v}) = h' \wedge \mathtt{Pf}_p(\bar{v}) = hp \wedge \mathsf{Astruct}(h,\tau,\bar{v}) * \mathsf{dlseg}(h',h,t,tn,\bar{\mathtt{V}}',\tau,\mathtt{Pf}_n,\mathtt{Pf}_p) & \text{if } \bar{\mathtt{V}} = \bar{v} :: \bar{\mathtt{V}}' \end{cases}$$

$$\mathsf{dl}(h,t,\bar{\mathtt{V}},\tau,\mathtt{Pf}_n,\mathtt{Pf}_p) \stackrel{\text{def}}{=} \mathsf{dlseg}(h,\mathtt{Vnull},t,\mathtt{Vnull},\bar{\mathtt{V}},\tau,\mathtt{Pf}_n,\mathtt{Pf}_p)$$

图 5.4　数据结构断言

数据结构断言的封装　因为操作系统中包含丰富的数据结构，CSL-R 中还包含很多描述数据结构的断言的封装，包括结构体，数组，单向链表和双向链表。所谓数据结构断言的封装，是指定义一些用于构造数据结构断言的函数。这些函数接收数据结构相关的参数，返回描述该数据结构的断言（图5.3 中的 asrt）。

如图5.4 所示，因为结构体中可能有数组类型的成员变量，数组的元素也有可能是结构体类型，所以这里用互递归的方式定义了两个函数 Astruct′ 和 Aarray′，分别是描述结构体和数组的断言的封装。Astruct′ 带三个参数分别是该结构体的起始地址 (b,o)，结构体成员变量声明列表 $\mathcal{D}$ 和一个用来保存结构体中每个成员变量在内存中的值的值表 $\bar{\mathrm{v}}$，返回描述结构体的断言（asrt）。Aarray′ 带四个参数分别是该数组的起始地址 (b,o)，数组元素的类型 τ，数组长度 n 以及保存每个数组元素的值的值表 $\bar{\mathrm{v}}$，返回描述数组的断言。

slseg 用来定义单向链表中的一段，它带五个参数，分别是：（1）该段单向链表头节点的的地址 h；（2）尾节点的后继节点的地址 tn；（3）保存该段单向链表中所有元素对应的内存的值表链 $\bar{\mathrm{v}}$，由于单向链表中每个节点都是一个结构体，所以 $\bar{\mathrm{v}}$ 是一个由值表构成的列表，$\bar{\mathrm{v}}$ 中每个元素都是一个用来保存结构体节点的值表；（4）链表中每个节点的类型 τ；（5）寻找单向链表元素的后继节点地址的函数 `Pf`，因为单向链表中每个节点都有指向下个节点的指针，所以 $\bar{\mathrm{v}}$ 的每个元素 $\bar{\mathrm{v}}$ 中有一个值是单向链表中下一个节点的地址，`Pf` 用来从 $\bar{\mathrm{v}}$ 中找出保存了下个节点指针的值。当一段单向链表为空时要求 $\mathrm{h}=\mathrm{tn}$。sl 表示一段完整的单向链表，其尾节点的后继节点的地址是 `Vnull`。

dlseg 用来定义双向链表中的一段，它带 8 个参数，分别是：（1）该段双向链表的头节点的地址 h；(2) 该段双向链表头节点的前驱节点的地址 hp；（3）该段双向链表尾节点的地址 t；（4）该段双向链表尾节点的后继节点的地址 tn；（5）保存该段双向链表中所有元素对应的内存的值表链 $\bar{\mathrm{v}}$；（6）链表中每个节点的类型 τ；（7）寻找双向链表元素的后继节点地址的函数 $\mathtt{Pf}_\mathrm{n}$；（8）寻找双向链表元素的前驱节点地址的函数 $\mathtt{Pf}_\mathrm{p}$。当一段双向链表为空时要求 $\mathrm{h}=\mathrm{tn}$ 并且 $\mathrm{hp}=\mathrm{t}$。dl 表示一段完整的双向链表，其尾节点的后继节点的地址和头节点的前驱节点的地址都是 `Vnull`。

在定义这些数据结构断言的时候需要尽量保证复用性。例如，操作系统中存在大量不同用途的单向链表，但是只需要给 sl 传入不同的参数 τ 和 `Pf` 就可以构造出描述这些结构体的断言，这样很多结构体相关的性质和证明可以复用（比如：$\forall\tau,\mathtt{Pf},\mathrm{h}.\ \mathsf{sl}(\mathrm{h},\mathtt{nil},\tau,\mathtt{Pf})\Longrightarrow\mathrm{h}=\mathtt{Vnull}$）。

5.2　多级中断的处理

第2.2节介绍了已有的关于如何使用占有权转移思想验证中断处理程序的方法。但是已有工作 [34] 针对的是一个简单的模型，并且只考虑了单级中断。

本文将上述思想扩展到一个实际的 x86 中断模型中，并加入了多级中断的支持。在多级中断模型中，中断能否发生不仅仅依赖于 CPU 是否允许中断（*ie* 的状态），还依赖于有无更高优先级的中断正被服务（*isr* 的状态）。所有可能改变 *ie* 和 *isr* 状态的指令都可能引发占有权的转移。图5.5给出了多级中断下的逻辑内存模型，这里假定中断一共有 6 个, 中断优先级根据中断号依次排列，0 号中断优先级最高，5 号中断优先级最低。图中所包含完整内存块对应图2.2中的 **B**，图2.2中的任务局部资源 **T**在这边没有对应。首先将全局共享资源按照中断优先级来划分，先取出优先级最高的 0 号中断访问的部分 $\mathbf{B_0}$，然后从剩余的部分中依次取出各个中断需要访问的共享资源 $\mathbf{B_1} \sim \mathbf{B_5}$，最后剩余的部分 $\mathbf{B_6}$ 是只在任务间共享的资源。对于 k 号中断，因为优先级高的中断先选择其要访问的资源，所以如果 k 号中断和高优先级的 n 号中断同时都要访问某个资源 R，R 会被划分在 $\mathbf{B_n}$ 中，所以 k 号中断可能访问的共享资源包括 $\mathbf{B_0} \sim \mathbf{B_k}$。任务可能访问的共享资源为 $B_0 \sim B_N$，N 为系统中的中断个数。每个资源块 $\mathbf{B_k}$ 都应一个不变式 I_k 来描述，I 是这些不变式的集合（I 定义在图5.1）。

图5.5给出了不同情况下中断相关指令引发的占有权转移。图中白色的块表示没有被任何中断或任务占有的共享资源，其他的不同形状的块表示被不同中断占有的共享资源。

假设一开始处于状态（1），此时 3 号中断正在被执行（$isr(3) = 1$），中断关闭（$ie = 0$）。如前所述 3 号中断需要访问的共享资源是 $\mathbf{B_0} \sim \mathbf{B_3}$，因为中断关闭所以更高优先级的中断无法打断 3 号中断执行，所以 3 号中断占有了所有需要访问的共享资源 $\mathbf{B_0} \sim \mathbf{B_3}$，图中灰色的方块表示 3 号中断的局部资源。

此时如果执行 **sti** 指令会进入图中状态（2），因为打开中断后更高优先级的中断可能会发生，所以需要保证 $\mathbf{B_0} \sim \mathbf{B_2}$ 是良型的，执行 **sti** 指令后会释放 $\mathbf{B_0} \sim \mathbf{B_2}$ 的占有权。这里因为 *isr* 中的第 3 位没有清空，所以低优先级的中断（包括 3 号中断）不会被响应，所以 $\mathbf{B_3}$ 不需要释放，可以继续占有。

如果执行 **eoi** 指令会进入图中状态（5），由于中断仍然关闭，3 号中断的执行不会被打断，所以不会发生占有权转移。但是在状态（5）下执行 **sti** 会释放 $\mathbf{B_0} \sim \mathbf{B_2}$ 和 $\mathbf{B_3}$，因为此时 $isr(3) = 0$，所以低优先级中断（包括 3 号中断）都可能发生。

进入状态（2）后，执行 **cli** 关闭中断，会获得 $\mathbf{B_0} \sim \mathbf{B_2}$ 的占有权并回到状态（1）。如果执行 **eoi** 会释放 $\mathbf{B_3}$ 来允许低优先级的中断发生并进入状态（4）。

只有在执行完 **eoi** 才允许执行中断返回指令 **iret**。执行 **iret** 时如果中断返回之前中断已经打开（状态（4）），此时占有的资源 $\mathbf{B_0} \sim \mathbf{B_3}$ 已经释放过了，不会再发生占有权转移。如果中断关闭（状态（5））则会释放占有的资源。

图中 “**irq 1**” 表示更高优先级的 1 号中断发生，并打断了 3 号中断的执行，1 号中断会从共享资源中获得 $\mathbf{B_0}$、$\mathbf{B_1}$。

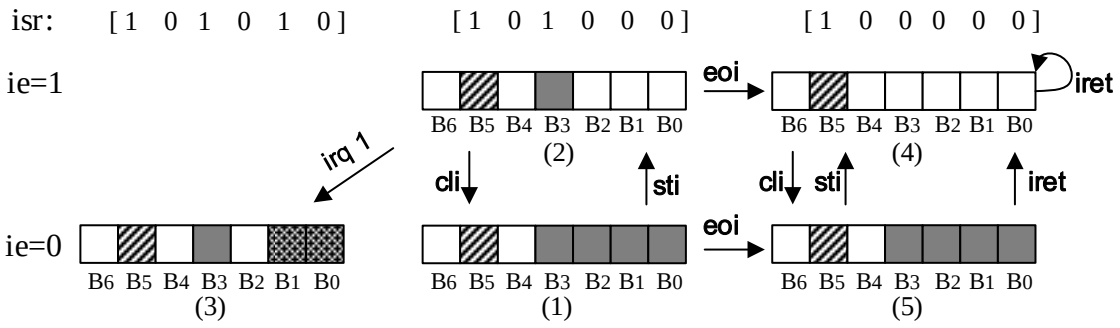

图 5.5　多级中断下的占有权转移

$$\frac{\begin{array}{c}O = (\eta_a, \eta_i, \theta) \quad \mathbb{O} = (\varphi, \varepsilon, \chi) \\ \chi; I \vdash \eta_i : \Gamma \quad \Gamma; \chi; I \vdash \eta_a : \varphi \quad \Gamma; \chi; I \vdash \theta : \varepsilon \\ \lfloor \psi \rfloor \Rightarrow I[0, N] * \mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * \mathsf{empE} \quad \mathsf{side}(I, \chi)\end{array}}{\vdash_\psi O : \mathbb{O}} \ (\textsc{TopRule})$$

$$\mathsf{side}(I, \chi) \stackrel{\text{def}}{=} \mathsf{GoodSc}(\chi) \wedge \mathsf{GoodI}(I) \quad \mathsf{GoodSc}(\chi) \stackrel{\text{def}}{=} \mathsf{sc\_local}(\chi) \wedge \mathsf{sc\_ind}(\chi)$$

$$\begin{aligned}
\mathsf{sc_ind}(\chi) &\stackrel{\text{def}}{=} \forall t, \Sigma.\ \chi\ \Sigma\ t \to \exists \alpha. \Sigma(\mathtt{tcbls}) = \alpha \wedge t \in \mathrm{dom}(\alpha) \\
\mathsf{sc_local}(\chi) &\stackrel{\text{def}}{=} \forall \Sigma_1, \Sigma_2, \Sigma, t.\ (\Sigma = \Sigma_1 \uplus \Sigma_2 \wedge \chi\ \Sigma_1\ t) \to \chi\ \Sigma\ t \\
\mathsf{GoodI}(I) &\stackrel{\text{def}}{=} \forall \sigma, \Sigma, \mathbf{s}, t.\ (\sigma, \Sigma, \mathbf{s}) \models \mathsf{SWINV}(I) \wedge \chi \triangleright t \to \\
&\qquad \exists \sigma', \Sigma'.\ \mathsf{set_t}(\sigma, \Sigma, t, \sigma', \Sigma') \wedge (\sigma', \Sigma', \mathbf{s}) \models \mathsf{SWINV}(I)
\end{aligned}$$

$$\begin{aligned}
\mathsf{set_t}(\sigma, \Sigma, t', \sigma', \Sigma') \stackrel{\text{def}}{=} \exists \nu, \mathbf{s}, p, t.\ &(\sigma, \Sigma, \chi) \models \mathtt{OSTCBCur} \mapsto \nu * \mathtt{ctid} \rightarrowtail t * p \wedge \\
&(\sigma', \Sigma', \chi) \models \mathtt{OSTCBCur} \mapsto t' * \mathtt{ctid} \rightarrowtail t' * p
\end{aligned}$$

图 5.6　顶层规则

5.3　推理规则

本节将结合占有权转移思想自顶向下的介绍推理规则。首先介绍顶层规则，然后介绍验证各个程序各个组成部分的验证，包括中断处理程序的验证，系统 API 的验证和内部函数的验证，最后介绍程序指令的推理规则和一些组合语句的规则。

5.3.1　顶层规则

图5.6给出了顶层规则 $\vdash_\psi O:\mathbb{O}$，用来验证在初始状态满足 ψ 的情况下，内核底层实现 O 关于高层抽象规范 $\mathbb{O}$ 的正确性定义。为了验证内核，首先需要提供内部函数 η_i 中函数的规范 Γ 和内核状态的不变式集合 I。Γ 定义在图5.7中。Γ 是从函数名到函数规范的映射，函数规范包含函数前条件 fp 和函数后条件 fq。

$$
\begin{array}{llcl}
(\mathit{LogicVal}) & \mathtt{lv} & \stackrel{\text{def}}{=} & \mathtt{v} \mid \bar{\mathtt{v}} \mid \cdots \qquad (\mathit{LogicVL}) \quad \mathcal{L} \stackrel{\text{def}}{=} \mathtt{nil} \mid \mathtt{lv} :: \mathcal{L} \\
(\mathit{FunPre}) & \mathtt{fp} & \in & \mathit{Vallist} \to \mathit{LogicVL} \to \mathit{Asrt} \\
(\mathit{FunPost}) & \mathtt{fq} & \in & \mathit{Vallist} \to \mathit{LogicVL} \to \mathit{Val} \cup \{\bot\} \to \mathit{Asrt} \\
(\mathit{FunSpec}) & \Gamma & \in & \mathit{FName} \rightharpoonup \mathit{FunPre} \times \mathit{FunPost}
\end{array}
$$

图 5.7　内部函数规范

函数的前条件 fp 是从值表和逻辑值表到程序断言的映射，因为函数每次调用时实参的值是不同的，所以这里用一个值表用来描述函数实参，逻辑值表主要用来描述函数对全局资源的变化。函数后条件需要描述返回值的信息，所以函数后条件比前条件多一个参数用来描述函数返回值，如果该函数没有返回值则为“ $\bot$ ”。后面在介绍函数调用的推理规则时会具体介绍如何定义内部函数规范并证明。

在确定了 Γ 和 I 后，需要通过内部函数规则 $\chi; \mathrm{I} \vdash \eta_i : \Gamma$、中断处理规则 $\Gamma; \chi; \mathrm{I} \vdash \theta : \varepsilon$ 和系统调用规则 $\Gamma; \chi; \mathrm{I} \vdash \eta_a : \varphi$ 分别验证内部函数、中断处理程序和系统 API 都满足各自的规范，这些规则会要求验证各自包含的每个函数，后面会具体介绍这三条规则。本文验证框架具有可组合验证的特点，就是指在验证一个操作系统时，可以根据顶层规则和上述三条规则将证明拆分成一个个独立的函数的证明，这些证明之间不会存在依赖，这使得证明的结构非常清楚，并且可以各个函数可以并行的验证。

规则还要求初始条件 ψ 能保证满足不变式 $\mathrm{I}[0, \mathrm{N}]$，每个任务中断相关的局部状态都满足初始条件 $\mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}]$，以及每个任务局部符号表一开始都是空。$\mathrm{I}[\mathfrak{n}, \mathfrak{m}]$ 定义在图5.8 中，表示 I 中 $\mathfrak{n}$ 到 $\mathfrak{m}$ 号不变式的分离连接。$\mathsf{OS}[\mathit{isr}, \mathit{ie}, \mathit{is}, \mathit{cs}]$ 用来描述每个中断相关的状态，同时要求当前正在处理的中断（*is* 的栈顶）是正在被服务的中断中优先级最高的（*isr* 中更高优先级的位都为 0）。$\lfloor \psi \rfloor$ 用来将 ψ 转换为关系型断言（定义在图5.8 中）。

同时规则对不变式 I 和调度策略 χ 有一些基本的要求。规则要求调度程序选择的任务一定是已经存在的任务（sc_ind），而且调度策略具有局部性（sc_local）。调度策略的局部性是指如果调度程序在一块小的抽象内核状态 Σ_1 上能选择出一个被调度到的任务 t，那么在包含 Σ_1 的更大的块 Σ 上 t 一定也能被调度到。调度策略的局部性要求了调度策略所关心的抽象内核模块是确定的，这个性质主要用于保证 Frame 规则的正确性，该规则可以帮助验证人员在推理的过程中去掉不关心的程序状态，使断言变得简单。GoodI 要求任务切换不会破坏全局不变式，这是为了保证每次任务被调度执行时全局资源都是良型的。$\mathsf{SWINV}(\mathrm{I})$ 的定义在图5.8 中。

$$
\begin{array}{lcl}
\mathsf{getD}(\eta, f) & \stackrel{\text{def}}{=} & \begin{cases} \mathsf{rev}(\mathcal{D}_1) +\!\!+ \mathcal{D}_2 & \text{if} \quad \eta(f) = (\tau, \mathcal{D}_1, \mathcal{D}_2, s) \\ \bot & \text{otherwise} \end{cases} \\
\mathsf{BuildP}(\mathcal{D}, \bar{v}) & \stackrel{\text{def}}{=} & \begin{cases} x \overset{\tau}{\mapsto}_l v * \mathsf{BuildP}(\mathcal{D}', \bar{v}') & \text{if} \quad \mathcal{D} = (x, \tau) :: \mathcal{D}' \wedge \bar{v} = v :: \bar{v}' \\ x \overset{\tau}{\mapsto}_l \_ * \mathsf{BuildP}(\mathcal{D}', \mathtt{nil}) & \text{if} \quad \mathcal{D} = (x, \tau) :: \mathcal{D}' \wedge \bar{v} = \mathtt{nil} \\ \mathtt{emp} & \text{if} \quad \mathcal{D} = \mathtt{nil} \\ \bot & \text{otherwise} \end{cases} \\
\mathsf{BuildR}(\mathcal{D}) & \stackrel{\text{def}}{=} & \mathsf{BuildP}(\mathcal{D}, \mathtt{nil}) \\
I[n, m] & \stackrel{\text{def}}{=} & \begin{cases} I(n) * I(n+1) * \ldots * I(m) & \text{if} \quad 0 \leqslant n \leqslant m \leqslant N \\ \mathtt{emp} & \text{otherwise} \end{cases} \\
\mathsf{OS}[isr, ie, is, cs] & \stackrel{\text{def}}{=} & \exists k.\, \mathsf{ISR}(isr) * \mathsf{IE}(ie) * \mathsf{IS}(is) * \mathsf{CS}(cs) * \llcorner k \lrcorner * \\
 & & \qquad (\forall k'.\, 0 \leqslant k' < k \rightarrow isr(k') = 0) \\
\lfloor \psi \rfloor & \stackrel{\text{def}}{=} & \lambda(\sigma, \Sigma, \mathbf{s}).\, \forall \Lambda.\, \psi\, \Lambda\, \Sigma \wedge \exists t.\, \Lambda|_t = \sigma \\
\mathsf{INV}(I, k) & \stackrel{\text{def}}{=} & \exists isr.\, \mathsf{ISR}(isr) * \\
 & & \qquad ((isr(k) = 1 \wedge \mathtt{emp}) \vee ((isr(k) = 0 \vee k = N) \wedge I(k))) \\
\mathsf{SWINV}(I) & \stackrel{\text{def}}{=} & \mathsf{ISR}(\bar{0}) * \mathsf{IE}(0) * (\exists\, k.\, \llcorner k \lrcorner * I[0, k])
\end{array}
$$

图 5.8　推理规则相关的辅助定义

$$
\frac{\begin{array}{c} dom(\eta) = dom(\Gamma) \quad \eta(f) = (\_, \_, \_, s) \quad \Gamma(f) = (fp, fq) \\ p = \mathsf{BuildFunP}(\eta, f, \bar{v}, \mathcal{L}, fp) \quad \rho = \mathsf{BuildFunR}(\eta, f, \bar{v}, \mathcal{L}, fq) \\ \Gamma; \chi; I; \rho; \mathsf{false} \vdash \left\{\ p\ \right\} s \left\{\ \mathsf{false}\ \right\} \end{array}}{\chi; I \vdash \eta : \Gamma}\ (\textsc{WfFun})
$$

$$
\begin{array}{lcl}
\mathsf{BuildFunP}(\eta_i, f, \bar{v}, \mathcal{L}, fp) & \stackrel{\text{def}}{=} & fp(\bar{v}, \mathcal{L}) * \mathsf{BuildP}(\mathsf{getD}(\eta_i, f), \bar{v}) \\
\mathsf{BuildFunR}(\eta_i, f, \bar{v}, \mathcal{L}, fq) & \stackrel{\text{def}}{=} & \lambda\, \hat{v}.\, fq(\bar{v}, \mathcal{L}, \hat{v}) * \mathsf{BuildR}(\mathsf{getD}(\eta_i, f))
\end{array}
$$

图 5.9　验证内部函数

5.3.2　内部函数的验证

图5.9给出了内部函数的验证规则 WfFun，这里首先要求对于每个内部函数 f 都要有一个对应的规范。然后对每个函数 f 通过 BuildFunP 和 BuildFunR 来构造其函数体语句的验证条件，并使用语句推理规则来验证。程序语句推理规则的格式如下：

$$\Gamma; \chi; I; \rho; p_i \vdash \left\{\ p\ \right\} s \left\{\ q\ \right\}$$

CSL-R 采用和并发分离逻辑风格的推理方式，I 用来描述共享资源，前后断言描述当前任务的局部资源。p 表示前断言，后断言包括 q、ρ 和 p_i，分别对应不同的程序退出方式，q 用于顺序组合，ρ 用于函数返回，p_i 用于中断返

$$\frac{\begin{array}{c}\mathrm{dom}(\theta)=\mathrm{dom}(\varepsilon)\\ p=\mathsf{BldItrpP}(k,\varepsilon,isr,is,I)\quad p_i=\mathsf{BldItrpR}(k,isr,is,I)\\ \Gamma;\chi;I;\lambda\hat{v}.\,\mathsf{false};p_i\vdash\{\ p\ \}\,\theta(k)\,\{\ \mathsf{false}\ \}\ \ \text{for all}\ k\in\{0,\ldots,N-1\}\end{array}}{\Gamma;\chi;I\vdash\theta:\varepsilon}\ (\textsc{Itrp})$$

$$\begin{array}{lcl}\mathsf{BldItrpP}(k,\varepsilon,isr,is,I) & \stackrel{\text{def}}{=} & \mathsf{OS}[isr\{k\rightsquigarrow 1\},0,k::is,\mathtt{nil}]*I[0,k]*[|\varepsilon(k)|]*\mathsf{empE}\\ \mathsf{BldItrpR}(k,isr,is,I) & \stackrel{\text{def}}{=} & \exists ie.\,\mathsf{OS}[isr\{k\rightsquigarrow 0\},ie,k::is,\mathtt{nil}]*\\ & & ((ie=1\wedge\mathtt{emp})\vee(ie=0\wedge I[0,k]))*[|\mathbf{end}\ \bot\ |]\end{array}$$

图 5.10　验证中断处理程序

回。对于内部函数的函数体语句，这里将 q 和 p_i 设置为 false 来强制必须通过 **return** 语句退出。

因为形参和局部变量的分配和初始化对于调用者来说是不可见的，所以无法描述在函数规范中，而函数的推理过程是从函数体 s 开始的，此时已经完成了形参和局部变量的分配和初始化；函数返回时也存在类似的问题。所以需要根据函数规范和函数中形参和局部变量的声明来构建函数体推理的前后断言。BuildFunP 跟据函数规范中的前条件 fp 和函数形参和局部变量的声明来构建函数体语句的前断言，BuildP 用来描述形参和局部变量分配和初始化完成后的内存状态。getD 用于获取函数形参和局部变量的声明，rev($\mathcal{D}$) 表示将形参声明表 $\mathcal{D}$ 翻转。

BuildFunR 跟据函数规范中的后条件 fq 和函数形参和局部变量的声明来构建函数返回时需要满足的断言，BuildR 用来描述函数返回前保存形参和局部变量的内存。因为函数的返回值取决于执行函数返回语句时返回值表达式的值，所以函数后条件 ρ 带着一个值作为参数来规范函数的返回值，在推理到函数返回语句 **return** e 时，会将 e 的值 v 传入 ρ 并要求后程序状态满足断言 $\rho\,v$。如果函数没有返回值，那么在执行 **return** 语句时会要求程序状态满足断言 $\rho\ \bot$。

BuildP、BuildR 和 getD 的定义在图5.8 中。

5.3.3　中断处理程序的验证

图5.10给出了中断处理程序的验证规则 Itrp，这里首先要求每个中断处理程序都有对应的规范（$\mathrm{dom}(\theta)=\mathrm{dom}(\varepsilon)$）。然后通过 BldtrpP 和 BldtrpR 构造出中断处理程序的前后断言，并验证中断处理程序满足其规范。

BldtrpP 表示在刚进入 k 号中断处理时，*isr* 中第 k 位为 1，$ie=0$ 表示中断关闭，*is* 栈顶为 k 表示当前在 k 号中断，并且 *isr* 中优先级比 k 高的中断对应的为都为 0，此时中断可以访问资源 I[0, k]，k 号中断精化的规范程序为 ε(k)。empE 表示中断处理程序一开始没有局部变量。一般内核中对于中断都有相对规范的处理流程，刚进入中断处理程序时用汇编代码保存上下文，然后调用一个 C 语言函数来实现该中断处理程序的主要功能，最后恢复上下文，所以这里

假定中断处理程序不会用汇编代码实现一些除了保存上下文之外的复杂的栈操作，中断处理程序一开始没有局部变量。

BldtrpR 要求中断退出前满足下列条件：（1）*isr* 中相应的位被清空；（2）如果中断打开（$ie = 1$），该中断处理程序不占有共享资源，否则需要保证占有的共享资源满足 $\mathrm{I}[0, \mathrm{k}]$（对应图5.5中的两种 **iret** 情况）；（3）前断言中的规范代码 $\varepsilon(\mathrm{k})$ 已经执行完，没有剩余的规范代码需要精化（$[|\textbf{end}\ \bot\ |]$），$\bot$ 表示中断处理程序没有返回值。

5.3.4　系统 API 的验证

$$\frac{\begin{array}{c}\mathrm{dom}(\eta) = \mathrm{dom}(\varphi) \quad \eta(\mathrm{f}) = (\_, \_, \_, \mathrm{s}) \quad \varphi(\mathrm{f}) = \omega \\ \mathrm{p} = \mathsf{BuildAPIP}(\eta, \mathrm{f}, \omega, \bar{\mathrm{v}}) \quad \rho = \mathsf{BuildAPIR}(\eta, \mathrm{f}) \\ \Gamma; \chi; \mathrm{I}; \rho; \mathsf{false} \vdash \{\ \mathrm{p}\ \}\, \mathrm{s}\, \{\ \mathsf{false}\ \}\end{array}}{\Gamma; \chi; \mathrm{I} \vdash \eta : \varphi}\ (\textsc{WfAPI})$$

$$\begin{array}{lcl}\mathsf{BuildAPIP}(\eta_a, \mathrm{f}, \omega, \bar{\mathrm{v}}) & \stackrel{\text{def}}{=} & \mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * \mathsf{BuildP}(\mathsf{getD}(\eta_a, \mathrm{f}), \bar{\mathrm{v}}) * [|\omega(\bar{\mathrm{v}})|] \\ \mathsf{BuildAPIR}(\eta_a, \mathrm{f}) & \stackrel{\text{def}}{=} & \lambda\, \hat{\mathrm{v}}.\, \mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * \\ & & \qquad \mathsf{BuildR}(\mathsf{getD}(\eta_a, \mathrm{f})) * [|\textbf{end}\ \hat{\mathrm{v}}|]\end{array}$$

图 5.11　验证系统 API

图5.11给出了验证系统 API 的规则，这里首先要求每个 API 函数 f 都有对应的规范（$\mathrm{dom}(\eta) = \mathrm{dom}(\varphi)$），然后验证每个 API 满足精化其对应的规范 $\varphi(\mathrm{f})$。BuildAPIP 和 BuildAPIR 用来构建系统 API 的前后断言。系统 API 开始时中断打开，并且没有中断处理程序在服务中（$\mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}]$）。因为系统 API 在非临界区执行时任何中断都有可能打断其执行，所以 API 开始执行时无法访问中断处理程序占有的资源（图5.5中的 $\mathbf{B_0} \sim \mathbf{B_{N-1}}$）；又由于抢占式内核的特点，中断处理程序中可能发生任务切换，系统 API 开始时也无法访问任务间共享的内核资源（$\mathbf{B_N}$）；所以系统 API 开始时不占有任何共享资源，其对共享资源的访问只能发生临界区中。BuildP 用来描述系统 API 开始执行时的参数和局部变量的状态，之前已经介绍过。$\omega(\bar{\mathrm{v}})$ 是需要精化的规范代码，其中 $\bar{\mathrm{v}}$ 是描述参数的值表。

BuildAPIR 表示系统 API 返回前需要满足以下要求：（1）中断相关的状态和刚进入时一样；（2）没有剩余需要精化的规范代码，且高层规范代码的返回值和底层相同（$[|\textbf{end}\ \hat{\mathrm{v}}|]$）。

5.3.5　程序语句推理规则

图5.12和图5.14中给出了 CSL-R 中一些重要的推理规则，图5.12中主要包含汇编原语指令的推理规则，图5.14中主要是 C 语句的推理规则和一些组合规则。

$$\frac{\mathrm{p} \Rightarrow \mathrm{p_i}}{\Gamma;\chi;\mathrm{I};\mathsf{false};\mathrm{p_i} \vdash \{\ \mathrm{p}\ \}\ \mathbf{iext}\ \{\ \mathsf{false}\ \}}\ (\textsc{iext})$$

$$\frac{\mathrm{p} = \mathsf{OS}[isr, 1, is, cs] * \llcorner \mathrm{k} \lrcorner * [|\mathbf{s}|]}{\Gamma;\chi;\mathrm{I};\rho;\mathrm{p_i} \vdash \{\ \mathrm{p}\ \}\ \mathbf{encrt}\ \{\ \mathsf{OS}[isr, 0, is, 1::cs] * \mathsf{INV}(\mathrm{I}, \mathrm{k}) * \mathrm{I}[0, \mathrm{k}-1] * [|\mathbf{s}|]\ \}}\ (\textsc{encrt})$$

$$\frac{}{\Gamma;\chi;\mathrm{I};\rho;\mathrm{p_i} \vdash \{\ \mathsf{OS}[isr, 0, is, cs] * [|\mathbf{s}|]\ \}\ \mathbf{encrt}\ \{\ \mathsf{OS}[isr, 0, is, 0::cs] * [|\mathbf{s}|]\ \}}\ (\textsc{encrt-0})$$

$$\frac{\mathrm{p} = \mathsf{OS}[isr, 0, is, 1::cs] * \llcorner \mathrm{k} \lrcorner * \mathsf{INV}(\mathrm{I}, \mathrm{k}) * \mathrm{I}[0, \mathrm{k}-1] * [|\mathbf{s}|]}{\Gamma;\chi;\mathrm{I};\rho;\mathrm{p_i} \vdash \{\ \mathrm{p}\ \}\ \mathbf{excrt}\ \{\ \mathsf{OS}[isr, 1, is, cs] * [|\mathbf{s}|]\ \}}\ (\textsc{excrt})$$

$$\frac{}{\Gamma;\chi;\mathrm{I};\rho;\mathrm{p_i} \vdash \{\ \mathsf{OS}[isr, 0, is, 0::cs] * [|\mathbf{s}|]\ \}\ \mathbf{excrt}\ \{\ \mathsf{OS}[isr, 0, is, cs] * [|\mathbf{s}|]\ \}}\ (\textsc{excrt-0})$$

$$\frac{\mathrm{p} = \mathsf{OS}[isr, 1, is, cs] * \llcorner \mathrm{k} \lrcorner * [|\mathbf{s}|]}{\Gamma;\chi;\mathrm{I};\rho;\mathrm{p_i} \vdash \{\ \mathrm{p}\ \}\ \mathbf{cli}\ \{\ \mathsf{OS}[isr, 0, is, cs] * \mathsf{INV}(\mathrm{I}, \mathrm{k}) * \mathrm{I}[0, \mathrm{k}-1] * [|\mathbf{s}|]\ \}}\ (\textsc{cli})$$

$$\frac{}{\Gamma;\chi;\mathrm{I};\rho;\mathrm{p_i} \vdash \{\ \mathsf{OS}[isr, 0, is, cs] * [|\mathbf{s}|]\ \}\ \mathbf{cli}\ \{\ \mathsf{OS}[isr, 0, is, cs] * [|\mathbf{s}|]\ \}}\ (\textsc{cli-0})$$

$$\frac{\mathrm{p} = \mathsf{OS}[isr, 0, is, cs] * \llcorner \mathrm{k} \lrcorner * \mathsf{INV}(\mathrm{I}, \mathrm{k}) * \mathrm{I}[0, \mathrm{k}-1] * [|\mathbf{s}|]}{\Gamma;\chi;\mathrm{I};\rho;\mathrm{p_i} \vdash \{\ \mathrm{p}\ \}\ \mathbf{sti}\ \{\ \mathsf{OS}[isr, 1, is, cs] * [|\mathbf{s}|]\ \}}\ (\textsc{sti})$$

$$\frac{}{\Gamma;\chi;\mathrm{I};\rho;\mathrm{p_i} \vdash \{\ \mathsf{OS}[isr, 0, is, cs] * [|\mathbf{s}|]\ \}\ \mathbf{sti}\ \{\ \mathsf{OS}[isr, 0, is, cs] * [|\mathbf{s}|]\ \}}\ (\textsc{sti-0})$$

$$\frac{0 \leqslant \mathrm{k} < \mathrm{N} \quad \mathrm{p} = \mathsf{OS}[isr, 1, \mathrm{k}::is, cs] * \mathrm{I}(\mathrm{k}) * [|\mathbf{s}|]}{\Gamma;\chi;\mathrm{I};\rho;\mathrm{p_i} \vdash \{\ \mathrm{p}\ \}\ \mathbf{eoi}\ \mathrm{k}\ \{\ \mathsf{OS}[isr\{\mathrm{k} \rightsquigarrow 0\}, 1, \mathrm{k}::is, cs] * [|\mathbf{s}|]\ \}}\ (\textsc{eoi})$$

$$\frac{0 \leqslant \mathrm{k} < \mathrm{N} \quad \mathrm{p} = \mathsf{OS}[isr, 0, \mathrm{k}::is, cs] * [|\mathbf{s}|]}{\Gamma;\chi;\mathrm{I};\rho;\mathrm{p_i} \vdash \{\ \mathrm{p}\ \}\ \mathbf{eoi}\ \mathrm{k}\ \{\ \mathsf{OS}[isr\{\mathrm{k} \rightsquigarrow 0\}, 0, \mathrm{k}::is, cs] * [|\mathbf{s}|]\ \}}\ (\textsc{eoi-0})$$

$$\frac{\mathrm{p} \Leftrightarrow \mathsf{SWINV}(\mathrm{I}) * \mathsf{IS}(is) * \mathsf{CS}(cs)}{\Gamma;\chi;\mathrm{I};\rho;\mathrm{p_i} \vdash \{\ (\mathrm{p} * [|\mathbf{sched}; \mathbf{s}|]) \wedge \chi \triangleright \mathrm{x}\ \}\ \mathbf{switch}\ \mathrm{x}\ \{\ \mathrm{p} * [|\mathbf{s}|]\ \}}\ (\textsc{switch})$$

图 5.12　部分语句规则 I

下面将解释这些规则的含义。

IEXT规则用于中断退出指令，其只要求当前状态下中断退出后条件 p_i 成立。ENCRT规则表现了在中断打开的情况下进入临界区时的发生的占有权转移。假设当前处于中断 k，进入临界区后阻止了高优先级的 $0 \sim k-1$ 号中断被响应，因此当前任务会获得 $I[0, k-1]$ 的占有权。另外还有两种情况会获得 $I(k)$ 的占有权：（1）如果当前不在中断处理程序中（$k = N$），那么此时可以获得任务间的共享资源 $I[N]$ 的占有权；（2）当前中断执行过 **eoi** 指令释放了 $I(k)$ 的占有权，此时进入临界区会重新占有权（图5.5中状态 4 进入临界区变为状态 5 的情况）。$\mathsf{INV}(I, k)$ 根据是否满足上述两种情况来决定是否获得 $I[k]$ 的占有权，其定义在图5.8中。因为支持嵌套临界区，允许在临界区中再次进入临界区，ENCRT-0用于处理重复进入临界区的情况，此时不会发生占有权转移。

EXCRT规则是 ENCRT规则的逆过程，其释放进入临界区时占有的资源（对应图5.5中 **sti** 指令）。EXCRT-0 规则用于临界区嵌套的情况。

cli/sti 指令的规则和 **encrt/excrt** 指令的占有权转移语义是一样的，但是 **cli/sti** 直接改变 *ie* 的状态，并不利用 *cs* 保存 *ie* 的历史信息。

EOI规则表示如果在中断打开的情况下，执行 **eoi** k 会释放 $I(k)$ 的占有权（对应图5.5中状态 2 执行 **eoi** 到状态 4 的情况）。这里注意推理规则要求执行 **eoi** k 时 *is* 的栈顶必须为 k，表明只允许中断处理程序清空自己在 *isr* 中对应的位，而不允许改变其他中断 *isr* 中的状态。EOI-0规则表示中断关闭的情况下执行 **eoi** 不会发生占有权转移（对应图5.5 中状态 1 执行 **eoi** 到状态 5 的情况），因为此时低优先级的中断仍然无法被响应。

SWITCH规则要求在执行任务切换之前 $\mathsf{SWINV}(I)$ 要被满足，并且在任务切换回来后 $\mathsf{SWINV}(I)$ 仍然成立。$\mathsf{SWINV}(I)$ 定义在图5.8 中，要求中断必须关闭并且 *isr* 中每一位都为 0，也就是说任务切换只能发生在非中断处理程序中或者最外层中断处理程序执行完 **eoi** 指令之后。因为切换到其他任务后，被切换到的任务可能通过进入临界区的方式访问共享资源，所以需要保证当前任务占有的资源是满足不变式 $I[0, k]$。该规则同时要求任务切换不会改变任务局部的中断状态 *is*、*cs*。

为了保证精化关系的成立，需要保证底层机器和高层机器当前正在运行的任务号是相同的，所以 SWITCH规则要求在底层代码执行任务切换的同时高层抽象规范也要执行一次抽象调度指令 **sched**，而且 $\chi \triangleright x$ 要求抽象调度策略 χ 可以调度到 x 中保存的任务号，这样保证高层机器和底层机器可以调度到相同的任务执行。

SEQ规则用于顺序语句的推理，IF规则用于处理 C 语言中的条件分支语句。IF规则首先要求前条件 p 保证条件表达式 e 可以计算出一个值 v，并且 v 不是未定义的（图5.2中 $e =_\tau v$ 的语义），然后在条件表达式的值为真（$\mathsf{istrue}(v)$）和为假（$\mathsf{isfalse}(v)$ ）的两个分支下分别推理。v 等于 0 或者 `Vnull` 时 $\mathsf{isfalse}(v)$

$$\mathsf{isfalse}(v) \stackrel{\text{def}}{=} (v = 0 \vee v = \texttt{Vnull}) \qquad \mathsf{istrue}(v) \stackrel{\text{def}}{=} \neg\mathsf{isfalse}(v)$$

$$\bar{e} =_{\mathcal{T}} \bar{v} \stackrel{\text{def}}{=} \begin{cases} e =_{\tau} v \wedge \bar{e}' =_{\mathcal{T}'} \bar{v}' & \text{iff} \quad \bar{e} = e::\bar{e}' \wedge \bar{v} = v::\bar{v}' \wedge \mathcal{T} = \tau::\mathcal{T}' \\ \mathsf{true} & \text{iff} \quad \bar{e} = \texttt{nil} \wedge \mathcal{T} = \texttt{nil} \wedge \bar{v} = \texttt{nil} \\ \mathsf{false} & otherwise \end{cases}$$

$$\tau_1 \propto \tau_2 \stackrel{\text{def}}{=} \begin{cases} True \quad \text{iff} \quad ((\tau_1 = \texttt{Tptr}(\tau) \vee \tau_1 = \texttt{Tcomptr}(\mathsf{id})) \wedge \\ \qquad\qquad (\tau_2 = \texttt{Tptr}(\tau') \vee \tau_2 = \texttt{Tnull})) \vee \\ \qquad\qquad ((\tau_1 = \texttt{Tint8} \vee \tau_1 = \texttt{Tint16} \vee \tau_1 = \texttt{Tint32}) \wedge \\ \qquad\qquad (\tau_2 = \texttt{Tint8} \vee \tau_2 = \texttt{Tint16} \vee \tau_2 = \texttt{Tint32})) \\ False \quad otherwise \end{cases}$$

图 5.13　语句规则相关的辅助定义

成立，其他情况下 istrue(v) 成立。isfalse(v) 和 istrue(v) 的定义5.13 中。

WHILE规则思想和传统 hoare 逻辑类似，需要找到一个循环不变式 p，并保证循环体 s 不会破坏循环不变式，和if规则类似，这里循环不变式需要保证 e 可以计算出一个值 v，且 v 不是未定义的。在 while 语句执行结束后，循环表达式 e 的值满足 isfalse。

ASSIGN规则用于处理赋值语句 $e_1 = e_2$，这里前条件需要保证能够计算出左表达式 e_1 表示的地址（$\&e_1 =_{\tau_1} a$），同时需要保证右表达式能够计算出值（$e_2 =_{\tau_2} v_2$）。$\tau_1 \propto \tau_2$（定义在图5.13中）要求两个表达式的类型必须是匹配的。这里前条件中还必须包含将要赋值的地址对应的内存（$a \overset{\tau_1}{\longmapsto} v_1$）。ASSIGN规则是通用的赋值语句的规则，$e_1$ 可以是变量表达式，结构体成员变量、数组成员变量等等。在很多情况下 ASSIGN规则用起来并不方便，例如对结构体成员变量赋值时，需要展开 Astruct 的定义得到描述该成员变量内存的断言 $a \overset{\tau_1}{\longmapsto} v_1$，而展开 Astruct 的定义会使得断言变得非常复杂。所以基于 ASSIGN 规则，验证框架中还包含了很多辅助推理规则来处理各种不同的情况，并将其应用于自动证明策略中，在第七章会具体介绍。

RETE规则用于带返回值的函数返回语句 **return** e。这里需要保证返回值表达式能够正确计算出一个值 v，并且前断言 p 需要满足结合返回值的函数后条件（$\rho\, v$）。由于 **return** e 后续的顺序语句不会再执行，所以此时用于顺序连接的后断言为 false。RET规则用于不带返回值的函数返回语句。

CALL规则用于函数调用语句的处理，其首先要求内部函数规范 Γ 中包含被调用的函数 f 的规范（$\Gamma(f) = (fp, fq)$），然后需要保证参数表达式能够正确计算出值（$\bar{e} =_{\mathcal{T}} \bar{v}$），最后要求前断言 p 中包含 f 所需的函数前条件（$fp\ \bar{v}\ \mathcal{L}$）和剩下的不被访问的部分 q。函数调用结束后，会得到函数的后条件。由于在“$p_1 * p_2$” 的语义中，p_1 和 p_2 描述的是同一个 *isr*、*is*、*ie*、*cs* 和 **s**，所以要求不被访问的部分 q 不能描述 *isr*、*is*、*ie*、*cs* 和 **s**，否则如果被调用函数修改了这些内容会导致规则不正确。CALLE 规则可以看作是 ASSIGN规则和 CALL规则的结合，

$$\frac{\Gamma;\chi;I;\rho;p_i \vdash \{\ p_1\ \}\, s_1 \,\{\ p_2\ \} \quad \Gamma;\chi;I;\rho;p_i \vdash \{\ p_2\ \}\, s_2 \,\{\ p_3\ \}}{\Gamma;\chi;I;\rho;p_i \vdash \{\ p_1\ \}\, s_1; s_2 \,\{\ p_3\ \}}\ (\textsc{seq})$$

$$\frac{\begin{array}{c} p \Rightarrow e =_\tau v \quad \Gamma;\chi;I;\rho;p_i \vdash \{\ p \wedge \mathsf{istrue}(v)\ \}\, s_1 \,\{\ q\ \} \\ \Gamma;\chi;I;\rho;p_i \vdash \{\ p \wedge \mathsf{isfalse}(v)\ \}\, s_2 \,\{\ q\ \} \end{array}}{\Gamma;\chi;I;\rho;p_i \vdash \{\ p\ \}\, \textbf{if}\ (e)\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \,\{\ q\ \}}\ (\textsc{if})$$

$$\frac{p \Rightarrow e =_\tau v \quad \Gamma;\chi;I;\rho;p_i \vdash \{\ p \wedge \mathsf{istrue}(v)\ \}\, s \,\{\ p\ \}}{\Gamma;\chi;I;\rho;p_i \vdash \{\ p\ \}\, \textbf{while}\ (e)\ s \,\{\ p \wedge \mathsf{isfalse}(v)\ \}}\ (\textsc{while})$$

$$\frac{p * a \overset{\tau_1}{\longmapsto} v_1 \Rightarrow \&e =_{\tau_1} a \wedge e_2 =_{\tau_2} v_2 \quad \tau_1 \propto \tau_2}{\Gamma;\chi;I;\rho;p_i \vdash \left\{\ p * a \overset{\tau_1}{\longmapsto} v_1 * [|\mathbf{s}|]\ \right\} e_1 = e_2 \left\{\ p * a \overset{\tau_1}{\longmapsto} v_2 * [|\mathbf{s}|]\ \right\}}\ (\textsc{assign})$$

$$\frac{p \Rightarrow (\rho\, v \wedge e =_\tau v)}{\Gamma;\chi;I;\rho;\mathsf{false} \vdash \{\ p\ \}\, \textbf{return}\ e \,\{\ \mathsf{false}\ \}}\ (\textsc{rete})$$

$$\frac{p \Rightarrow \rho\ \ \bot}{\Gamma;\chi;I;\rho;\mathsf{false} \vdash \{\ p\ \}\, \textbf{return} \,\{\ \mathsf{false}\ \}}\ (\textsc{ret})$$

$$\frac{\begin{array}{c} \Gamma(f) = (fp, fq) \quad p \Rightarrow (fp\ \bar{v}\ \mathcal{L}) * q \quad p \Rightarrow \bar{e} =_{\mathcal{T}} \bar{v} \\ q \text{ does not specify } ie, is, cs, isr \text{ and } \mathbf{s} \end{array}}{\Gamma;\chi;I;\rho;p_i \vdash \{\ p\ \}\, f(\bar{e}) \,\{\ \exists \hat{v}.\, (fq\ \bar{v}\ \mathcal{L}\ \hat{v}) * q\ \}}\ (\textsc{call})$$

$$\frac{\begin{array}{c} \Gamma(f) = (fp, fq) \quad p \Rightarrow (fp\ \bar{v}\ \mathcal{L}) * a \overset{\tau}{\mapsto} v * q \quad p \Rightarrow \bar{e} =_{\mathcal{T}} \bar{v} \\ a \overset{\tau}{\mapsto} v * q \Rightarrow \&e =_\tau a \qquad q \text{ does not specify } ie, is, cs, isr \text{ and } \mathbf{s} \end{array}}{\Gamma;\chi;I;\rho;p_i \vdash \{\ p\ \}\, e = f(\bar{e}) \left\{\ \exists v'.\, (fq\ \bar{v}\ \mathcal{L}\ v') * a \overset{\tau}{\mapsto} v' * q\ \right\}}\ (\textsc{calle})$$

$$\frac{p \Rightarrow p' \quad \Gamma;\chi;I;\rho;p_i \vdash \{\ p'\ \}\, s \,\{\ q'\ \} \quad q' \Rightarrow q}{\Gamma;\chi;I;\rho;p_i \vdash \{\ p\ \}\, s \,\{\ q\ \}}\ (\textsc{conseq})$$

$$\frac{\Gamma;\chi;I;\rho;p_i \vdash \{\ p_1\ \}\, s \,\{\ p_2\ \} \quad q \text{ does not specify } ie, is, cs, isr \text{ and } \mathbf{s}}{\Gamma;\chi;I;\rho * q;p_i * q \vdash \{\ p_1 * q\ \}\, s \,\{\ p_2 * q\ \}}\ (\textsc{frm})$$

$$\frac{p \Rrightarrow p' \quad \Gamma;\chi;I;\rho;p_i \vdash \{\ p'\ \}\, s \,\{\ q'\ \} \quad q' \Rrightarrow q}{\Gamma;\chi;I;\rho;p_i \vdash \{\ p\ \}\, s \,\{\ q\ \}}\ (\textsc{abscsq})$$

图 5.14　部分语句规则 II

INT32U f (INT32U x) :

```
1  INT32U y;
2  OS_ENTER_CRITICAL();
3  y = add(x);
4  OS_EXIT_CRITICAL();
5  return y;
```

INT32U add (INT32U x) :

```
1  cnt = cnt + x;
2  return cnt;
```

$$\mathrm{I} = [\,\exists\, v.\mathtt{cnt} \xmapsto{\mathtt{Tint32}} v * \mathsf{CNT} \rightarrowtail v;\ \mathtt{emp}\,]$$

$$\varphi(\mathtt{f}) = \lambda\, v.\ \mathsf{ADD}(v) \qquad \Gamma(\mathtt{add}) = (\mathsf{addp}, \mathsf{addr})$$

$$\mathsf{ADD}(v) \stackrel{\text{def}}{=} \lambda\, \Sigma, (\hat{v}, \Sigma).\ \exists V.\ \Sigma(\mathsf{CNT}) = V \wedge \Sigma'(\mathsf{CNT}) = V + v \wedge \hat{v} = V + v$$

$$\mathsf{addp} \stackrel{\text{def}}{=} \lambda\, \bar{v}, \mathcal{L}.\ \exists\, v\, vc.\ \bar{v} = v :: \mathtt{nil} \wedge \mathcal{L} = vc :: \mathbf{s} :: \mathtt{nil} \wedge [|\mathbf{s}|] * \mathtt{cnt} \xmapsto{\mathtt{Tint32}} vc * \mathsf{OS}[\bar{0}, 0, \mathtt{nil}, 1 :: cs]$$

$$\mathsf{addr} \stackrel{\text{def}}{=} \lambda\, \bar{v}, \mathcal{L}, \hat{v}.\ \exists\, v\, vc.\ \bar{v} = v :: \mathtt{nil} \wedge \mathcal{L} = vc :: \mathbf{s} :: \mathtt{nil} \wedge [|\mathbf{s}|] * \mathtt{cnt} \xmapsto{\mathtt{Tint32}} vc + v * \mathsf{OS}[\bar{0}, 0, \mathtt{nil}, 1 :: cs] \wedge \hat{v} = vc + v$$

图 5.15　使用推理规则验证程序

只不过赋值的内容是函数的返回值。

CONSEQ规则允许验证人员减弱前条件和增强后条件。FRM规则允许去掉当前推理中不被访问的资源 q，从而简化证明过程，这里 q 同样不能描述 *isr*、*is*、*ie*、*cs* 和 **s**。

参考 [9]，ABSCSQ 规则看起来和 CONSEQ规则类似，但是其允许执行一次抽象规范代码。抽象蕴含 $\mathrm{p} \Rrightarrow \mathrm{p}'$ 定义如下：

$$\forall \sigma, \Sigma, \mathbf{s}.\ ((\sigma, \Sigma, \mathbf{s}) \models \mathrm{p}) \Longrightarrow \exists \Sigma', \mathbf{s}'.\ \Big((\mathbf{s}, \Sigma) \bullet\!\!\rightarrow\!\!\!\!\!\!\vdash\!\!\!\!\!\!\triangleright^{*} (\mathbf{s}', \Sigma')\Big) \wedge ((\sigma, \Sigma', \mathbf{s}') \models \mathrm{p}')$$

给定一个满足断言 p 的关系型状态 $(\sigma, \Sigma, \mathbf{s})$，抽象代码 **s** 在抽象状态 σ 下可以执行 0 或多步直到状态 $(\Sigma', \mathbf{s}')$，并且执行结束后的关系型状态 $(\sigma, \Sigma', \mathbf{s}')$ 满足 p'。这条规则允许在推理的过程中改变抽象规范代码和抽象内核状态，来建立底层和高层之间的模拟关系（simulation），从而保证精化关系的成立。

5.3.6　一个例子

在介绍完了主要的规则之后，下面来看一个简单的例子，结合该例子可以更好地理解这些推理规则。假设有一个系统 API 函数 f（定义在图5.15中），f 有一个参数 x 和一个局部变量 y，f 调用一个函数 add，并将其返回值赋值给 y，并返回 y。函数 add 带一个参数，add 将全局变量 cnt 的值加上该参数的值并重新赋值给变量 cnt，最后返回新的 cnt 的值。

f 精化的高层代码为 $\varphi(f) = \lambda\, v.\mathsf{ADD}(v)$，ADD 带一个参数 v，其将抽象内核状态中的一个计数器 CNT 加 v，并返回新的 CNT 的内容。

addp 和 addr 是内部函数规范 Γ 中保存的函数 add 的前后条件。这里为了简化，要求前后条件中中断相关的状态都满足 $\mathsf{OS}[\bar{0}, 0, \mathtt{nil}, 1 :: cs]$，其实可以将其放入逻辑值表中，保证前后中断相关状态不发生改变。参数值表 $\bar{v} = v :: \mathtt{nil}$ 表示该函数只有一个参数，其值为 v。因为 CALLE 规则给被调用函数的前后条件传入相同的逻辑值表作为参数，所以可以用逻辑值表来描述一些状态在函数调用前后的变化，这里 $\mathcal{L} = vc :: \mathbf{s} :: \mathtt{nil}$ 表示我们关心一个全局状态值 vc （对应的是全局变量 cnt 的初始值）和抽象规范语句 $\mathbf{s}$ 的变化。addr 中 $\hat{v} = vc + v$ 表示返回值是将全局变量 cnt 的初始值和参数的值 v 相加，$cnt \xmapsto{\mathtt{Tint32}} vc + v$ 表示该函数会将参数的值和 cnt 的初始值相加并重新赋给 cnt，$[|\mathbf{s}|]$ 表示抽象规范语句没有执行。可以看出前后条件清楚地反映了该函数的行为。

这里假设系统中只有一个中断，中断和任务共享资源为 $\exists\, v.cnt \mapsto v * \mathsf{CNT} \rightarrowtail v$，其要求底层机器上的全局变量 cnt 和高层抽象内核状态 CNT 的内容始终相同，任务间的共享资源为 emp，表示任务间没有共享资源。

图5.16给出了 f 的推理过程。首先根据图5.11中定义的 BuildAPIP 和 BuildAPIR 来构造 f 的前后断言 $\mathsf{fpre}(v)$ 和 $\mathsf{fpost}(v)$：

$$\mathsf{fpre}(v) \stackrel{\text{def}}{=} \mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * x \xmapsto{\mathtt{Tint32}}_{l} v * y \xmapsto{\mathtt{Tint32}}_{l} \_ * [|\mathsf{ADD}(v)|]$$
$$\mathsf{fpost}(v) \stackrel{\text{def}}{=} \lambda\, \hat{v}.\ \mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * x \xmapsto{\mathtt{Tint32}}_{l} \_ * y \xmapsto{\mathtt{Tint32}}_{l} \_ * [|\mathbf{end}\ \hat{v}|]$$

根据 WFAPI规则，对于 API 函数 f, 需要完成的如下推理：

$$\Gamma; \chi; I; \mathsf{fpost}(v); \mathsf{false} \vdash \left\{\ \mathsf{fpre}(v)\ \right\} s_f \left\{\ \mathsf{false}\ \right\}$$

因为 f 只有一个参数，为了方便简洁，此处将保存参数的值链 $v :: \mathtt{nil}$ 简写为 v，v 是自由变量，表示对任意的参数下面的推理都要成立，s_f 代表 f 中除了局部变量声明剩下的函数语句。图中箭头 $\Downarrow$ 表示根据 CONSEQ规则变换前断言，$\Downarrow\!\!\Downarrow$ 表示应用 ABSCEQ 规则让高层规范代码执行一次。根据断言语义可知：$x \overset{\tau}{\mapsto} v \iff \exists a.\ a \overset{\tau}{\mapsto} v \wedge x@(a, \tau)$；$x \overset{\tau}{\mapsto}_{l} v \iff \exists a.\ a \overset{\tau}{\mapsto} v \wedge x@_{l}(a, \tau)$；$\forall p.\ p * \mathtt{emp} \iff p$；这几个关系在 f 的推理中会被用到。

这里简单看下图5.16中的推理过程，首先应用 ENCRT 规则，会获得共享资源 $(\exists\, vc.cnt \xmapsto{\mathtt{Tint32}} vc * \mathsf{CNT} \rightarrowtail vc * \mathtt{emp})$，并将中断相关的状态变为 $\mathsf{OS}[\bar{0}, 0, \mathtt{nil}, 1 :: \mathtt{nil}]$。然后为了便于应用 CALLE 规则，先用 CONSEQ规则将前断言作了一次转化，具体的转化过程不再叙述。然后应用 CALLE 规则并进行了相应的转化。在应用 EXCRT 规则之前，需要保证即将释放的资源满足 $(\exists\, vc.cnt \xmapsto{\mathtt{Tint32}} vc * \mathsf{CNT} \rightarrowtail vc) * \mathtt{emp}$，因为函数 add 修改了全局变量 cnt 的值，所以此时需要应用 ABSCSQ规则让高层规范代码执行一次。然后应用 EXCRT 释放占有的全局资源。最后应用 RETE 规则完成推理。

INT32U f (INT32U x) :

1 INT32U y;

$\boxed{\mathsf{fpre}(v)}$

2 OS_ENTER_CRITICAL();

$$\boxed{\mathsf{OS}[\bar{0}, 0, \mathtt{nil}, 1 :: \mathtt{nil}] * x \xmapsto{\mathtt{Tint32}}_{\iota} v * y \xmapsto{\mathtt{Tint32}}_{\iota} \_ * [|\mathsf{ADD}(v)|] * (\exists\, vc.cnt \xmapsto{\mathtt{Tint32}} vc * \mathrm{CNT} \rightarrowtail vc) * \mathtt{emp}}$$

$$\Downarrow$$

$$\boxed{\mathsf{addp}(v :: \mathtt{nil}, vc :: \mathsf{ADD}(v) :: \mathtt{nil}) * a \xmapsto{\mathtt{Tint32}} \_ * y@_{\iota}(a, \mathtt{Tint32}) * x \xmapsto{\mathtt{Tint32}} v * \mathrm{CNT} \rightarrowtail vc}$$

3 y = add(x);

$$\boxed{\exists\, v'.\ \mathsf{addr}(v :: \mathtt{nil}, vc :: \mathsf{ADD}(v) :: \mathtt{nil}, v') * a \xmapsto{\mathtt{Tint32}} v' * y@_{\iota}(a, \mathtt{Tint32}) * x \xmapsto{\mathtt{Tint32}} v * \mathrm{CNT} \rightarrowtail vc}$$

$$\Downarrow$$

$$\boxed{\exists\, vc.\mathsf{OS}[\bar{0}, 0, \mathtt{nil}, 1 :: \mathtt{nil}] * x \xmapsto{\mathtt{Tint32}}_{\iota} v * y \xmapsto{\mathtt{Tint32}}_{\iota} (vc + v) * [|\mathsf{ADD}(v)|] * (cnt \xmapsto{\mathtt{Tint32}} vc + v * \mathrm{CNT} \rightarrowtail vc)}$$

$$\Downarrow$$

$$\boxed{\exists\, vc.\mathsf{OS}[\bar{0}, 0, \mathtt{nil}, 1 :: \mathtt{nil}] * x \xmapsto{\mathtt{Tint32}}_{\iota} v * y \xmapsto{\mathtt{Tint32}}_{\iota} (vc + v) * [|\mathbf{end}(vc + v)|] * (cnt \xmapsto{\mathtt{Tint32}} vc + v * \mathrm{CNT} \rightarrowtail vc + v)}$$

4 OS_EXIT_CRITICAL();

$$\boxed{\mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * x \xmapsto{\mathtt{Tint32}}_{\iota} v * y \xmapsto{\mathtt{Tint32}}_{\iota} (vc + v) * [|\mathbf{end}(vc + v)|]}$$

$$\Downarrow$$

$$\boxed{\mathsf{fpost}(v)\ (vc + v)}$$

5 return y;

图 5.16　函数 f 的推理过程

INT32U add (INT32U x) :

$$\boxed{\exists\, vc, \mathbf{s}.\mathsf{OS}[\bar{0}, 0, \mathtt{nil}, 1 :: \mathtt{nil}] * x \overset{\mathtt{Tint32}}{\longmapsto}_{\iota} v * cnt \overset{\mathtt{Tint32}}{\longmapsto} vc * [|\mathbf{s}|] \wedge \mathcal{L} = vc :: \mathbf{s} :: \mathtt{nil}}$$

1 cnt=cnt+x;

$$\boxed{\exists\, vc, \mathbf{s}.\mathsf{OS}[\bar{0}, 0, \mathtt{nil}, 1 :: \mathtt{nil}] * x \overset{\mathtt{Tint32}}{\longmapsto}_{\iota} v * cnt \overset{\mathtt{Tint32}}{\longmapsto} vc + v * [|\mathbf{s}|] \wedge \mathcal{L} = vc :: \mathbf{s} :: \mathtt{nil}}$$

$$\Downarrow$$

$$\boxed{\mathsf{apost}(v, \mathcal{L})\ (vc + v)}$$

2 return cnt;

图 5.17　内部函数 add 的推理过程

下面来看 $\mathtt{add}$ 函数，首先根据图5.9中定义的 BuildFunP 和 BuildFunR 来构造 $\mathtt{add}$ 的前后断言 $\mathsf{apre}(v, \mathcal{L})$ 和 $\mathsf{apost}(v, \mathcal{L})$：

$$\mathsf{apre}(v, \mathcal{L}) \overset{\text{def}}{=} \exists\, vc, \mathbf{s}.\mathsf{OS}[\bar{0}, 0, \mathtt{nil}, 1 :: \mathtt{nil}] * x \overset{\mathtt{Tint32}}{\longmapsto}_{\iota} v * cnt \overset{\mathtt{Tint32}}{\longmapsto} vc * [|\mathbf{s}|] \wedge \mathcal{L} = vc :: \mathbf{s} :: \mathtt{nil}$$

$$\mathsf{apost}(v, \mathcal{L}) \overset{\text{def}}{=} \lambda\, \hat{v}.\exists\, vc, \mathbf{s}.\mathsf{OS}[\bar{0}, 0, \mathtt{nil}, 1 :: \mathtt{nil}] * x \overset{\mathtt{Tint32}}{\longmapsto}_{\iota} v * cnt \overset{\mathtt{Tint32}}{\longmapsto} vc + c * [|\mathbf{s}|] \wedge \hat{v} = vc + v \wedge \mathcal{L} = vc :: \mathbf{s} :: \mathtt{nil}$$

根据 WFFUN规则，对于内部函数 $\mathtt{add}$ 需要完成如下推理：

$$\Gamma; \chi; I; \mathsf{apost}(v, \mathcal{L}); \mathsf{false} \vdash \left\{\ \mathsf{apre}(v, \mathcal{L})\ \right\} s_{\mathtt{add}} \left\{\ \mathsf{false}\ \right\}$$

因为这里只有一个参数，和之前一样，将保存参数的值表简写为 v，这里 v 和 $\mathcal{L}$ 是自由变量，表示在任意的 v 和 $\mathcal{L}$ 下推理都要成立。$s_{\mathtt{add}}$ 表示 $\mathtt{add}$ 的函数体语句。这里通过 $\mathcal{L}$ 保存了如下信息：（1）该函数不会执行抽象规范语句，$\mathbf{s}$ 前后不变；（2）全局变量 cnt 的内存被修改，其在内存中的值被加上了参数 x 的值。理论上来说，也可以用 $\mathcal{L}$ 描述参数的状态，但是为了含义明确，这里没有将用来保存参数信息的值表和 $\mathcal{L}$ 统一。通过逻辑值表，在推理 f 中的 $\mathtt{add}(x)$ 语句时才不会丢失信息并正确的推理下去。图5.17 给出了 $\mathtt{add}$ 的推理过程。

5.4　可靠性证明

定理 5.4.1 (推理系统可靠性). $\vdash_{\psi} O{:}\mathbb{O} \Longrightarrow O \sqsubseteq_{\psi} \mathbb{O}$.

上节介绍了推理系统，定理5.4.1给出了推理系统正确性的定义，其表示经过推理系统验证的操作系统（$\vdash_{\psi} O : \mathbb{O}$）都满足操作系统正确性定义（$O \sqsubseteq_{\psi} \mathbb{O}$）。由于精化关系不具有可组合性，因此很难直接验证基于精化关系

$$
\begin{array}{lcl}
\sigma \perp \sigma' & \stackrel{\text{def}}{=} & \sigma = ((G, E, M), isr, \delta) \wedge \sigma' = ((G, E, M'), isr, \delta) \wedge M \perp M' \\
I\{n, m\} & \stackrel{\text{def}}{=} & \begin{cases} \mathsf{INV}(I, n) * \mathsf{INV}(I, n+1) * \ldots * \mathsf{INV}(I, m) & \text{if} \quad 0 \leqslant n \leqslant m \leqslant N \\ \mathtt{emp} & \text{otherwise} \end{cases} \\
\lfloor I \rfloor & \stackrel{\text{def}}{=} & ((\mathsf{IE}(1) * I\{0, N\}) \vee (\mathsf{IE}(0) * (\exists\, k. \llcorner k \lrcorner * I\{k+1, N\}))
\end{array}
$$

图 5.18　程序模拟中的一些辅助定义

定义的操作系统正确性 $O \sqsubseteq_{\psi} \mathbb{O}$。参考 [8] 的工作，本文通过构建一系列具有可组合性的程序模拟（simulation）来完成定理5.4.1 的证明。本节将介绍如何使用程序模拟技术证明推理系统的可靠性。下面我们在5.4.1节先介绍推理规则中各种逻辑判断（Judgement）的语义，然后在5.4.2节证明定理5.4.1。

5.4.1　逻辑判断的语义

定义5.4.1通过内核方法模拟（定义5.4.2）给出了语句逻辑判断 $\Gamma; \chi; I; \rho; p_i \vdash \left\{\ p\ \right\} s \left\{\ q\ \right\}$ 的语义 $\Gamma; \chi; I; \rho; p_i \models \{p\}s\{q\}$。

定义 5.4.1 (语句逻辑判断语义). $\Gamma; \chi; I; \rho; p_i \models \{p\}s\{q\}$ 成立，当且仅当，对于任意的 σ, Σ 和 $\mathbf{s}$，如果 $(\sigma, \Sigma, \mathbf{s}) \models p$ 成立，那么 $\Gamma; \chi; I; \rho; p_i; q \models ((s, (\circ, \bullet)), \sigma) \preceq (\mathbf{s}, \Sigma)$。

$\Gamma; \chi; I; \rho; p_i \models \{p\}s\{q\}$ 要求对于任意满足前断言 p 的关系型状态 $(\sigma, \Sigma, \mathbf{s})$，底层代码 s 和高层规范 $\mathbf{s}$ 都需要满足内核方法模拟关系。由于推理规则是以程序语句为单位进行，而模拟关系考虑执行中的每一步，一个程序语句会执行多步，所以这里需要加上了一个程序后继来保存程序语句执行的中间状态。

定义 5.4.2 (内核方法模拟). 内核方法模拟是满足下述性质的最大关系。若 $\Gamma; \chi; I; \rho; p_i; q \models (C, \sigma) \preceq (\mathbf{s}, \Sigma)$ 成立，则下述全部成立：

- ***Normal Steps:*** 对于任意的 P、C'、σ_s、Σ_s、σ_1 和 σ_1'，如果 $C \neq (\textbf{\textit{fexec}}(\_, \_), \_)$，$(\sigma_s, \Sigma_s, \_) \models \lfloor I \rfloor$，$\sigma_1 = \sigma \uplus \sigma_s$，$\Sigma \perp \Sigma_s$ 和 $P \vdash (C, \sigma_1) \bullet\!\!-\!\!\text{L}\!\!\triangleright (C', \sigma_1')$ 成立，那么存在 Σ_s'、$\mathbf{s}'$、Σ'、σ' 和 σ_s'，使得下述内容成立：
  - $\sigma_1' = \sigma' \uplus \sigma_s'$，$(\sigma_s', \Sigma_s', \_) \models \lfloor I \rfloor$,
  - $(\mathbf{s}, \Sigma \uplus \Sigma_s) \bullet\!\!-\!\!\text{H}\!\!\triangleright^{*} (\mathbf{s}', \Sigma' \uplus \Sigma_s')$,
  - $\Gamma; \chi; I; \rho; p_i; q \models (C', \sigma') \preceq (\mathbf{s}', \Sigma')$.
- ***Function Call:*** 对于任意的 σ_s、Σ_s、κ_s、f 和 $\bar{v}$，如果 $C = (\textbf{\textit{fexec}}(f, \bar{v}), (\circ, \kappa_s))$、$\sigma \perp \sigma_s$、$(\sigma_s, \Sigma_s, \_) \models \lfloor I \rfloor$ 和 $\Sigma \perp \Sigma_s$ 成立，那么存在 σ_1、σ_f、Σ_1、Σ_f、Σ'、Σ_s'、$\mathbf{s}'$、$\mathcal{L}$、fp 和 fq，使得下述内容成立:
  - $\Gamma(f) = (fp, fq)$
  - $(\mathbf{s}, \Sigma \uplus \Sigma_s) \bullet\!\!-\!\!\text{H}\!\!\triangleright^{*} (\mathbf{s}', \Sigma' \uplus \Sigma_s')$，$(\sigma_s, \Sigma_s', \mathbf{s}') \models \lfloor I \rfloor$
  - $\sigma = \sigma_1 \uplus \sigma_f$，$\Sigma' = \Sigma_1 \uplus \Sigma_f$，$(\sigma_1, \Sigma_1, \mathbf{s}') \models fp\ (\mathsf{rev}(\bar{v}))\ \mathcal{L}$

  - 对于任意 $\hat{v}$、σ'、σ'_1、Σ''、Σ'_1 和 $\mathbf{s}''$，如果 $\sigma.m.G = \sigma'.m.G$、$\sigma.m.E = \sigma'.m.E$、$(\sigma'_1, \Sigma'_1, \mathbf{s}'') \models \mathsf{fq}\ (\mathsf{rev}(\bar{v}))\ \mathcal{L}\ \hat{v}$、$\sigma' = \sigma'_1 \uplus \sigma_f$ 和 $\Sigma'' = \Sigma'_1 \uplus \Sigma_f$ 成立，那么 $\Gamma; \chi; I; \rho; p_i; q \models ((\mathbf{skip}, (\circ, \kappa_s)), \sigma') \preceq (\mathbf{s}'', \Sigma'')$

- ***ContextSwitch:*** 对于任意 σ_s、κ_s、x 和 Σ_s，如果 $\sigma \perp \sigma_s$、$C = (\mathbf{switch}\ x, (\circ, \kappa_s))$、$(\sigma_s, \Sigma_s, \_) \models \lfloor I \rfloor$ 和 $\Sigma \perp \Sigma_s$ 成立，那么存在 σ_1、σ'、Σ_1、Σ'、Σ'_s 和 $\mathbf{s}'$，使得下述内容成立：

  - $(\mathbf{s}, \Sigma \uplus \Sigma_s) \bullet\!\!\rightarrow^{*}_{H} (\mathbf{sched}; \mathbf{s}', \Sigma_1 \uplus \Sigma' \uplus \Sigma'_s)$
  - $(\sigma_s, \Sigma'_s, \_) \models \lfloor I \rfloor$，$\sigma = \sigma_1 \uplus \sigma'$
  - $(\sigma', \Sigma', \_) \models \mathsf{SWINV}(I) \wedge (\chi \triangleright x)$
  - 对于任意 σ''、σ'''、Σ'' 和 Σ'''，如果 $\sigma''' = \sigma_1 \uplus \sigma''$、$\Sigma''' = \Sigma_1 \uplus \Sigma''$ 和 $(\sigma'', \Sigma'', \_) \models \mathsf{SWINV}(I)$ 成立，那么 $\Gamma; \chi; I; \rho; p_i; q \models ((\mathbf{skip}, (\circ, \kappa_s)), \sigma''') \preceq (\mathbf{s}', \Sigma''')$ 成立

- ***Skip:*** 对于任意 σ_s 和 Σ_s，如果 $C = (\mathbf{skip}, (\circ, \bullet))$、$\sigma \perp \sigma_s$、$(\sigma_s, \Sigma_s, \_) \models \lfloor I \rfloor$ 和 $\Sigma \perp \Sigma_s$ 成立，那么存在 Σ'、Σ'_s 和 $\mathbf{s}'$，使得 $(\mathbf{s}, \Sigma \uplus \Sigma_s) \bullet\!\!\rightarrow^{*}_{H} (\mathbf{s}', \Sigma' \uplus \Sigma'_s)$、$(\sigma_s, \Sigma'_s, \_) \models \lfloor I \rfloor$ 和 $(\sigma, \Sigma', \mathbf{s}') \models q$ 成立

- ***IRet:*** 对于任意 κ_s、σ_s 和 Σ_s，如果 $C = (\mathbf{iext}, (\circ, \kappa_s))$、$\lfloor \kappa_s \rfloor = \perp$、$\lfloor \kappa_s \rfloor_c = \perp$、$\sigma \perp \sigma_s$、$(\sigma_s, \Sigma_s, \_) \models \lfloor I \rfloor$ 和 $\Sigma \perp \Sigma_s$，那么存在 Σ'、Σ'_s 和 $\mathbf{s}'$，使得 $(\mathbf{s}, \Sigma \uplus \Sigma_s) \bullet\!\!\rightarrow^{*}_{H} (\mathbf{s}', \Sigma' \uplus \Sigma'_s)$、$(\sigma_s, \Sigma'_s, \_) \models \lfloor I \rfloor$ 和 $(\sigma, \Sigma', \mathbf{s}') \models p_i$ 成立

- ***ReturnE*** 和 ***Return*** 的情况和 ***IRet*** 类似，此处不再展开

- ***Abort:*** 对于任意 P、Σ_s、σ_s 和 σ'，如果 $C \neq (\mathit{fexec}(\_, \_), \_)$、$C \neq (\mathbf{switch}\ \_, \_)$、$\sigma' = \sigma \uplus \sigma_s$、$(\sigma_s, \Sigma_s, \_) \models \lfloor I \rfloor$ 和 $\Sigma \perp \Sigma_s$ 成立，那么 $\neg(P \vdash (C, \sigma') \bullet\!\!\rightarrow_{L} \mathbf{abort})$ 成立

具有可组合性的内核方法模拟融合了 RGSim 对于开放环境下精化关系的证明技术和 CSL 占有权转移的思想。其使用 $\lfloor I \rfloor$ (定义在图5.18中)，而不是 RGSim 中使用的依赖 -保证条件，来约束中断处理程序以及任务代码对共享资源的访问所需要遵循的原则。$\lfloor I \rfloor$ 的定义从另一个方面反映了图5.5中描述的占有权转移语义。$\lfloor I \rfloor$ 精确地描述了所有的全局共享资源（$I[0, N]$）中未被占有的部分（图5.5中白色的方块代表的资源），因为这部分未被占有的资源时刻可能被某个任务或中断占有，所以 $\lfloor I \rfloor$ 保证这部分资源是良型的。

在内核代码执行时，因为可能执行 **encrt**、**excrt** 等指令，这些指令会改变 *ie*、*isr* 的状态从而改变占有权，所以在考虑内核代码执行时，会在每一步带上满足 $\lfloor I \rfloor$ 的共享资源 $(\sigma_s, \Sigma_s, \_)$，并在每一步结束时保证此时存在 $(\sigma', \Sigma', \_)$ 满足 $\lfloor I \rfloor$，并将 $(\sigma'_s, \Sigma'_s, \_)$ 重新放回共享资源中，由于 $\lfloor I \rfloor$ 是随着 *ie*、*isr* 的状态动态变化的，$(\sigma'_s, \Sigma'_s, \_)$ 和 $(\sigma_s, \Sigma_s, \_)$ 虽然都满足 $\lfloor I \rfloor$，但其对应的资源已经发生变化，从而发生占有权转移。例如，图5.5中，状态（1）执行开中断操作进入状态（2），虽然（1）（2）中白色的块都满足 $\lfloor I \rfloor$，但（1）中有一部分的共享资源被当前任务占有。体现在内核方法模拟中，一开始获得的 $(\sigma_s, \Sigma_s, \_)$ 对应状态（1）中的白色的块，而 $(\sigma'_s, \Sigma'_s, \_)$ 对应状态（2）中白色的块，（1）（2）中相差

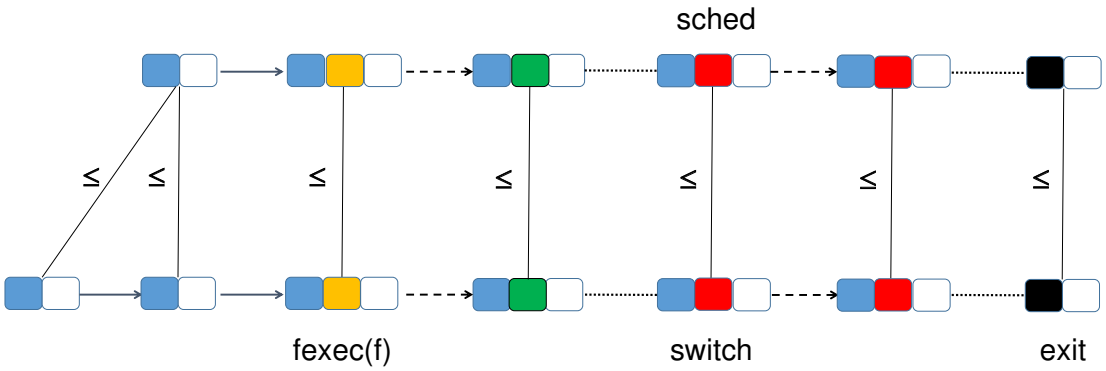

图 5.19　内核方法模拟示意图

的那部分白色块被当前任务占有。也就是说，在内核方法模拟中，通过动态变化的 $\lfloor I \rfloor$ 来体现占有权转移思想。

图5.19体现了内核方法模拟 $\Gamma; \chi; I; \rho; p_i; q \models (C, \sigma) \preceq (\mathbf{s}, \Sigma)$ 的主要思想，图中白色的块表示共享资源（满足 $\lfloor I \rfloor$），其他块表示任务局部资源。下面将结合图5.19来理解定义5.4.2中的各种情况：

- 如果当前代码不是调用内部函数，并且共享关系型状态 $(\sigma_s, \Sigma_s, \_)$（图5.19中白色的块）满足 $\lfloor I \rfloor$，而且底层代码能够在 $\sigma \uplus \sigma_s$ 上执行一步到新的状态 σ_1'，那么高层抽象规范代码能够在 $\Sigma \uplus \Sigma_s$ 下执行 0 步或多步到 Σ_1'，并且执行后的状态 $(\sigma_1', \Sigma_1', \_)$ 中可以划分出新的一块 $(\sigma_s', \Sigma_s', \_)$ 满足 $\lfloor I \rfloor$。

- 为了支持函数的可组合证明，也即每个内部函数已经被证明过满足其规范，在调用时不需要再去重新验证。当遇到内部函数调用时，首先允许高层代码先执行 0 步或多步（为了 ABSCSQ规则的正确性），但是要求高层代码的执行不会破坏共享资源满足 $\lfloor I \rfloor$ 的性质。然后要求当前局部资源中包含被调用函数的前条件中所需要的资源（图5.19中黄色的块），这部分资源用来保证被调用函数能够正确执行。被调用函数返回后会交还一部分资源（图5.19中绿色的块），如果交还的资源满足函数的后条件，那么要求模拟关系继续成立。

- 为了保证程序模拟的可组合性，需要保证底层和高层的当前任务号是相同的。因此要求每次底层执行任务切换时，高层必须也要执行一次调度，并且二者能够选择相同的任务执行。并且由于每次执行任务切换时，被切换到的任务的局部中断状态是未知的，所以需要保证每个任务在被切换到时，其他所有任务不占有任何共享资源并且所有的共享资源都是良型的（共享资源满足 $I[0, N]$）。为了保持该性质，在执行切换前需要保证所有的共享资源是良型的，$\lfloor I \rfloor$ 只保证了不被任务占有的共享资源是良型的，所以需要保证当前任务占有的其他共享资源是良型的。这里要求局部状态中

有一块满足 SWINV(I) （图5.19中红色的块，定义在图5.8中），SWINV(I) 表示任务切换只能发生在最外层中断或非中断程序中，并且中断关闭，同时也保证了当前任务占有的共享资源是良型的，从 SWINV(I) 和 $\lfloor \mathrm{I} \rfloor$ 的定义可知，$\mathsf{SWINV}(\mathrm{I}) * \lfloor \mathrm{I} \rfloor \Longrightarrow \mathrm{I}[0, \mathrm{N}]$。并且要求当前任务被重新调度到后，如果共享资源是良型的，那么程序模拟关系继续成立。

- 如果底层代码为 $(\textbf{skip}, (\circ, \bullet))$，表示底层机器没有代码继续执行，此时允许高层代码可以继续执行 0 或多步，且最后局部状态和高层规范代码需要满足后条件 q。

- 如果底层代码为中断处理程序，并且即将执行中断返回原语（$\mathrm{C} = (\textbf{iext}, (\circ, \kappa_s))$），那么此时高层代码可以执行 0 或多步，并且最后局部状态满足 $\mathrm{p_i}$。函数返回和中断处理程序返回类似。**skip**，**iext** 和函数返回语句等可以理解为当前推理结束的标志，对应图5.19中 exit 的情况，黑色的块表示局部状态需要满足后条件。

- 如果共享资源是良型的，内核代码在局部资源和共享资源 $\sigma \uplus \sigma_s$ 上执行不会出错。这里不仅仅要求程序在全局状态上执行不会出错，同时要求了当前任务代码不会去访问其他任务的局部资源。程序执行出错定义如下：$\mathrm{P} \vdash (\mathrm{C}, \sigma) \bullet\!\!-\!\!\mathrm{L}\!\!\triangleright \textbf{abort} \stackrel{\text{def}}{=} \neg \exists \mathrm{C}', \sigma'. \mathrm{P} \vdash (\mathrm{C}, \sigma) \bullet\!\!-\!\!\mathrm{L}\!\!\triangleright (\mathrm{C}', \sigma')$。

定理 5.4.2 (语句逻辑判断可靠性). $\Gamma; \chi; \mathrm{I}; \rho; \mathrm{p_i} \vdash \{\ \mathrm{p}\ \}\, \mathrm{s}\, \{\ \mathrm{q}\ \} \Longrightarrow \Gamma; \chi; \mathrm{I}; \rho; \mathrm{p_i} \models \{\mathrm{p}\}\mathrm{s}\{\mathrm{q}\}$.

基于语句逻辑判断的语义，定义5.4.3给出了中断逻辑判断 $\Gamma; \chi; \mathrm{I} \vdash \theta : \varepsilon$ 的语义 $\Gamma; \chi; \mathrm{I} \models \theta : \varepsilon$。

定义 5.4.3 (中断逻辑判断语义). $\Gamma; \chi; \mathrm{I} \models \theta : \varepsilon$ 成立，当且仅当，$\mathrm{dom}(\theta) = \mathrm{dom}(\varepsilon)$，并且对于任意 k，*isr*，*is*，p，s 和 $\mathrm{p_i}$，如果 $\varepsilon(\mathrm{k}) = \mathbf{s}$，$\theta(\mathrm{k}) = \mathrm{s}$，$\mathrm{p} = \mathsf{BldItrpP}(\mathrm{k}, \mathbf{s}, \mathit{isr}, \mathit{is}, \mathrm{I})$，$\mathrm{p_i} = \mathsf{BldItrpR}(\mathrm{k}, \mathit{isr}, \mathit{is}, \mathrm{I})$，那么 $\Gamma; \chi; \mathrm{I}; \mathsf{false}; \mathrm{p_i} \models \{\mathrm{p}\}\mathrm{s}\{\mathsf{false}\}$

根据 WfInt规则和定理5.4.2，容易得到：

定理 5.4.3 (中断逻辑判断可靠性). $\Gamma; \chi; \mathrm{I} \vdash \theta : \varepsilon \Longrightarrow \Gamma; \chi; \mathrm{I} \models \theta : \varepsilon$.

类似的，可以得到系统调用逻辑判断和内部函数逻辑判断的语义及正确性。

定义 5.4.4 (系统调用逻辑判断语义). $\Gamma; \chi; \mathrm{I} \models \eta_a : \varphi$ 成立，当且仅当，$\mathrm{dom}(\eta_a) = \mathrm{dom}(\varphi)$，并且对于任意的 f、$\bar{\nu}$、$\omega$、p 和 ρ，如果 $\varphi(\mathrm{f}) = \omega$，$\mathrm{p} = \mathsf{BuildAPIP}(\eta_a, \mathrm{f}, \omega, \bar{\nu})$，$\rho = \mathsf{BuildAPIR}(\eta_a, \mathrm{f})$，$\eta_a(\mathrm{f}) = (\_, \_, \_, \mathrm{s})$，那么 $\Gamma; \chi; \mathrm{I}; \rho; \mathsf{false} \models \{\mathrm{p}\}\mathrm{s}\{\mathsf{false}\}$。

定理 5.4.4 (系统调用逻辑判断可靠性). $\Gamma; \chi; \mathrm{I} \vdash \eta_a : \varphi \Longrightarrow \Gamma; \chi; \mathrm{I} \models \eta_a : \varphi$。

定义 5.4.5 (内部函数逻辑判断语义). $\chi; \mathrm{I} \models \eta_i : \Gamma$ 成立，当且仅当，$\mathrm{dom}(\Gamma) = \mathrm{dom}(\eta_i)$，并且对于任意的 f、s、fp、fq、$\bar{\nu}$、$\mathcal{L}$、p 和 ρ，如果 $\eta_i(\mathrm{f}) = (\_, \_, \_, \mathrm{s})$，$\Gamma(\mathrm{f}) = (\mathrm{fp}, \mathrm{fq})$，$\mathrm{p} = \mathsf{BuildFunP}(\eta_i, \mathrm{f}, \bar{\nu}, \mathcal{L}, \mathrm{fp})$ 和 $\rho = \mathsf{BuildFunR}(\eta_i, \mathrm{f}, \bar{\nu}, \mathcal{L}, \mathrm{fq})$ 成立，那么 $\Gamma; \chi; \mathrm{I}; \rho; \mathsf{false} \models \{\mathrm{p}\}\mathrm{s}\{\mathsf{false}\}$。

定理 5.4.5 (内部函数逻辑判断可靠性). $\chi; \mathrm{I} \vdash \eta_i : \Gamma \Longrightarrow \chi; \mathrm{I} \models \eta_i : \Gamma$。

5.4.2　推理系统可靠性的证明

上节给出了各种逻辑判断的语义和可靠性定理，为了证明定理5.4.1验证框架中还定义了辅助证明的全局模拟关系（定义5.4.6）和任务模拟关系（定义5.4.7）。全局模拟关系用于证明可见事件精化关系，任务模拟关系用于证明全局模拟关系，并且任务模拟关系可以由内核方法模拟关系推出。通过这几个程序模拟关系，可以建立起推理规则和操作系统正确性之间的关系。

定义 5.4.6 (全局模拟)**.** 全局模拟式满足下述性质的最大关系，如果 $(P, W) \preceq (\mathbb{P}, \mathbb{W})$ 成立，则下述全部成立：

- 对于任意的 W'，如果 $P \vdash W =\!\!\Rightarrow_{L} W'$ 成立，那么存在 $\mathbb{W}'$，使得下述内容成立：
  - $\mathbb{P} \vdash \mathbb{W} =\!\!\Rightarrow_{H}^{*} \mathbb{W}'$，$W'.\Delta = \mathbb{W}'.\Delta$
  - $(P, W') \preceq (\mathbb{P}, \mathbb{W}')$
- 对于任意的 W' 和 ϑ，如果 $P \vdash W =\!\!\Rightarrow_{L,\vartheta} W'$ 成立，那么存在 $\mathbb{W}'$，使得下述内容成立：
  - $\mathbb{P} \vdash \mathbb{W} =\!\!\overset{\vartheta}{\Rightarrow}_{H}^{*} \mathbb{W}'$，$W'.\Delta = \mathbb{W}'.\Delta$,
  - $(P, W') \preceq (\mathbb{P}, \mathbb{W}')$.
- 如果 $P \vdash W =\!\!\Rightarrow_{L}$ **abort**，那么 $\mathbb{P} \vdash \mathbb{W} =\!\!\Rightarrow_{H}^{*}$**abort**

全局模拟描述了执行过程中底层机器和高层机器需要保持的性质。如果底层机器 W 执行一步到状态 W'，那么高层机器 $\mathbb{W}$ 可以执行 0 步或多步到状态 $\mathbb{W}'$，并且全局模拟关系继续成立。如果底层机器执行中产生了一个可见事件 ϑ，那么高层机器一定也要产生相同的可见事件。如果底层机器出错，那么高层机器执行也会在 0 步或多步后执行出错。因为在操作系统验证中，底层机器和高层机器中应用代码相同，所以要求在执行中底层机器和高层机器的用户状态始终相同。

定理 5.4.6 (全局模拟蕴含精化关系)**.** $(P, W) \preceq (\mathbb{P}, \mathbb{W}) \Longrightarrow (P, W) \preccurlyeq (\mathbb{P}, \mathbb{W})$.

定理5.4.6表示如果 (P, W) 和 $(\mathbb{P}, \mathbb{W})$ 满足全局模拟关系，那么 (P, W) 和 $(\mathbb{P}, \mathbb{W})$ 满足可见事件精化关系，其将程序执行序列上的性质转化为验证执行中每一步需要保持的某个性质，简化了证明的复杂性。定理5.4.6可以通过对底层机器全局操作语义归纳来证明。

定义 5.4.7 (任务模拟)**.** 任务模拟是满足下述性质的最大关系，如果 $P; \mathbb{P}; I; p \models (C_l, \sigma) \preceq (C_h, \Sigma)$ 成立，则下述全部成立：

- **Normal Steps:** 对于任意的 C_l'、m、m'、σ_s、Σ_s、σ_1 和 σ_1'，如果 $(\sigma_s, \Sigma_s, \_) \models \lfloor I \rfloor$，$\sigma_1 = \sigma \uplus \sigma_s$，$\Sigma \perp \Sigma_s$ 和 $P \vdash (C_l, m, \sigma_1) —_{L}\!\!\rightarrow (C_l', m', \sigma_1')$ 成立，那么存在 Σ_s'、C_h'、Σ'、σ' 和 σ_s'，使得下述内容成立：

  - $\sigma_1' = \sigma' \uplus \sigma_s'$，$(\sigma_s', \Sigma_s', \_) \models \lfloor I \rfloor$
  - $\mathbb{P} \vdash (C_h, m, \Sigma \uplus \Sigma_s) -\!\!\!\rightarrow^{*} (C_h', m', \Sigma' \uplus \Sigma_s')$
  - $P; \mathbb{P}; I; p \models (C_l', \sigma') \preceq (C_h', \Sigma')$

- **Event Steps:** 对于任意的 ϑ、C_l'、m、m'、σ_s、Σ_s、σ_1 和 σ_1'，如果 $(\sigma_s, \Sigma_s, \_) \models \lfloor I \rfloor$，$\sigma_1 = \sigma \uplus \sigma_s$，$\Sigma \perp \Sigma_s$ 和 $P \vdash (C_l, m, \sigma_1) -\!\!\!\xrightarrow{\vartheta}\!\!\!\rightarrow (C_l', m', \sigma_1')$ 成立，那么存在 Σ_s'、C_h'、Σ'、σ' 和 σ_s'，使得下述内容成立：
  - $\sigma_1' = \sigma' \uplus \sigma_s'$，$(\sigma_s', \Sigma_s', \_) \models \lfloor I \rfloor$
  - $\mathbb{P} \vdash (C_h, m, \Sigma \uplus \Sigma_s) -\!\!\!\xrightarrow{\vartheta}_{H}\!\!\!\rightarrow^{*} (C_h', m', \Sigma' \uplus \Sigma_s')$，
  - $P; \mathbb{P}; I; p \models (C_l', \sigma') \preceq (C_h', \Sigma')$

- **Context Switch:** 对于任意的 m、σ_s、κ_s、x 和 Σ_s，如果 $\sigma \perp \sigma_s$、$C = (\textbf{switch}\ x, (\circ, \kappa_s))$、$(\sigma_s, \Sigma_s, \_) \models \lfloor I \rfloor$ 和 $\Sigma \perp \Sigma_s$ 成立，那么存在 σ_1、σ'、Σ'、Σ_1、Σ_s'、χ、$\mathbf{s}'$ 和 K，使得下述内容成立
  - $\mathbb{P} = (\_, (\_, \_, \chi))$
  - $\mathbb{P} \vdash (C_h, m, \Sigma \uplus \Sigma_s) -\!\!\!\rightarrow^{*} ((\textbf{sched}; \mathbf{s}', K), m, \Sigma_1 \uplus \Sigma' \uplus \Sigma_s')$
  - $(\sigma_s, \Sigma_s', \_) \models \lfloor I \rfloor$，$\sigma = \sigma_1 \uplus \sigma'$
  - $(\sigma', \Sigma', \_) \models \mathsf{SWINV}(I) \wedge (\chi \triangleright x)$
  - 对于任意的 σ''、σ'''、Σ'' 和 Σ'''，如果 $\sigma''' = \sigma_1 \uplus \sigma''$，$\Sigma''' = \Sigma_1 \uplus \Sigma''$ 和 $(\sigma'', \Sigma'', \_) \models \mathsf{SWINV}(I)$ 成立，那么 $P; \mathbb{P}; I; p \models ((\textbf{skip}, (\circ, \kappa_s)), \sigma''') \preceq ((\mathbf{s}', K), \Sigma''')$

- **Skip:** 对于任意的 m、σ_s 和 Σ_s，如果 $C = (\textbf{skip}, (\circ, \bullet))$，$\sigma \perp \sigma_s$，$(\sigma_s, \Sigma_s, \_) \models \lfloor I \rfloor$ 并且 $\Sigma \perp \Sigma_s$，那么存在 Σ'、Σ_s'，使得 $(\sigma_s, \Sigma_s', \_) \models \lfloor I \rfloor$，$(\sigma, \Sigma', \_) \models p$ 和 $\mathbb{P} \vdash (C_h, m, \Sigma \uplus \Sigma_s) -\!\!\!\rightarrow^{*} ((\textbf{skip}, (\circ, \bullet)), m, \Sigma' \uplus \Sigma_s')$ 成立

- **Abort:** 对于任意的 m、Σ_s、σ_s、σ' 和 Σ'，如果 $\sigma' = \sigma \uplus \sigma_s$，$\Sigma' = \Sigma \uplus \Sigma_s$，$(\sigma_s, \Sigma_s, \_) \models \lfloor I \rfloor$ 和 $P \vdash (C_l, m, \sigma') -\!\!\!\rightarrow_{L}$ **abort** 成立，那么 $\mathbb{P} \vdash (C_h, m, \Sigma') -\!\!\!\rightarrow_{H}^{*}$ **abort**

因为用户创建的任务代码不仅会执行内核代码，还会执行应用程序代码，要保证全局模拟不仅仅需要考虑每个任务执行内核代码时底层和高层机器需要保持的性质，还要保证在执行应用程序代码时底层和高层机器需要保持的性质。任务模拟描述了底层机器和高层机器中的每个任务在执行过程中需要保持的性质。任务模拟是在内核方法模拟和全局模拟之间建立关系的桥梁。任务模拟和内核方法模拟在结构上类似，但有以下几点不同需要注意：

- 任务模拟中的一步考虑的是任务操作语义的一步，其可能是在执行应用程序代码改变用户状态，也可能执行内核代码改变内核状态。由于底层和高层的应用程序代码是相同的，区别在于系统调用时，底层会进入内核代码执行，高层会去执行系统调用的抽象规范代码。所以要求底层和高层的用户状态 m 始终保持一致，这很容易保持，因为初始状态保证底层和高层

用户状态相同，而且底层内核代码和高层抽象规范不会修改用户状态，只需要在执行应用程序代码时，保持底层每走一步高层也执行相同的一步即可。

- 内核代码在执行时如果发生函数调用那么一定是调用内部函数，并且每个内部函数在 Γ 中都有对应的规范，但是任务在执行过程中不仅仅调用内部函数，还可能调用系统 API 和用户自己实现的函数，所以在任务模拟中不去模块化处理函数调用，函数调用只是普通的一步执行。

- 任务入口函数不会返回，如果任务入口函数返回则会出错。而由于底层和高层每个任务的入口函数都是相同的，如果底层机器因为在用户任务入口函数中执行函数返回语句而出错，那么高层机器也会出现同时错误，所以和内核方法模拟不同任务模拟中不需要单独考虑函数返回的情况。

- 内核代码执行中不会产生外部可见事件，但应用程序代码中可能执行 **print** e 产生一个外部可见事件，此时要求高层也会产生相同的外部事件。

- 任务模拟定义中的 $\mathfrak{p}$ 是任务创建和执行结束时内核状态需要满足的规范。很多操作系统会在任务创建时分配一些局部资源，$\mathfrak{p}$ 可以根据不同的系统而不同，但是 $\mathfrak{p}$ 至少需要保证 $\mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * \mathsf{empE}$。$isr = \bar{0}$ 表示不允许在中断处理程序中进行任务创建，并且任务局部的中断状态满足初始值 $(1, \mathtt{nil}, \mathtt{nil})$，而且由于任务刚创建时是在任务代码中执行，此时内核状态中该任务的局部符号表是空（empE）。

引理 5.4.1 (任务模拟的可组合性). 对于任意的 P、$\mathbb{P}$、ψ、T_l、T_h、Δ、t_c、I、Γ、Λ 和 Σ，如果下述内容成立:

(1) $P = (A, (\eta_a, \eta_i, \theta))$，$\mathbb{P} = (A, (\varphi, \varepsilon, \chi))$，$\Lambda = ((G, \Pi, M), isr, \pi)$

(2) $t_c \in \mathrm{dom}(T)$，$t_c = \Sigma(\mathtt{ctid})$，$\mathrm{dom}(T_l) = \mathrm{dom}(\Sigma(\mathtt{tcbls}))$

(3) $T_l = \{t_1 \rightsquigarrow C_{l1}, \ldots, t_n \rightsquigarrow C_{ln}\}$，$T_h = \{t_1 \rightsquigarrow C_{h1}, \ldots, t_n \rightsquigarrow C_{hn}\}$

(4) $\chi; I \vDash \eta_i : \Gamma$，$\mathsf{side}(I, \chi)$

(5) $M = M_1 \uplus M_2 \uplus \cdots \uplus M_n \uplus M_s$，$\Sigma = \Sigma_1 \uplus \Sigma_2 \uplus \cdots \uplus \Sigma_n \uplus \Sigma_s$

(6) $\Lambda|_{t_c} = \sigma_c$，$(\sigma_c \lhd M_s, \Sigma_s, \_) \vDash \lfloor I \rfloor$

(7) $P; \mathbb{P}; I; \mathfrak{p} \vDash (C_{lc}, \sigma_c \lhd M_c) \preceq (C_{hc}, \Sigma_c)$

(8) 对于任意的 i、σ_i、σ_r、Σ_r，如果 $i \neq c$，$\sigma_i = (\Lambda|_{t_i}) \lhd M_i$，$(\sigma_r, \Sigma_r, \_) \vDash \mathsf{SWINV}(I)$，$\sigma_i \perp \sigma_r$ 和 $\Sigma_i \perp \Sigma_r$ 成立，那么 $P; \mathbb{P}; I; \mathfrak{p} \vDash (C_{li}, \sigma_i \uplus \sigma_r) \preceq (C_{hi}, \Sigma_i \uplus \Sigma_r)$

那么 $(P, (T_l, \Lambda, \Delta, t_c)) \preceq (\mathbb{P}, (T_h, \Sigma, \Delta))$ 成立.

引理5.4.1给出了任务模拟和全局模拟之间的关系。可以看出为了得到全局模拟关系，在执行过程中需要时刻保持下列性质：

- 底层任务池 T_l 和高层任务池 T_h 中的任务一一对应，并且抽象内核状态中的抽象任务表和任务池中的任务也是一一对应的。底层和高层正在执行的任务是相同的。
- 因为中断发生的操作语义定义在底层全局操作语义中，在对底层全局操作语义归纳证明时需要考虑中断发生的情况，所以要求中断处理程序和对应的中断规范都满足内核方法模拟关系。
- 内存 M 和抽象内核状态 Σ 可以被逻辑划分为 $n+1$ 块，n 是系统任务个数，M_k 和 Σ_k 表示任务 k 的局部资源，M_s 和 Σ_s 表示未被任务占有的共享资源。
- 共享资源满足 $\lfloor I \rfloor$，$\sigma \lhd M$ 表示将 σ 中的内存替换为 M。
- 底层和高层正在执行的任务之间满足任务模拟关系。
- 如果不在运行的任务获得该任务所需的共享资源（$\mathsf{SWINV}(I)$），那么满足任务模拟关系。

引理 5.4.2. 对于任意的 s、P、$\mathbb{P}$、σ、Σ 和 I，如果下述内容成立：

(1) $P = (A, (\eta_a, \eta_i, \theta))$，$\mathbb{P} = (A, (\varphi, \varepsilon, \chi))$

(2) $(\sigma, \Sigma, \_) \models \mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * \mathsf{empE}$

(3) $\Gamma; \chi; I \models \theta : \varepsilon$，$\Gamma; \chi; I \models \theta : \varepsilon$

那么，$P; \mathbb{P}; I; \mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * \mathsf{empE} \models ((s, (\circ, \bullet)), \sigma) \preceq ((s, (\circ, \bullet)), \Sigma)$ 成立。

引理5.4.2建立了内核方法模拟和任务模拟之间的关系，其表示如果每个系统 API 和内部函数都满足内核方法模拟关系，并且任务初始时，任务内核状态满足 $\mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * \mathsf{empE}$，那么可以得到对于任何相同的任务入口代码 s，底层和高层满足任务模拟关系。

定理5.4.1的证明　结合之前介绍的引理和定理，下面给出逻辑系统可靠性定理 5.4.1 的证明。

证明. 首先展开操作系统正确性定义，并且根据 TopRule规则，将定理 5.4.1 的证明为如下所示的命题的证明，横线上方是已知条件，下方是要证的结论：

$$\frac{O = (\eta_a, \eta_i, \theta) \quad \mathbb{O} = (\varphi, \varepsilon, \chi) \quad \chi; I \vdash \eta_i : \Gamma \quad \Gamma; \chi; I \vdash \eta_a : \varphi \quad \Gamma; \chi; I \vdash \theta : \varepsilon \quad \lfloor \psi \rfloor \Rightarrow I[0, N] * \mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * \mathsf{empE} \quad \mathsf{side}(I, \chi) \quad \mathsf{Match}(\psi, W, \mathbb{W})}{((A, O), W) \preccurlyeq ((A, \mathbb{O}), \mathbb{W})}$$

然后展开 W、$\mathbb{W}$ 和 Match 的定义，应用定理5.4.6将精化关系的证明转化为证明全局模拟关系，然后分别应用定理5.4.3、5.4.5 和5.4.4得到中断处理程序逻辑判断、内部函数逻辑判断和系统调用逻辑判断的语义，最终转化为证明如下命题：

$$\frac{\begin{array}{c} O = (\eta_a, \eta_i, \theta) \quad \mathbb{O} = (\varphi, \varepsilon, \chi) \quad \chi; I \vDash \eta_i : \Gamma \quad \Gamma; \chi; I \vDash \eta_a : \varphi \quad \Gamma; \chi; I \vDash \theta : \varepsilon \\ \lfloor \psi \rfloor \Rightarrow I[0, N] * \mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * \mathsf{empE} \quad \mathsf{side}(I, \chi) \\ t_c \in \mathrm{dom}(T) \quad \psi \wedge \Sigma \quad t_c = \Sigma(\mathtt{ctid}) \quad \mathrm{dom}(T) = \mathrm{dom}(\Sigma(\mathtt{tcbls})) \\ \forall t.t \in \mathrm{dom}(T) \Longrightarrow \exists s.T(t) = (s, (\circ, \bullet)) \end{array}}{((A, O), (T, \Delta, \Lambda, t)) \preceq ((A, \mathbb{O}), (T, \Delta, \Sigma))}$$

根据 $\lfloor \psi \rfloor \Rightarrow I[0, N] * \mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * \mathsf{empE}$ 的语义（$\lfloor \psi \rfloor$ 定义在5.8中），可知存在划分 $M_1, \ldots, M_n$ 和 M_s，其中 $M_1 = M_2 = \cdots = M_n = \emptyset$、$\Lambda.\mathfrak{m}.M = M_s$，使得 $\Lambda.\mathfrak{m}.M = M_1 \uplus \cdots \uplus M_n \uplus M_s$。$\Sigma$ 按照同样的方式划分，让 $\Sigma_s = \Sigma$，每个任务占有的抽象内核状态为空。

然后根据断言语义知道:

$$\forall t \in [1, \ldots, n].(\Lambda|_t, \Sigma_t, \_) \vDash \mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * \mathsf{empE}$$

并且根据 $\lfloor I \rfloor$ 和 $\lfloor \psi \rfloor$ 的定义，知道

$$(\Lambda|_{t_c} \lhd M_s, \Sigma_s, \_) \vDash \lfloor I \rfloor$$

然后根据前提 $\forall t.t \in \mathrm{dom}(T) \Longrightarrow \exists s.T(t) = (s, (\circ, \bullet))$ 和定理5.4.2，得到所有任务 t 都满足：

$$P; \mathbb{P}; I; \mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * \mathsf{empE} \vDash (T(t), \sigma_t) \preceq (T(t), \Sigma_t) \qquad \text{(P1)}$$

然后应用定理5.4.1，对于需要证明的定理5.4.1中的前提（1）~（7）都显然成立。

下面来看条件（8）的证明，首先根据 $\mathsf{SWINV}(I)$ 和底层任务状态不相交（$\sigma \perp \sigma'$）的定义知道下列三个性质，

$$\forall \sigma, \Sigma.\ (\sigma, \Sigma, \_) \vDash \mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * \mathsf{empE} \Longrightarrow (\sigma, \Sigma, \_) \vDash \mathsf{SWINV}(I)$$

$$\forall \sigma, \Sigma, \sigma', \Sigma'.(\sigma, \Sigma, \_) \vDash \mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * \mathsf{empE} \wedge \sigma \perp \sigma' \wedge \Sigma \perp \Sigma' \Longrightarrow (\sigma', \Sigma', \_) \vDash \mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * \mathsf{empE} * \mathsf{true}$$

$$\forall \sigma, \Sigma.\ (\sigma, \Sigma, \_) \vDash (\mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * \mathsf{empE} * \mathsf{true} \wedge \mathsf{SWINV}(I)) \Longrightarrow \Sigma = \emptyset \wedge \sigma.\mathfrak{m}.M = \emptyset$$

结合这三个性质得到对于条件（8）来说，$\sigma_i \uplus \sigma_r = \sigma_i$ 并且 $\Sigma_i \uplus \Sigma_r = \Sigma_i$。再结合已知条件 (P1)，可以得到条件（8）成立。

综上所述，可知定理5.4.1是正确的。

□

对于其他的定理和引理这里不再一一给出证明，具体可以看 Coq 代码 [3] 中的“/CertiOS/framework/theory/lemmasfortoptheo.v”文件。

5.5　本章小结

本章介绍了用于验证操作系统正确性的程序逻辑 CSL-R。首先介绍了关系型断言语言及其语义；然后介绍了推理规则并举了一个例子来说明如何使用推理规则来验证程序；接着给出了推理规则中各种逻辑判断的语义，最后证明了推理系统的可靠性。CSL-R 结合了并发分离逻辑占有权转移思想和 [34] 中对于非对称并发的处理方法，并扩展到精化关系的验证上。本文参考了 [8] 中的利用程序模拟技术验证精化关系的思想，构建了全局模拟关系、任务模拟关系和内核方法模拟关系来完成推理系统的可靠性证明。

第六章　局部不变式和分数权限（fractional permission）

在实际的代码证明过程中，我们发现全局不变式的表达力是有限的，对于很多情况难以处理。下面我们先看 μC/OS-II 中用于任务删除的函数 OSTaskDel，结合这段代码来解释为何全局不变式无法处理这种情况。

OSTaskDel 的具体代码在图6.1中，该函数传入一个参数 prio，表示将被删除的任务的优先级，μC/OS-II 中每个优先级只有一个任务，所以通过任务优先

INT8U OSTaskDel (INT8U prio):

```
1   OS_TCB       *ptcb;
2   OS_ENTER_CRITICAL();
3   ptcb = OSTCBPrioTbl[prio];
4   if (ptcb != NULL) {
5   ..... /* 如果任务是就绪态，则从就绪任务表中删除 */
6   ..... /* 如果任务等待某个事件，则从该事件等待任务表中删除 */
7       OSTCBPrioTbl[prio] = NULL;
8       if (ptcb->OSTCBPrev == NULL) {
9           ptcb->OSTCBNext->OSTCBPrev = NULL;
10          OSTCBList= ptcb->OSTCBNext;
        }
11      else {
12          ptcb->OSTCBPrev->OSTCBNext = ptcb->OSTCBNext;
13          ptcb->OSTCBNext->OSTCBPrev = ptcb->OSTCBPrev;
        }
14      ptcb->OSTCBNext = OSTCBFreeList;
15      OSTCBFreeList   = ptcb;
16      OS_EXIT_CRITICAL();
17      OS_Sched();
18      return (OS_NO_ERR);
    }
19  OS_EXIT_CRITICAL();
20  return(OS_TASK_DEL_ERR);
```

图 6.1　μC/OS-II 中任务删除函数

级可以唯一确定一个待删除的任务。该函数的返回值用于表示删除工作是否正确执行，如果正确删除则返回 OS_NO_ERR，如果删除过程出错则返回相应的出错信息。

下面来看下这个函数的主要内容。

- 第 1 行是局部变量 ptcb 的声明，该局部变量是用于指向被删除任务 TCB 的指针。
- 第 2 行执行 OS_ENTER_CRITICAL() 语句进入临界区（OS_ENTER_CRITICAL() 对应验证框架中的汇编原语 **encrt**）。
- 第 3 行根据待删除任务优先级 prio 和 OSTCBPrioTbl 数组，来获得指向待删除任务 TCB 的指针并赋值给局部变量 ptcb，这里 OSTCBPrioTbl 是一个数组，数组下标是任务优先级，数组元素内容是指向该优先级的任务 TCB 的指针。
- 第 4 行的 if 语句用来判断待删除任务是否存在，如果不存在那么直接退出临界区并报错（19，20 行）。
- 第 5-6 行用于检查待删除任务的状态。第 5 行先检查任务是否是就绪状态，如果是就绪状态则先将该任务从就绪任务表中删除（就绪任务表是报保存了所有就绪任务的优先级）。第 6 行检查任务是否在等待某个事件（例如：信号量），如果是则将该任务从该事件的等待任务表中删除（等待任务表保存了所有等待该事件的任务的优先级），这部分的代码这里不再展开。
- 第 7 行将该任务从 OSTCBPrioTbl 中删除。
- 8-13 行将被删除任务从全局任务表中删除，全局任务表是系统中所有任务组成的一个双向链表。图6.2 中给出了该双向链表的示意图，其中 OSTCBPrev 是指向双向链表中前驱节点的指针，OSTCBNext 是指向后继节点的指针，OSTCBCur 是指向双向链表中当前正在执行的任务 TCB 的指针，OSTCBList 是指向该双向链表头节点的指针。
- 第 14-15 行将被删除的任务 TCB 加入 OSTCBLFreeList 指向的单向链表，该单向链表保存所有空闲的 TCB，因此将任务 TCB 加入该单向链表表示任务 TCB 被释放，图6.2中给出了该单向链表的示意图。
- 第 16 行退出临界区（OS_EXIT_CRITICAL() 对应汇编原语 **excrt**）。
- 第 17 行执行一次任务调度。任务调度的具体代码在图4.17中，其代码主要包括四步：（1）执行 OS_ENTER_CRITICAL() 进入临界区；（2）找到当前优先级最高的就绪任务 t；（3）执行上下文切换 **switch** t，切换到任务 t 执行；（4）执行 OS_EXIT_CRITICAL() 退出临界区。

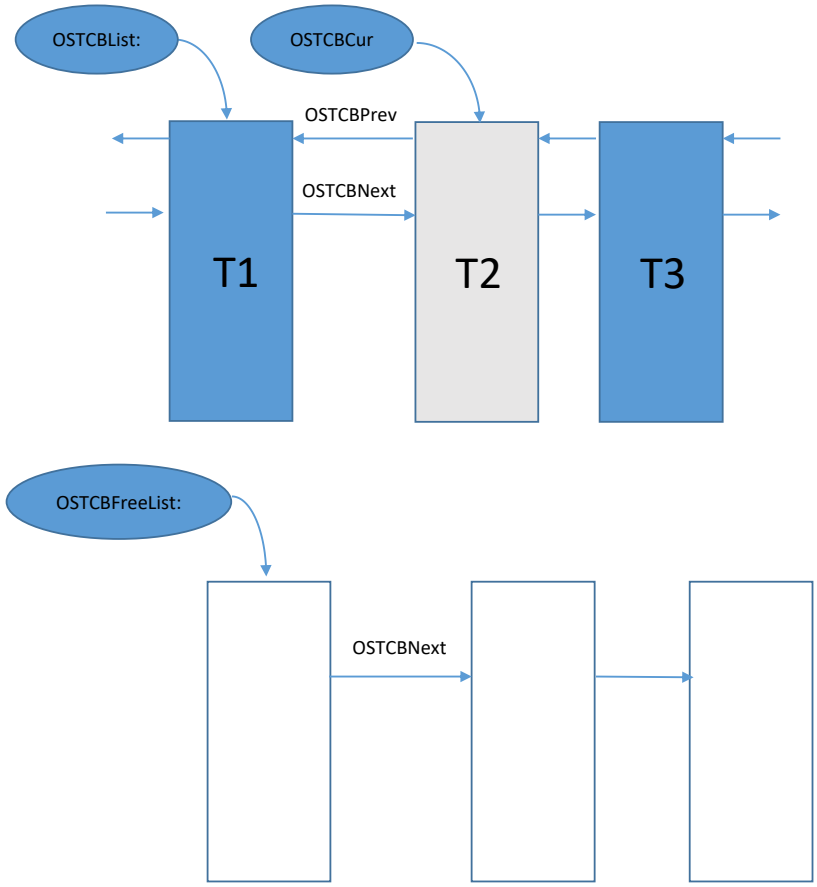

图 6.2　OSTCBList 和 OSTCBFreeList 结构图

$$\mathsf{AOSTCBList}(\bar{\mathtt{v}}) \stackrel{\text{def}}{=} \exists \mathtt{a_h},\ \mathtt{tail}.\ \mathtt{OSTCBList} \mapsto \mathtt{Vptr}(\mathtt{a_h}) * \mathsf{dl}(\mathtt{a_h}, \mathtt{tail}, \bar{\mathtt{v}}, \mathtt{TCB\_N}, \mathtt{TCB\_P}) * \mathtt{OSTCBCur} \mapsto \mathtt{t} * \langle \mathsf{R}(\bar{\mathtt{v}}) \wedge \mathtt{t} \in \bar{\mathtt{v}} \rangle$$

$$\mathsf{AOSTCBFreeList}(\bar{\mathtt{v}}) \stackrel{\text{def}}{=} \exists \mathtt{a_h}.\mathtt{OSTCBFreeList} \mapsto \mathtt{Vptr}(\mathtt{a_h}) * \mathsf{sl}(\mathtt{a_h}, \bar{\mathtt{v}}, \mathtt{TCB\_N})$$

$$\mathtt{I} \stackrel{\text{def}}{=} \exists \bar{\mathtt{v}}, \bar{\mathtt{v}}_\mathtt{f}.\ \mathsf{AOSTCBList}(\bar{\mathtt{v}}) * \mathsf{AOSTCBFreeList}(\bar{\mathtt{v}}_\mathtt{f}) * \cdots$$

图 6.3　不变式定义

- 最后第 18 行返回 OS_NO_ERR，表示正确删除。

为了理解为什么不变式表达力不够，需要先看下引入分数权限（fractional permission）和局部不变式之前的不变式定义。不变式定义在图6.3中，这里只给出了不变式中和 OSTaskDel 函数代码相关部分的定义，主要包括描述全局任务表 OSTCBList 的断言 AOSTCBList 和 AOSTCBFreeList。

断言 AOSTCBList 描述了图6.2中上半部分的内容，$\mathtt{OSTCBList} \mapsto \mathtt{Vptr}(\mathtt{a_h})$ 表示 OSTCBList 是一个全局指针变量，$\mathtt{Vptr}(\mathtt{a_h})$ 表示 OSTCBList 指向地址

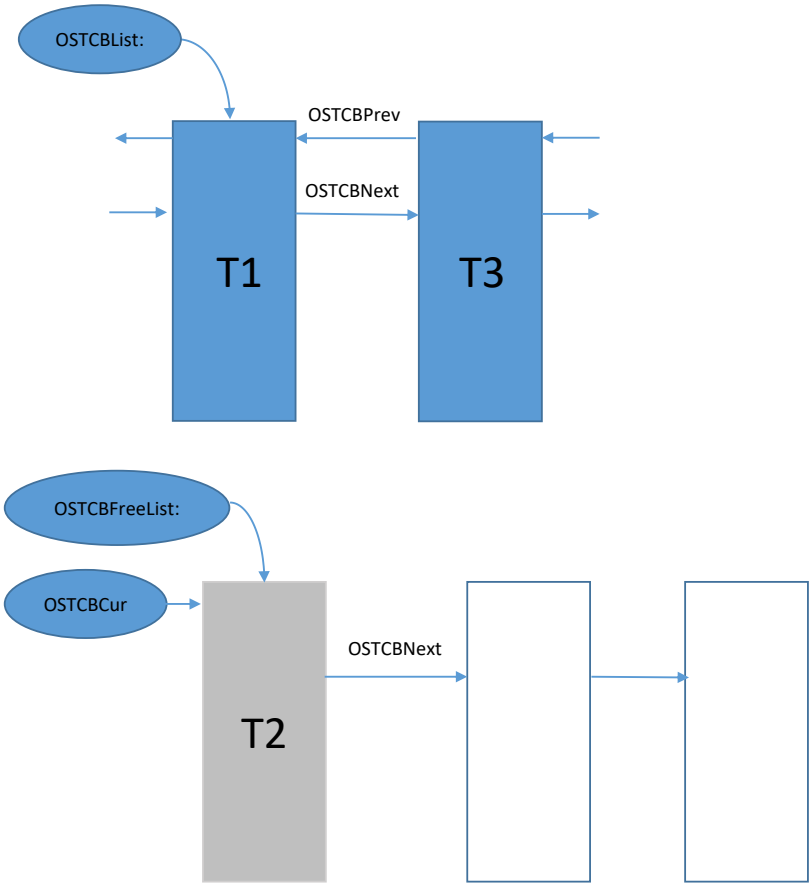

图 6.4　删除当前任务后 OSTCBList 和 OSTCBFreeList 结构图

a_h；$\mathsf{dl}(a_h, tail, \bar{v}, TCB\_N, TCB\_P)$ 描述了一个双向链表，该双向链表头节点的地址为 a_h，尾节点地址为 $tail$，$\bar{v}$ 中保存了双向链表中每个节点的内容，TCB_N（TCB_P）用于表示链表节点中哪个域用于指向后继（前驱）节点；$OSTCBCur \mapsto t$ 表示 OSTCBCur 是一个全局指针变量，其指向任务 t；$R(\bar{v})$ 用于描述全局任务表中的每个任务 TCB 需要满足的性质；$t \in \bar{v}$ 表示 t 是全局任务表中的一个任务。dl 的定义在5.1节，$R(\bar{v})$ 和 $t \in \bar{v}$ 的定义这里不再展开。

断言 AOSTCBFreeList 描述了图6.2中下半部分的内容，$OSTCBFreeList \mapsto \mathtt{Vptr}(a_h)$ 表示 OSTCBList 是一个全局指针变量，$\mathtt{Vptr}(a_h)$ 表示 OSTCBFreeList 指向地址 a_h；$\mathsf{sl}(a_h, \bar{v}, TCB\_N)$ 描述了一个单向链表，该单向链表头节点地址为 a_h，$\bar{v}$ 中保存了单向链表中每个节点的内容，TCB_N 用于表示链表节点中哪个域用于指向后继节点。sl 的定义在5.1 节。

上面介绍了不变式，下面接着来看在验证任务删除函数时存在的问题。这里 OSTaskDel 函数是可以用来删除当前正在运行的任务自己的（如果参数 prio 是当前执行的任务自己的优先级）。如果 OSTaskDel 用来删除自己，那么当执行到图6.1中代码 16 行前，OSTCBList 和 OSTCBFreeList 的结构如图6.4所示。第

16 行代码是退出临界区，此时要求全局资源是良构的，需要满足不变式 I 要求，而不变式 I 要求 OSTCBList 的结构应该和图6.2一致，其中 OSTCBCur 指向的任务控制块应该在全局任务表中。显然这里图6.4所示的结构不满足不变式要求。所以 OSTaskDel 函数没法通过验证。

无法通过验证可能是如下原因：（1）代码实现本身是错误的，不应该在全局数据结构如图6.4所示时退出临界区；（2）不变式定义太强，导致在退出临界区时无法满足，应当弱化不变式定义。下面来看代码实现本身是否存在错误，以及不变式定义是否存在问题。

该段代码实现没有 bug。当任务删除自己，退出临界区（执行完图6.1第 16 行代码）后，这里存在两种情况：（1）中断到来，但 μC/OS-II 中断处理程序不关心 OSTCBCur 指针的指向，所以在中断处理程序中 OSTCBList 可以是如图6.4所示的结构，同时由于 μC/OS-II 中最外层中断处理程序退出时会进行任务调度，调度程序会将 OSTCBCur 指针指向被调度到的任务 TCB，调度程序选择的任务一定是在 OSTCBList 中（图6.4中的 T1 或 T3），所以最终中断处理程序退出后 OSTCBList 会变回图6.2所示的结构，并满足不变式 I 的定义；（2）没有中断发生，那么继续执行后面 17 行的任务调度程序，任务调度发生后满足不变式 I。这段代码没有问题的原因是，图6.4中任务 T2 删除自己后，在其他任何任务开始执行前，不变式 I 可以不满足，但是一旦其他任务开始被调度执行，能够保证不变式 I 继续成立。

从上述内容可以看出这里不变式 I 的定义的确太强了，不变式 I 的定义应该考虑当前代码是否处于任务 T2 已经将自己从 OSTCBList 中删除并且尚未调度到其他任务执行的特殊时间段（将这一特殊时间段记为 $\mathrm{t}_{\mathrm{delself}}$），正确的不变式定义 I_{c} 应该类似如下结构：

$$\mathrm{I}_{\mathrm{c}} \stackrel{\mathrm{def}}{=} (\mathrm{C} \in \mathrm{t}_{\mathrm{delself}} \wedge \mathrm{I}') \vee ((\neg\, \mathrm{C} \in \mathrm{t}_{\mathrm{delself}}) \wedge \mathrm{I})$$

这里 $\mathrm{C} \in \mathrm{t}_{\mathrm{delself}}$ 只是示意，并没有形式化的定义，表示当前代码执行正处于特殊时间段 $\mathrm{t}_{\mathrm{delself}}$ 中，I' 是刻画图6.4所示的特殊情况下的不变式，这里也不再展开介绍。那么接下来的问题就是如何去定义 $\mathrm{C} \in \mathrm{t}_{\mathrm{delself}}$。

由于本文的断言语言无法刻画当前程序代码，所以无法直接通过已有的断言语言刻画出 $\mathrm{C} \in \mathrm{t}_{\mathrm{delself}}$。需要通过在任务 TCB 中引入一个特殊的辅助的 flag 域，同时在退出临界区（图6.1中 16 行代码）前加入如下代码：

```
if (ptcb == OSTCBCur)
    ptcb->flag = 0;
```

来刻画出 $\mathrm{C} \in \mathrm{t}_{\mathrm{delself}}$。

并在任务创建初始化时将 flag 域赋值为 1。这样就可以通过每个任务的 flag 域的值来判断当前该任务是否处于特殊时间段 $\mathrm{t}_{\mathrm{delself}}$，如果 flag 域当前值为 0，表示 $\mathrm{C} \in \mathrm{t}_{\mathrm{delself}}$，如果 flag 域为 1，则表示 $\neg\, \mathrm{C} \in \mathrm{t}_{\mathrm{delself}}$。

这样新的不变式定义 I_c 如下所示：

$$\begin{aligned} I_c \stackrel{\text{def}}{=} \ & ((get\_addr(OSTCBCur, flag) \xmapsto{\texttt{Tint32}} \texttt{Vint}(0) * I') \vee \\ & ((get\_addr(OSTCBCur, flag) \xmapsto{\texttt{Tint32}} \texttt{Vint}(1)) * I) \end{aligned}$$

这里 $get\_addr(OSTCBCur, flag)$ 表示通过全局变量 OSTCBCur 获取当前任务 TCB 中 flag 域的地址，不再给出具体定义。在这样定义之后全局不变式的确通过 flag 域区分了图6.2和图6.4两种不同的情况，但是在实际验证的过程中仍然存在问题，主要的问题是在任务执行 API 并进入临界区后（例如：图6.1中的第 2 行），拿到的全局不变式资源是 I_c（flag 域可能为 1，也可能为 0），但此时任务明确知道自己在全局任务表中，flag 域的值是 1（因为任务不可能在删除自己后还去调用一个新的 API）。需要刻画出每次任务在开始执行 API 时，此时该任务的 flag 域为 1，否则任务无法拿到自己 TCB 满足的不变式 I 中 $R(\bar{v})$ 中刻画的那些性质（因为无法确定此时取 I_c 中的析取式右分支 $(get\_addr(OSTCBCur, flag) \xmapsto{\texttt{Tint32}} \texttt{Vint}(1)) * I$）。为了刻画出任务每次开始执行 API 时 flag 域都为 1 的这一性质（此时尚未进入临界区拿到全局不变式的占有权），需要在任务的局部资源中描述 flag 域，但同时 flag 域也在全局不变式中被描述。

6.1　解决问题的思路

为了解决一个资源同时需要在全局和局部被描述的问题，本文对验证框架进一步作了扩展和调整，引入了分数权限（fractional permission）[54]。这里的基本思想是，通过分数权限机制将这种需要同时在全局和局部被描述的资源划分为两半，每个任务自己占有其中一半（将任务占有的这一半称为部分共享资源），另外一半放入全局不变式中。当一个任务占有一半资源时可以去读该资源中的内容，但是不能写，只有某个任务占有了全部的资源时才允许写。这样就解决了资源同时在全局和局部被描述的问题，并且保证了该资源只会被任务自己修改，从而可以准确的刻画当前任务代码的执行位置。

仅仅引入分数权限机制仍然是不够的，因为中断处理程序中可能需要任务的部分共享资源，需要保证部分共享资源在中断可能到来的点都是良构的（well-formed）。下面来看为何中断处理程序中需要任务的部分共享资源以及如何解决这个问题。

在引入 flag 域和分数权限之后，执行完图6.1中 16 行代码退出临界区后，OSTCBList 和 OSTCBFreeList 结构如图6.5所示。此时中断可能会发生并被响应，在中断处理程序中可能会执行上下文切换 **switch** x，(x 表示将要被切换到的任务，对应图6.5中的 T1 或 T3，下面假设其为 T3)，**switch** x 原语会将 OSTCBCur 指向 x。执行完上下文切换后 OSTCBList 和 OSTCBFreeList 结构如图6.6所示。

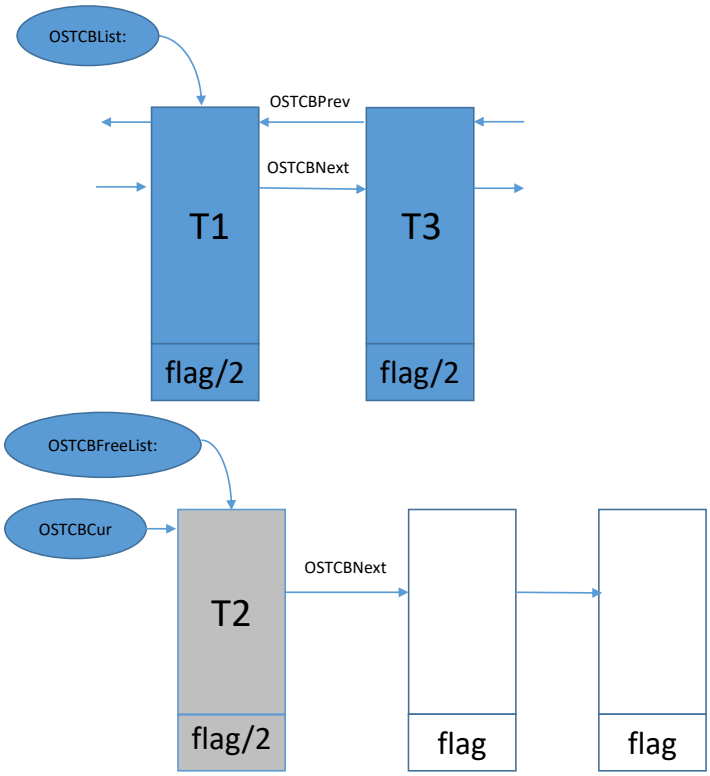

图 6.5　引入 flag 域和分数权限后 OSTCBList 和 OSTCBFreeList 结构图

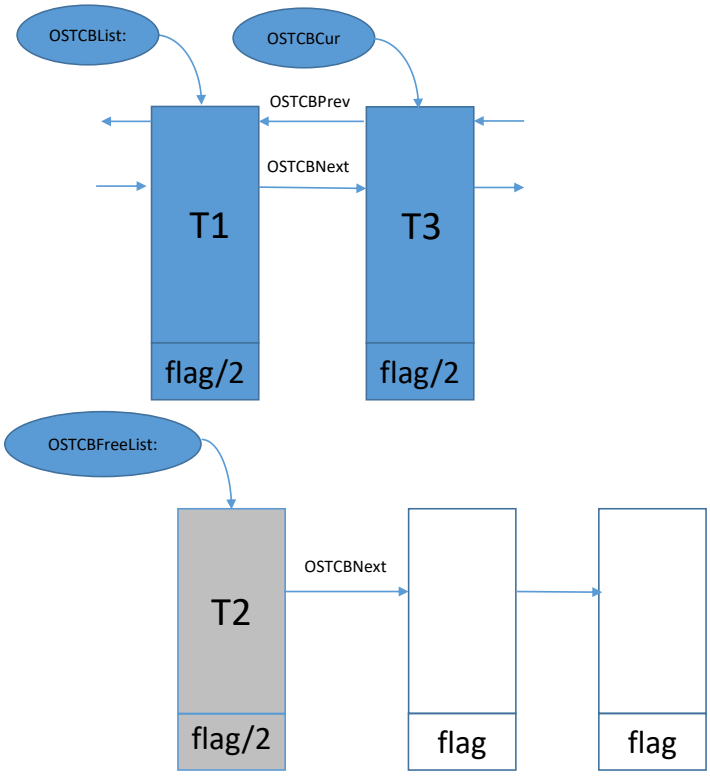

图 6.6　中断处理程序中执行上下文切换后 OSTCBList 和 OSTCBFreeList 结构图

验证框架要求执行任务切换后需要保证全局资源满足全局不变式 I（因为上下文切换后 $\neg\, C \in t_{delself}$，所以应该取 I_c 的右分支），但图6.6 所示结构不满足全局不变式 I，因为 I 要求 OSTCBFreelist 中每个空闲任务控制块的 flag 域是完整的，而图6.6 中 T2 的 flag 域只有一半。T2 的 flag 域的另外一半在 T2 的局部资源中（也即前述的部分共享资源），因此这里需要将 T2 的部分共享资源的占有权转移到全局资源中。

所以如前所述，中断处理程序中可能需要任务的部分共享资源，而中断的发生是不确定的，所以需要保证中断可能发生的时刻部分共享资源都是良型的（well-formed），例如在上述问题中，需要保证 T2 部分共享资源中的一半 flag 可以和 OSTCBFreeList 中 T2 的另一半 flag 在一起组成一个完整的 flag 域。因此在分数权限的基础上，我们进一步引入了局部不变式，局部不变式用于刻画任务的部分共享资源的良型性。在中断发生时，部分共享资源会发生占有权转移，从当前任务的局部资源中转移至中断处理程序的资源中。

6.2　对验证框架的扩展和调整

对验证框架的扩展主要包含三个方面：

- 首先，在内存模型中引入了分数权限，这里的分数权限是经过简化的，权限只有 $\frac{1}{2}$ 和 1 两种，分别对应读权限和写权限。内存的类型不再是从内存地址到内存值的映射，而是如下所示的内存地址到内存值和内存权限组成的二元组的映射。

  $$(\textit{Permission})\quad \mathrm{pm} ::= \tfrac{1}{2} \mid 1$$
  $$(\textit{Memory})\quad M \in \textit{Addr} \rightharpoonup (\textit{MemVal} \times \textit{Permission})$$

  当某个内存单元只有 $\frac{1}{2}$ 权限时，表示该内存单元只允许读，不允许写。当某个内存单元有完整的 1 权限时，表示该内存单元同时允许读和写。各自拥有 $\frac{1}{2}$ 权限，且内存值 *MemVal* 相同的两块内存，合在一起可以构成一块完整的拥有 1 权限的内存。

- 然后需要扩展断言语言，引入用于描述只读内存单元的断言，例如：$(\sigma, \Sigma, \mathbf{s}) \models a \overset{\tau}{\underset{\circ}{\mapsto}} v$ 和 $(\sigma, \Sigma, \mathbf{s}) \models x \overset{\tau}{\underset{\circ}{\mapsto}} v$，断言语义定义如下：

  $$(\sigma, \Sigma, \mathbf{s}) \models a \overset{\tau}{\underset{\circ}{\mapsto}} v \quad \text{iff} \quad \sigma.m.M = \{a \overset{\tau}{\rightsquigarrow}_r v\} \wedge \Sigma = \emptyset$$
  $$(\sigma, \Sigma, \mathbf{s}) \models x \overset{\tau}{\underset{\circ}{\mapsto}} v \quad \text{iff} \quad \exists b.(\sigma.m.G)(x) = (b, \tau) \wedge (\sigma, \Sigma, \mathbf{s}) \models (b, 0) \overset{\tau}{\underset{\circ}{\mapsto}} v$$
  $$M = \{(b, o) \overset{\tau}{\rightsquigarrow}_\circ v\} \overset{\text{def}}{=} \exists v_1, \cdots, v_{|\tau|}.\, \mathsf{encode}(v, \tau) = \{v_1, \cdots, v_{|\tau|}\} \wedge M = \{(b, o) \rightsquigarrow (v_1, \tfrac{1}{2}), \cdots, (b, o + |\tau| - 1) \rightsquigarrow (v_{|\tau|}, \tfrac{1}{2})\}$$

  原有的描述内存的断言描述的都是完整权限的内存单元。例如：

$$(\sigma,\Sigma,\mathbf{s})\models a\overset{\tau}{\mapsto}v \quad \text{iff} \quad \sigma.m.M=\{a\overset{\tau}{\rightsquigarrow}v\}\wedge\Sigma=\emptyset$$

$$M=\{(b,o)\overset{\tau}{\rightsquigarrow}v\} \overset{\text{def}}{=} \exists v_1,\cdots,v_{|\tau|}.\ \mathsf{encode}(v,\tau)=\{v_1,\cdots,v_{|\tau|}\}\wedge M=\{(b,o)\rightsquigarrow(v_1,1),\cdots,(b,o+|\tau|-1)\rightsquigarrow(v_{|\tau|},1)\}$$

所以有，$a\overset{\tau}{\mapsto}v \Leftrightarrow a\underset{\circ}{\overset{\tau}{\mapsto}}v * a\underset{\circ}{\overset{\tau}{\mapsto}}v$，并且 $(a\underset{\circ}{\overset{\tau}{\mapsto}}v_1 * a\underset{\circ}{\overset{\tau}{\mapsto}}v_2\wedge v_1\neq v_2)\Rightarrow \mathsf{false}$。

- 引入描述部分共享资源的局部不变式 LI。引入局部不变式 LI 的主要原因是中断处理程序中可能会访问部分共享资源，又由于中断随时可能发生，所以需要保证部分共享资源时刻是良型的（well-formed），所以引入局部不变式 LI 来刻画该良型性。在中断处理程序被响应时，部分共享资源的占有权会转移至中断处理程序中。

有了上述扩展后需要对推理规则和逻辑判断（Judgement）作一些调整，相应的逻辑判断的语义（Judgement Semantics）也要作调整。下面将分别介绍这些调整。

推理规则的调整　首先各个逻辑判断（Judgement）调整为如下格式：

$$\chi;I;\mathsf{LI}\vdash\eta_i:\Gamma \qquad \Gamma;\chi;I;\mathsf{LI};p_{init}\vdash\eta_a:\varphi \qquad \Gamma;\chi;I;\mathsf{LI}\vdash\theta:\varepsilon$$

$$\Gamma;\chi;I;\mathsf{LI};\rho;p_i\vdash\left\{\ p\ \right\}s\left\{\ q\ \right\}$$

主要调整是增加了局部不变式 LI，并且在用于验证系统 API 的逻辑判断 $\Gamma;\chi;I;\mathsf{LI};p_{init}\vdash\eta_a:\varphi$ 中加入 p_{init} 来描述系统 API 开始执行时局部资源满足的规范。

推理规则的调整主要包含四个部分：

- 顶层规则（TopRule）的调整。为了验证内核，现在不仅需要提供内部函数 η_i 中函数的规范 Γ 和内核状态的全局不变式集合 I，还需要提供局部不变式 LI 以及系统 API 开始执行时对局部资源的要求 p_{init}，同时要求系统 API 开始执行时部分共享资源必须满足局部不变式（$p_{init}\Rightarrow\mathsf{LI}$），调整后的 TopRule如下所示：

$$\frac{\begin{array}{c}O=(\eta_a,\eta_i,\theta)\quad \mathbb{O}=(\varphi,\varepsilon,\chi)\\ \chi;I;\mathsf{LI}\vdash\eta_i:\Gamma \qquad \Gamma;\chi;I;\mathsf{LI};p_{init}\vdash\eta_a:\varphi \qquad \Gamma;\chi;I;\mathsf{LI}\vdash\theta:\varepsilon\\ \lfloor\psi\rfloor\Rightarrow I[0,N]*\mathsf{OS}[\bar{0},1,\mathtt{nil},\mathtt{nil}]*\mathsf{empE}\quad \mathsf{side}(I,\chi)\quad p_{init}\Rightarrow\mathsf{LI}\end{array}}{\vdash_\psi O:\mathbb{O}}\ (\textsc{TopRule})$$

- 对验证系统 API 的规则 WfAPI作出相应调整，主要是用于构造系统 API 前后断言的定义 BuildAPIP 和 BuildAPIR 中加入了描述部分共享资源的

断言 p_{init}，调整后的 WFAPI规则和 BuildAPIP、BuildAPIR 的定义如下：

$$\frac{\begin{array}{c}dom(\eta) = dom(\varphi) \quad \eta(f) = (\_,\_,\_,s) \quad \varphi(f) = \omega \\ p = \mathsf{BuildAPIP}(\eta, f, \omega, \bar{v}, p_{init}) \quad \rho = \mathsf{BuildAPIR}(\eta, f, p_{init}) \\ \Gamma; \chi; I; \mathsf{LI}; \rho; \mathsf{false} \vdash \left\{\ p\ \right\} s \left\{\ \mathsf{false}\ \right\}\end{array}}{\Gamma; \chi; I; \mathsf{LI}; p_{init} \vdash \eta : \varphi}\ (\textsc{WFAPI})$$

$$\mathsf{BuildAPIP}(\eta_a, f, \omega, \bar{v}, p_{init}) \stackrel{\text{def}}{=} \mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * \mathsf{BuildP}(\mathsf{getD}(\eta_a, f), \bar{v}) * [|\omega(\bar{v})|] * p_{init}$$

$$\mathsf{BuildAPIR}(\eta_a, f, p_{init}) \stackrel{\text{def}}{=} \lambda\, \hat{v}.\ \mathsf{OS}[\bar{0}, 1, \mathtt{nil}, \mathtt{nil}] * \mathsf{BuildR}(\mathsf{getD}(\eta_a, f)) * [|\mathbf{end}\ \hat{v}|] * p_{init}$$

- 对验证中断处理程序的规则 ITRP作出相应调整，主要是用于构造中断处理程序前后断言的定义 BuildItrpP 和 BuildItrpR 中加入了局部不变式 LI，表示中断发生时部分共享资源是良型的，调整后的 ITRP规则和 BuildItrpP、BuildItrpR 的定义如下：

$$\frac{\begin{array}{c}dom(\theta) = dom(\varepsilon) \\ p = \mathsf{BldItrpP}(k, \varepsilon, \mathit{isr}, \mathit{is}, I; \mathsf{LI}) \quad p_i = \mathsf{BldItrpR}(k, \mathit{isr}, \mathit{is}, I; \mathsf{LI}) \\ \Gamma; \chi; I; \mathsf{LI}; \lambda\, \hat{v}.\ \mathsf{false}; p_i \vdash \left\{\ p\ \right\} \theta(k) \left\{\ \mathsf{false}\ \right\}\ \text{for all } k \in \{0, \ldots, N-1\}\end{array}}{\Gamma; \chi; I; \mathsf{LI} \vdash \theta : \varepsilon}\ (\textsc{Itrp})$$

$$\mathsf{BldItrpP}(k, \varepsilon, \mathit{isr}, \mathit{is}, I; \mathsf{LI}) \stackrel{\text{def}}{=} \mathsf{OS}[\mathit{isr}\{k \rightsquigarrow 1\}, 0, k :: \mathit{is}, \mathtt{nil}] * I[0, k] * [|\varepsilon(k)|] * \mathsf{empE} * \mathsf{LI}$$

$$\mathsf{BldItrpR}(k, \mathit{isr}, \mathit{is}, I; \mathsf{LI}) \stackrel{\text{def}}{=} \exists\, \mathit{ie}.\ \mathsf{OS}[\mathit{isr}\{k \rightsquigarrow 0\}, \mathit{ie}, k :: \mathit{is}, \mathtt{nil}] * ((\mathit{ie} = 1 \wedge \mathtt{emp}) \vee (\mathit{ie} = 0 \wedge I[0, k])) * [|\mathbf{end}\ \bot\ |] * \mathsf{LI}$$

- 对于可能会破坏局部不变式的程序语句，在推理规则中要求执行这些程序语句之后局部不变式继续成立，例如对于赋值语句：新的推理规则如下：

$$\frac{p * a \xmapsto{\tau_1} v_1 \Rightarrow \&e =_{\tau_1} a \wedge e_2 =_{\tau_2} v_2 \quad \tau_1 \propto \tau_2 \quad p * a \xmapsto{\tau_1} v_2 * [|\mathbf{s}|] \Rightarrow \mathsf{LI}}{\Gamma; \chi; I; \mathsf{LI}; \rho; p_i \vdash \left\{\ p * a \xmapsto{\tau_1} v_1 * [|\mathbf{s}|]\ \right\} e_1 = e_2 \left\{\ p * a \xmapsto{\tau_1} v_2 * [|\mathbf{s}|]\ \right\}}$$

逻辑判断语义（Judgement Semantics）的调整　逻辑判断的语义也需要作相应的调整，定义6.2.1和定义6.2.2给出了调整后的语句逻辑判断（$\Gamma; \chi; I; \mathsf{LI}; \rho; p_i \vdash \left\{\ p\ \right\} s \left\{\ q\ \right\}$）的语义以及内核方法模拟（分别对应第五章的定义5.4.1 和5.4.2）。定义6.2.3、6.2.4和6.2.5分别给出了逻辑判断 $\Gamma; \chi; I; \mathsf{LI} \vdash \theta : \varepsilon$、$\Gamma; \chi; I; \mathsf{LI}; p_{init} \vdash \eta_a : \varphi$ 和 $\chi; I; \mathsf{LI} \vdash \eta_i : \Gamma$ 的语义（对应第五章的定义5.4.3、5.4.4和5.4.5），定义6.2.3、6.2.4和6.2.5 基本没有变化，这里不再一一介绍。

定义6.2.2中只给出了**Normal Steps**、**Function call**以及**Skip**的情况。**Switch**情况基本没有改动，**Ret**、**IRet**的情况和**Skip**类似，这里都没有给出具体定义。

定义6.2.1的调整在于，要求初始状态不仅满足前断言 p，同时还要满足初始状态满足局部不变式。因为局部不变式只是刻画一部分局部状态并不会刻画完整的局部状态（例如：局部不变式不会去刻画内核函数的局部变量之类的信息），所以这里是要求初始状态满足 $\mathsf{LI} * \mathsf{true}$。

对于内核方法模拟，主要调整包括三个部分：

- 要求底层内核代码正常执行非函数调用的一步（**Normal Steps**），高层代码执行相应的 0 步或多步后，要求局部不变式继续成立。（对应定义6.2.2中 **Normal Steps**情况下加入的 $(\sigma', \Sigma', \_) \models \mathsf{LI} * \mathsf{true}$）。

- 如果执行的是一步函数调用，那么要求转移给被调用函数的资源满足局部不变式（对应定义6.2.2中 **Function call**情况下加入的 $(\sigma_1, \Sigma_1, \mathbf{s}') \models \mathsf{LI} * \mathsf{true}$），同时要求被调用函数返回后返回的资源满足局部不变式（对应定义6.2.2中 **Function call**情况下加入的 $(\sigma_1', \Sigma_1', \mathbf{s}'') \models \mathsf{LI} * \mathsf{true}$）。

- 如果是底层代码执行到 **skip** 的情况，那么需要保证结束状态满足局部不变式（对应定义6.2.2中 **Skip**情况下加入的 $(\sigma, \Sigma', \mathbf{s}') \models (\mathsf{LI} * \mathsf{true})$）。

定义 6.2.1 (逻辑判断语义). $\Gamma; \chi; \mathrm{I}; \mathsf{LI}; \rho; \mathrm{p_i} \models \{\mathrm{p}\}\mathrm{s}\{\mathrm{q}\}$ 成立，当且仅当，对于任意的 σ, Σ 和 $\mathbf{s}$，如果 $(\sigma, \Sigma, \mathbf{s}) \models \mathrm{p}$ 并且 $(\sigma, \Sigma, \mathbf{s}) \models \mathsf{LI} * \mathsf{true}$ 成立，那么 $\Gamma; \chi; \mathrm{I}; \mathsf{LI}; \rho; \mathrm{p_i}; \mathrm{q} \models ((\mathrm{s}, (\circ, \bullet)), \sigma) \preceq (\mathbf{s}, \Sigma)$。

定义 6.2.2 (内核方法模拟). 内核方法模拟是满足下述性质的最大关系。若 $\Gamma; \chi; \mathrm{I}; \rho; \mathrm{p_i}; \mathrm{q}; \mathsf{LI} \models (\mathrm{C}, \sigma) \preceq (\mathbf{s}, \Sigma)$ 成立，则下述全部成立：

- ***Normal Steps:*** 对于任意的 P、C'、σ_s、Σ_s、σ_1 和 σ_1'，如果 $\mathrm{C} \neq (\textit{fexec}(\_, \_), \_)$，$(\sigma_s, \Sigma_s, \_) \models \lfloor \mathrm{I} \rfloor$，$\sigma_1 = \sigma \uplus \sigma_s$，$\Sigma \perp \Sigma_s$ 和 $\mathrm{P} \vdash (\mathrm{C}, \sigma_1) \bullet\!\!-\!\!\mathrm{L}\!\!\triangleright (\mathrm{C}', \sigma_1')$ 成立，那么存在 Σ_s'、$\mathbf{s}'$、Σ'、σ' 和 σ_s'，使得下述内容成立：

  - $\sigma_1' = \sigma' \uplus \sigma_s'$，$(\sigma_s', \Sigma_s', \_) \models \lfloor \mathrm{I} \rfloor$，$(\sigma', \Sigma', \_) \models \mathsf{LI} * \mathsf{true}$,
  - $(\mathbf{s}, \Sigma \uplus \Sigma_s) \bullet\!\!-\!\!\mathrm{H}\!\!\triangleright^{*} (\mathbf{s}', \Sigma' \uplus \Sigma_s')$,
  - $\Gamma; \chi; \mathrm{I}; \mathsf{LI}; \rho; \mathrm{p_i}; \mathrm{q} \models (\mathrm{C}', \sigma') \preceq (\mathbf{s}', \Sigma')$.

- ***Function Call:*** 对于任意的 σ_s、Σ_s、κ_s、f 和 $\bar{\mathrm{v}}$，如果 $\mathrm{C} = (\textit{fexec}(\mathrm{f}, \bar{\mathrm{v}}), (\circ, \kappa_s))$、$\sigma \perp \sigma_s$、$(\sigma_s, \Sigma_s, \_) \models \lfloor \mathrm{I} \rfloor$ 和 $\Sigma \perp \Sigma_s$ 成立，那么存在 σ_1、σ_f、Σ_1、Σ_f、Σ'、Σ_s'、$\mathbf{s}'$、$\mathcal{L}$、fp 和 fq，使得下述内容成立:

  - $\Gamma(\mathrm{f}) = (\mathrm{fp}, \mathrm{fq})$
  - $(\mathbf{s}, \Sigma \uplus \Sigma_s) \bullet\!\!-\!\!\mathrm{H}\!\!\triangleright^{*} (\mathbf{s}', \Sigma' \uplus \Sigma_s')$，$(\sigma_s, \Sigma_s', \mathbf{s}') \models \lfloor \mathrm{I} \rfloor$
  - $\sigma = \sigma_1 \uplus \sigma_\mathrm{f}$，$\Sigma' = \Sigma_1 \uplus \Sigma_\mathrm{f}$，
    $(\sigma_1, \Sigma_1, \mathbf{s}') \models \mathrm{fp}\ (\mathsf{rev}(\bar{\mathrm{v}}))\ \mathcal{L}$，$\Sigma' = \Sigma_1 \uplus \Sigma_\mathrm{f}$，$(\sigma_1, \Sigma_1, \mathbf{s}') \models \mathsf{LI} * \mathsf{true}$

  - 对于任意 $\hat{v}$、σ'、σ'_1、Σ''、Σ'_1 和 $\mathbf{s}''$，如果 $\sigma.m.G = \sigma'.m.G$、$\sigma.m.E = \sigma'.m.E$、$(\sigma'_1, \Sigma'_1, \mathbf{s}'') \models fq\ (rev(\bar{v}))\ \mathcal{L}\ \hat{v}$、$\sigma' = \sigma'_1 \uplus \sigma_f$ 和 $\Sigma'' = \Sigma'_1 \uplus \Sigma_f$ 成立，那么 $(\sigma'_1, \Sigma'_1, \mathbf{s}'') \models \mathsf{LI} * \mathsf{true}$ 并且 $\Gamma; \chi; I; \mathsf{LI}; \rho; p_i; q \models ((\mathbf{skip}, (\circ, \kappa_s)), \sigma') \preceq (\mathbf{s}'', \Sigma'')$ 成立

- ***Skip:*** 对于任意 σ_s 和 Σ_s，如果 $C = (\mathbf{skip}, (\circ, \bullet))$、$\sigma \perp \sigma_s$、$(\sigma_s, \Sigma_s, \_) \models \lfloor I \rfloor$ 和 $\Sigma \perp \Sigma_s$ 成立，那么存在 Σ'、Σ'_s 和 $\mathbf{s}'$，使得 $(\mathbf{s}, \Sigma \uplus \Sigma_s) \bullet\!\!-\!\!\text{H}\!\!-\!\!\triangleright^{*} (\mathbf{s}', \Sigma' \uplus \Sigma'_s)$、$(\sigma_s, \Sigma'_s, \_) \models \lfloor I \rfloor$、$(\sigma, \Sigma', \mathbf{s}') \models (\mathsf{LI} * \mathsf{true})$ 和 $(\sigma, \Sigma', \mathbf{s}') \models q$ 成立

- ……

定义 6.2.3 (中断逻辑判断语义). $\Gamma; \chi; I; \mathsf{LI} \models \theta : \varepsilon$ 成立，当且仅当，$dom(\theta) = dom(\varepsilon)$，并且对于任意 k，*isr*，*is*，p，s 和 p_i，如果 $\varepsilon(k) = \mathbf{s}$，$\theta(k) = s$，$p = \mathsf{BldItrpP}(k, \mathbf{s}, isr, is, I)$，$p_i = \mathsf{BldItrpR}(k, isr, is, I)$，那么 $\Gamma; \chi; I; \mathsf{LI}; \mathsf{false}; p_i \models \{p\}s\{\mathsf{false}\}$

定义 6.2.4 (系统调用逻辑判断语义). $\Gamma; \chi; I; \mathsf{LI}; p_{init} \models \eta_a : \varphi$ 成立，当且仅当，$dom(\eta_a) = dom(\varphi)$，并且对于任意的 f、$\bar{v}$、ω、p 和 ρ，如果 $\varphi(f) = \omega$，$p = \mathsf{BuildAPIP}(\eta_a, f, \omega, \bar{v}; p_{init})$，$\rho = \mathsf{BuildAPIR}(\eta_a, f; p_{init})$，$\eta_a(f) = (\_, \_, \_, s)$，那么 $\Gamma; \chi; I; \mathsf{LI}; \rho; \mathsf{false} \models \{p\}s\{\mathsf{false}\}$。

定义 6.2.5 (内部函数逻辑判断语义). $\chi; I; \mathsf{LI} \models \eta_i : \Gamma$ 成立，当且仅当，$dom(\Gamma) = dom(\eta_i)$，并且对于任意的 f、s、fp、fq、$\bar{v}$、$\mathcal{L}$、p 和 ρ，如果 $\eta_i(f) = (\_, \_, \_, s)$，$\Gamma(f) = (fp, fq)$，$p = \mathsf{BuildFunP}(\eta_i, f, \bar{v}, \mathcal{L}, fp)$ 以及 $\rho = \mathsf{BuildFunR}(\eta_i, f, \bar{v}, \mathcal{L}, fq)$ 成立，那么 $\Gamma; \chi; I; \mathsf{LI}; \rho; \mathsf{false} \models \{p\}s\{\mathsf{false}\}$。

6.3　扩展后 OSTaskDel 的验证过程

在介绍完了对验证框架的调整和扩展之后，下面给出扩展后 OSTaskDel 的验证过程，首先给出 p_{init} 和 LI 的定义如下：

$$\begin{array}{lcl} p_{init} & \stackrel{\text{def}}{=} & get\_addr(OSTCBCur, flag) \xmapsto{\texttt{Tint32}} 1 \\ \mathsf{LI} & \stackrel{\text{def}}{=} & \exists v.\ get\_addr(OSTCBCur, flag) \xmapsto{\texttt{Tint32}} v \end{array}$$

新的全局不变式 I_c 定义如下：

$$\begin{array}{rcl} I_c & \stackrel{\text{def}}{=} & ((get\_addr(OSTCBCur, flag) \xmapsto[\circ]{\texttt{Tint32}} \texttt{Vint}(0) * I') \vee \\ & & ((get\_addr(OSTCBCur, flag) \xmapsto[\circ]{\texttt{Tint32}} \texttt{Vint}(1)) * I) \end{array}$$

p_{init} 表示每个 API 开始执行时 flag 域都为 1，表示 API 开始执行时 flag 域为 1。LI 只要求 flag 域存在，不关心其中具体的值。图6.7 给出了 OSTaskDel 在引入局部不变式和分数权限（fractional permission）后的删除自己情况下大致推理过程（其中省去了一些和局部不变式和分数权限不相关的断言，例如 $\mathsf{OS}[\_, \_, \_, \_]$），推理过程这里不再逐条介绍。

INT8U OSTaskDel (INT8U prio) :

1　OS_TCB　*ptcb;

$$\boxed{\mathrm{p_{init}} * \exists v.\mathrm{ptcb} \overset{\mathtt{Tptr(OS\_TCB)}}{\longmapsto_{\mathrm{l}}} v}$$

2　OS_ENTER_CRITICAL();

$$\boxed{\mathrm{p_{init}} * \exists v.\mathrm{ptcb} \overset{\mathtt{Tptr(OS\_TCB)}}{\longmapsto_{\mathrm{l}}} v * \mathrm{I_c}}$$

$$\Downarrow$$

$$\boxed{(\exists v.\mathrm{ptcb} \overset{\mathtt{Tptr(OS\_TCB)}}{\longmapsto_{\mathrm{l}}} v * \mathrm{get\_addr}(\mathrm{OSTCBCur}, \mathrm{flag}) \overset{\mathtt{Tint32}}{\longmapsto} \mathtt{Vint}(1)) * \mathrm{I}}$$

. . . . . .

```
if (ptcb == OSTCBCur)
   ptcb->flag = 0;
```

$$\boxed{(\exists v.\mathrm{ptcb} \overset{\mathtt{Tptr(OS\_TCB)}}{\longmapsto_{\mathrm{l}}} v * \mathrm{get\_addr}(\mathrm{OSTCBCur}, \mathrm{flag}) \overset{\mathtt{Tint32}}{\longmapsto} \mathtt{Vint}(0)) * \mathrm{I}'}$$

$$\Downarrow$$

$$\boxed{(\mathsf{LI} * \exists v.\mathrm{ptcb} \overset{\mathtt{Tptr(OS\_TCB)}}{\longmapsto_{\mathrm{l}}} v * \mathrm{get\_addr}(\mathrm{OSTCBCur}, \mathrm{flag}) \underset{\circ}{\overset{\mathtt{Tint32}}{\longmapsto}} \mathtt{Vint}(0)) * \mathrm{I}'}$$

$$\Downarrow$$

$$\boxed{(\mathsf{LI} * \exists v.\mathrm{ptcb} \overset{\mathtt{Tptr(OS\_TCB)}}{\longmapsto_{\mathrm{l}}} v * \mathrm{I_c}}$$

16　OS_EXIT_CRITICAL();

$$\boxed{\mathsf{LI} * \exists v.\mathrm{ptcb} \overset{\mathtt{Tptr(OS\_TCB)}}{\longmapsto_{\mathrm{l}}} v}$$

. . . . . .

图 6.7　OSTaskDel 的大致推理过程

6.4　本章小结

在内核代码验证中，由于大多数对共享资源的访问都是发生在临界区中，所以通过全局不变式来描述共享资源是合适的，但是存在一些特殊的时刻（例如前述的 $t_{delself}$），全局不变式结构会被破坏。分数权限（fractional permission）和局部不变式可以刻画出这些特殊的程序点，从而提高程序逻辑的表达力，并完成更多程序的验证。

分数权限（fractional permission）和局部不变式是在验证 OSTaskDel 函数过程中发现问题后扩展的，这也可以看出验证框架的设计和确立不是一蹴而就的，在应用验证框架验证实际系统内核的过程中，需要调整和扩展。

第七章　自动证明策略

为了提高验证人员使用 Coq 证明时的效率，本文开发了一系列的自动证明策略来帮助验证。自动证明策略侧重于解决三个问题：

（1） 推理系统中的规则是针对一般情况设计的，证明过程中直接使用这些规则会很不方便。例如处理赋值语句的 ASSIGN规则，在实际的使用过程中因为左表达式存在多种情况，可能是变量、数组元素、结构体成员变量等等，如果是结构体成员变量或者是数组元素还需要去展开结构体或数组的定义（图5.4），结构体和数组断言都是递归定义的，展开后会使前条件变得非常复杂，而且可读性很差。所以本文开发了大量的辅助规则，这些规则针对具体情况设计，例如，对于结构体成员变量的赋值有专门的辅助规则，这些规则能够大大简化证明。

（2） 推理系统中存在大量的推理规则（包括辅助推理规则），对于验证人员来说记忆这些内容是很大的负担。本文的自动证明策略可以根据当前证明上下文来自动选择推理规则，这样验证人员只需要使用推理系统提供的自动证明策略，底层的具体规则对于验证人员不再可见，大大减轻了验证人员的学习负担。

（3） Coq 证明的过程中有大量看起来明显成立，但是机器证明非常繁琐的命题。例如，有时需要证明如下命题：

$$p_1 * p_2 * \cdots * p_i * \cdots * p_n \Longrightarrow p_i * p_1 * p_2 * \cdots * \cdots * p_n$$

因为已知“*”满足交换律和结合律，所以上述命题显然成立。但是如果没有自动证明策略，验证人员可能要应用很多次“*”上的交换律和结合律才能完成证明。再比如很多简单的数学上的性质证明，比如：

$$\forall\, n,\ m.\ (0 \leqslant n < 8 \wedge 0 \leqslant m < 8) \Longrightarrow 0 \leqslant n * m < 64$$

因为 Coq 自带的证明策略无法自动证明多个变量乘法上的性质，所以要验证上述命题，验证人员需要考虑 n 和 m 的每种可能的取值。这些简单而冗长的证明会耗费验证人员大量不必要的时间。所以针对很多这类问题，本文开发了很多领域专用（domain-specific）的自动证明策略。

下面以 μC/OS-II 中的一个简单的 API 函数 OSTimeGet 的证明作为例子来看下如何使用自动证明策略。图7.1上方是 OSTimeGet 的实现，下方是 Coq 证明脚本。下面来解释每行自动证明策略的用途：

(1) “init spec”策略来对验证条件做一些初始化，“init spec”是每个函数证明都会使用的自动证明策略。

```
void OSTimeGet() {
    INT32U ticks;
L1  OS_ENTER_CRITICAL();
L2  ticks = OSTime;
L3  OS_EXIT_CRITICAL();
L4  return (ticks);
}
```

```
Lemma OSTimeGetRight:
  forall r p  ,
    Some r = BuildRetA' os_api OSTimeGet tmgetapi nil ->
    Some p = BuildPreA' os_api OSTimeGet tmgetapi nil ->
    exists t decl1 decl2 s,
      os_api OSTimeGet = Some (t,decl1,decl2,s) /\
      {|OSQ_spec , GetHPrio, I, r, Afalse|} |- {{p}} s {{Afalse}}.
Proof.
  init spec.   (*Initialize Specification*)
  hoare forward. (*L1:  OS_ENTER_CRITICAL();*)

  hoare unfold pre. (*L2: ticks = OSTime; *)
  hoare forward.

  hoare abscsq.
  eapply OSTimeGet_high_level_step; pauto.

  hoare forward.  (*L3:  OS_EXIT_CRITICAL();*)
  unfold AOSTime.
  sep pauto.
  pauto.

  hoare forward.   (*L4: return (ticks); *)
Qed.
```

图 7.1　使用自动证明策略验证 OSTimeGet()

(2) 因为采用前向推理，“hoare forward”会取出第一行代码，并根据该行代码的内容来自动选择推理规则。这里判断是汇编原语 **encrt**，那么会自动选择 ENCRT 规则。

(3) 因为下一行是赋值语句，“hoare unfold pre”会根据赋值语句的左表达式和右表达式对前断言做一些预处理。这是每行赋值语句之前会使用的策略。

(4) 此行“hoare forward”和之前一样，取出下一行代码并判断出这是一个对全局变量赋值的语句，然后选择对应的辅助推理规则并应用。因为很多推理规则涉及到表达式值的计算和断言推导，这里“hoare forward”内部会

分别调用“symbolic execution”策略和“sep auto”策略来处理。

(5) 因为推理过程中有时需要使用 ABSCSQ 规则让高层规范代码执行一次，但何时执行是和具体函数密切相关的，所以很难用自动证明策略来判断何时使用 ABSCSQ 规则。当需要应用 ABSCSQ 规则时，可以使用“hoare abscsq”策略，它会应用 ABSCSQ 规则并自动处理要证明的附加条件（side-condition）。

(6) 这里将高层规范代码对前断言的修改抽象为一个引理，在证明脚本中只需要应用这个引理就好了。该引理的证明和函数体的证明是相互独立的。

(7) “hoare forward”策略判断当前代码是汇编原语 **excrt**，那么会自动选择 EXCRT 规则并应用。应用 EXCRT 规则需要证明大量的附加条件，“hoare forward”无法全部处理，8-10 行用来处理剩下的附加条件。

(11) “hoare forward”策略判断当前代码是不带返回值的函数返回语句，自动选择 RET 规则并应用。

在自动证明策略的帮助下，一共只用了 11 行 Coq 代码就完成了证明，而在一开始没有自动证明策略的情况下，只有 4 行 C 代码的函数 OSTimeGet 的证明一共使用了近 400 行 Coq 代码。在借鉴了已有的关于断言蕴含关系的自动推导和验证条件自动生成的工作 [51, 52, 55] 后，本文采用了 [51] 中的自动证明策略设计思想，并作了如下扩展：

- 将断言蕴含关系的自动推导策略“sep auto”扩展到关系型断言上。
- 将验证条件自动生成策略扩展到并发精化逻辑推理规则中。
- 开发了自动证明策略“mauto”来处理数学性质的证明。

下面将主要介绍这些扩展。

7.1　关系型断言的自动推导

断言自动推导是指自动证明“$p \Rightarrow p'$”这类命题。这里要求断言 p 和 p' 中不使用“$\wedge$”和“$\vee$”。sep auto 主要是按序调用下列四个子策略：sep_normal、sep_split、sep_cancel 和 sep_pure，下面将分别介绍，图7.2中是这些子策略会用到的一些引理。

sep_normal 用于将断言转化为一个规范的形式。满足要求的断言 p_f 的形式为 $\exists\, x_1, \cdots, x_n. p'_f$，并且 p'_f 要满足如下要求：（1）p'_f 中不包含存在量词；（2）p'_f 中不包含 emp，并且至多存在一个 true；（3）组成 p'_f 的子断言满足右结合；（4）ISR(_)、IE(_)、IS(_)、IS(_) 和 [|_|] 在 p'_f 中至多出现一次。

sep_normal 将断言 p 规范化的过程中首先会寻找 p 中包含存在量词的子断言 p_i，然后通过“*”上的交换律和结合律将 p_i 放到最左边，然后应用图7.2中

$$\frac{\mathrm{t} : \mathsf{Type} \quad A : \mathrm{t} \to \mathit{Asrt}}{(\exists\, \mathrm{x}.\, A\ \mathrm{x}) * \mathrm{p} \Leftrightarrow \exists\, \mathrm{x}.\, A\ \mathrm{x} * \mathrm{p}}\ \mathrm{L1} \qquad \frac{}{\Theta \models \langle \mathrm{P} \rangle * \mathrm{p} \Leftrightarrow (\Theta \models \mathrm{p} \wedge \mathrm{P})}\ \mathrm{L2}$$

$$\frac{}{\mathsf{ISR}(\mathit{isr}) * \mathsf{ISR}(\mathit{isr}') * \mathrm{p} \Leftrightarrow \langle \mathit{isr} = \mathit{isr}' \rangle * \mathsf{ISR}(\mathit{isr}) * \mathrm{p}}\ \mathrm{L3}$$

$$\frac{}{[|\mathbf{s}|] * [|\mathbf{s}'|] * \mathrm{p} \Leftrightarrow \langle \mathbf{s} = \mathbf{s}' \rangle * [|\mathbf{s}|] * \mathrm{p}}\ \mathrm{L4}$$

$$\frac{\mathrm{p} \Rightarrow \mathrm{p}'}{\mathrm{p} * \mathrm{q} \Rightarrow \mathrm{p}' * \mathrm{q}}\ \mathrm{L5} \qquad \frac{v = v' \quad \mathrm{p} \Rightarrow \mathrm{p}'}{\mathrm{a} \overset{\tau}{\mapsto} v * \mathrm{p} \Rightarrow \mathrm{a} \overset{\tau}{\mapsto} v' * \mathrm{p}'}\ \mathrm{L6}$$

$$\frac{v = v' \quad \mathrm{p} \Rightarrow \mathrm{p}'}{\mathrm{x} \mapsto v * \mathrm{p} \Rightarrow \mathrm{x} \mapsto v' * \mathrm{p}'}\ \mathrm{L7} \qquad \frac{v = v' \quad \mathrm{p} \Rightarrow \mathrm{p}'}{\mathrm{x} \mapsto_{\mathrm{l}} v * \mathrm{p} \Rightarrow \mathrm{x} \mapsto_{\mathrm{l}} v' * \mathrm{p}'}\ \mathrm{L8}$$

$$\frac{\bar{v} = \bar{v}' \quad \mathrm{p} \Rightarrow \mathrm{p}'}{\mathsf{Astruct}(\mathrm{a}, \tau, \bar{v}) * \mathrm{p} \Rightarrow \mathsf{Astruct}(\mathrm{a}, \tau, \bar{v}') * \mathrm{p}'}\ \mathrm{L9}$$

$$\frac{\bar{v} = \bar{v}' \quad \mathrm{p} \Rightarrow \mathrm{p}'}{\mathsf{Aarray}(\mathrm{a}, \tau, \bar{v}) * \mathrm{p} \Rightarrow \mathsf{Aarray}(\mathrm{a}, \tau, \bar{v}') * \mathrm{p}'}\ \mathrm{L10}$$

$$\frac{\bar{\mathrm{v}} = \bar{\mathrm{v}}' \quad \mathrm{p} \Rightarrow \mathrm{p}'}{\mathsf{sl}(\mathit{head}, \tau, \bar{\mathrm{v}}, \mathtt{Pf}) * \mathrm{p} \Rightarrow \mathsf{sl}(\mathit{head}, \tau, \bar{\mathrm{v}}', \mathtt{Pf}) * \mathrm{p}'}\ \mathrm{L11}$$

$$\frac{\bar{\mathrm{v}} = \bar{\mathrm{v}}' \quad \mathrm{p} \Rightarrow \mathrm{p}'}{\mathsf{dl}(\mathit{head}, \mathit{tail}, \tau, \bar{\mathrm{v}}, \mathtt{Pf}_{\mathrm{n}}, \mathtt{Pf}_{\mathrm{p}}) * \mathrm{p} \Rightarrow \mathsf{dl}(\mathit{head}, \mathit{tail}, \tau, \bar{\mathrm{v}}', \mathtt{Pf}_{\mathrm{n}}, \mathtt{Pf}_{\mathrm{p}}) * \mathrm{p}'}\ \mathrm{L12}$$

$$\frac{\mathbf{s} = \mathbf{s}' \quad \mathrm{p} \Rightarrow \mathrm{p}'}{[|\mathbf{s}|] * \mathrm{p} \Rightarrow [|\mathbf{s}'|] * \mathrm{p}'}\ \mathrm{L13} \qquad \frac{\mathit{isr} = \mathit{isr}' \quad \mathrm{p} \Rightarrow \mathrm{p}'}{\mathsf{ISR}(\mathit{isr}) * \mathrm{p} \Rightarrow \mathsf{ISR}(\mathit{isr}') * \mathrm{p}'}\ \mathrm{L14}$$

图 7.2　用于断言自动推理的引理

的引理 L1 将存在量词提前；在确定所有子断言中的存在量词都被提前之后，会去应用 L3、L4 等引理来去除 p 中多余的断言。

sep_split 首先使用 Coq 中的“destruct”策略去除 p 中的存在量词，然后通过“eexists”策略，来推迟 p' 中存在量词的实例化。然后通过引理 L2，来去掉 p 和 p' 中和状态无关的断言。

sep_cancel 主要是不停地使用 L5 ~ L14 等引理来消去子断言，直到所有子断言都被消去。这里不仅仅能够自动消去基本断言，还处理了封装后的断言 Astruct、Aarray 等。在具体的验证项目中可能存在很多不同的复杂数据结构，只要能够证明类似的引理，就可以通过调整自动证明策略来支持这些包含这些复杂数据结构的断言的自动推理。

sep_pure 主要是用来处理在使用上述引理过程中产生的需要验证的附加条件。

$$\frac{\begin{array}{l} p = x \xmapsto{\texttt{Tptr}(\tau)}_l \texttt{Vptr}(a) * \mathsf{Astruct}(a,\tau,\bar{v}) * p' * [|s|] \quad p \Rightarrow e =_{\tau_2} v \\ \tau = \texttt{Tstruct}(id_t, \mathcal{D}) \quad \texttt{upd\_str}(\bar{v}, \mathcal{D}, id, \tau_1, v, \bar{v}') \quad \tau_1 \propto \tau_2 \end{array}}{\Gamma;\chi;I;\rho;p_i \vdash \{\ p\ \}(*x).id = e \left\{ \begin{array}{l} x \xmapsto{\texttt{Tptr}(\tau)}_l \texttt{Vptr}(a) * \\ \mathsf{Astruct}(a,\tau,\bar{v}') * p' * [|s|] \end{array} \right\}} \ (\textsc{AssStr})$$

$$dlvl(\mathcal{D}) \stackrel{\text{def}}{=} \begin{cases} dlvl(\mathcal{D}') + 1 & \text{if} \quad \mathcal{D} = (\_,\tau) :: \mathcal{D}' \wedge \tau \neq \texttt{Tarray}(\_,\_) \wedge \\ & \qquad \tau \neq \texttt{Tstruct}(\_,\_) \\ dlvl(\mathcal{D}') + n * tvl(\tau) & \text{if} \quad \mathcal{D} = (\_,\tau) :: \mathcal{D}' \wedge \tau = \texttt{Tarray}(\tau, n) \\ dlvl(\mathcal{D}') + dlvl(\mathcal{D}_1) & \text{if} \quad \mathcal{D} = (\_,\tau) :: \mathcal{D}' \wedge \tau = \texttt{Tstruct}(\_, \mathcal{D}_1) \\ 0 & \text{otherwise} \end{cases}$$

$$tvl(\tau) \stackrel{\text{def}}{=} \begin{cases} n * tvl(\tau') & \text{if} \quad \tau = \texttt{Tarray}(\tau', n) \\ dlvl(\mathcal{D}) & \text{if} \quad \tau = \texttt{Tstruct}(\_, \mathcal{D}) \\ 1 & \text{otherwise} \end{cases}$$

$$\begin{array}{l} \texttt{upd\_str}(\bar{v}, \mathcal{D}, id, \tau, v, \bar{v}') \stackrel{\text{def}}{=} \\ \quad \exists \mathcal{D}_1, \mathcal{D}_2, \bar{v}_1, \bar{v}_2, v_o, n.\ \mathcal{D} = \mathcal{D}_1 +\!+ (id,\tau) :: \mathcal{D}_2 \wedge dlvl(\mathcal{D}_1) = n \wedge \\ \qquad\qquad \bar{v} = \bar{v}_1 +\!+ v_o :: \bar{v}_2 \wedge \bar{v}' = \bar{v}_1 +\!+ v :: \bar{v}_2 \wedge len(\bar{v}_1) = n \end{array}$$

图 7.3　结构体赋值规则

7.2　并发精化逻辑推理规则的验证条件自动生成

本文对验证条件自动生成策略的扩展主要包括以下几个方面：

（1） 扩展 hoare forward，使其可以自动自动识别汇编原语并应用相应的推理规则。

（2） 引入大量赋值语句的辅助推理规则，使得 hoare forward 对于赋值语句的处理更加自动化。

（3） 引入 hoare abscsq 策略，hoare abscsq 会应用 ABSCSQ规则，并自动处理生成的验证条件。

（1）和（3）比较简单，这里不再具体介绍。下面重点介绍如何设计赋值语句的辅助推理规则。

在实际程序中经常需要对结构体成员变量或者是数组成员变量赋值，而 ASSIGN 规则需要展开封装好的描述结构体或者数组的断言，这些断言展开后会使得验证条件变得非常复杂，可读性也非常差。为了解决这个问题，本文设计了很多用于结构体、数组等复杂数据结构的赋值规则，使得验证过程中不需要展开封装好的断言的定义，从而减轻验证人员的负担。图7.3给出了结构体成员变量赋值的辅助推理规则 AssStr，下面以此为例来了解下辅助推理规则。x 是一个指向结构体的指针，赋值语句 $(*x).id = e$ 表示将表达式 e 的

值赋值给 x 指向的结构体的成员 id。前条件中仅需要给出描述变量 x 的断言 $x \xmapsto{\texttt{Vptr}(a)}_l \texttt{Tptr}(\tau)$，以及描述 x 指向的结构体的断言 $\mathsf{Astruct}(a, \tau, \bar{v})$，而不需要展开 Astruct 的定义得到描述 $(*x).\mathrm{id}$ 的断言。赋值规则只是将保存结构体内容的值表 $\bar{v}$ 变换为赋值后的 $\bar{v}'$，从而保证后条件中是完整的 Astruct。$\mathrm{upd\_str}(\bar{v}, \mathcal{D}, \mathrm{id}, \tau, v, \bar{v}')$ 表示 $\bar{v}'$ 只是将值表 $\bar{v}$ 中 id 对应的值替换为 v，其他域相同，并且 id 的类型为 τ，upd_str 定义中的 $\mathrm{len}(\bar{v}_1)$ 表示获取 $\bar{v}_1$ 的长度。

7.3　数学性质的自动证明策略

由于操作系统内核中存在大量位运算，这些位运算会产生很多明显成立但是证明过程非常繁杂的命题，例如：

$$\forall x.x < 64 \rightarrow x \gg 3 < 8\,; \quad \forall x.x < 8 \rightarrow (x \ll 3)\&7 = 0$$

同时观察到这些命题中的变量（上例中的 x）都是在有限范围内的，所以本文开发了策略“mauto”，mauto 暴力穷举的方法，尝试变量在有限范围内的每种取值。虽然在变量取值范围较大时 mauto 策略的执行时间较长，但是相对于不用 mauto 的情况还是节省了大量的验证时间。mauto 的具体实现在 Coq 代码 [3] 中的“/CertiOS/tactics/mathsolver.v”文件中。

第八章　验证 μC/OS-II

本文成功地使用验证框架验证了一个商用操作系统 μC/OS-II V2.52 的核心模块，一共验证了 1400 行左右的 C 代码（不计算空行和注释），其中包含调度程序、时钟中断、互斥信号量模块、消息队列模块、消息邮箱模块、信号量模块和时间管理模块。这 1400 行 C 代码在操作系统原格式下一共 3468 行（包含空行和注释）。本文验证的函数覆盖了 μC/OS-II 中 68% 的常用函数 [1]。本文没有验证一些功能和已验证的模块类似的通信和同步模块（例如，消息标志模块），系统初始化的代码，以及一些不常用的 API （例如，OSTaskChangePrio）。图8.1上半部分展示了 μC/OS-II 的函数调用关系图，其中黄色的函数是常用的 API，下半部分展示了本文的验证工作，其中红色的函数验证过的函数。

本文一共通过 225,000 行 Coq 证明脚本（Coq8.4pl6）完成了验证框架和 μC/OS-II 核心模块的验证。表8.1给出了每个组成部分 Coq 证明脚本的统计。其中 76000 行左右用于验证框架的定义和正确性证明；有 100000 行左右是用于编码 μC/OS-II、定义 μC/OS-II 的规范、一些基本库函数以及 PIF 性质的证明（PIF 会在下一章具体介绍），这其中有 25000 行是通过 emacs 脚本自动生成的，用于将较大的证明拆分，从而提高证明脚本的执行速度；还有 39000 行左右用于证明 μC/OS-II 中各个函数的精化关系。Coq 脚本的编译一共需要 16 个小时（机器要求 3.6GHz 的 CPU 主频和至少 36G 的内存）。整个验证工作一共耗费了验证团队 5.5 人年的时间，其中 4 人年用于开发验证框架，1 人年用于验证 μC/OS-II 中的第一个模块（消息队列模块），但随着验证框架的逐步稳定以及自动证明策略功能越来越完善，剩余的代码证明（900 多行 C 代码）只耗费了 6 人月的时间。在自动证明策略的帮助下最终 1 行 C 代码的验证只需要 27 行左右的 Coq 脚本。

验证框架	Coq 行数
基本库	32061
建模和程序逻辑	23095
自动证明策略	21050
总计	76206

验证 μC/OS-II	Coq 行数
C 程序 Coq 编码规范	1824
	6012
PIF 定义和证明	9570
μC/OS-II 性质库	65818
自动生成代码	25357
总计	108581

验证的模块	C 代码行数	Coq 行数
全局变量声明	187	-
消息队列模块	240	4537
信号量模块	166	2441
消息邮箱模块	171	3326
互斥信号量模块	301	17331
时间管理模块	39	861
时钟中断	17	443
任务管理	73	4880
内部函数	195	5447
正确性定理	-	501
总计	1389	39767

表 8.1　μC/OS-II 验证代码量统计

已有的操作系统验证工作都是针对自己开发的操作系统，这些操作系统在开发过程中已经考虑到验证的问题。μC/OS-II 是一个被广泛使用的操作系统，在开发过程中并没有考虑到验证的问题，其在设计过程中为了效率会使用一些复杂数据结构和算法，这会给验证工作带来很多困难。例如，μC/OS-II 中为了调度程序能够迅速找到当前最高优先级的就绪任务，其会用一个位图来保存所有的就绪任务，每次调度程序会利用位图的性质在 O(1) 时间内找出最高优先级的就绪任务。而很多只考虑验证的操作系统可能只是简单的用一个链表来保存所有就绪任务，这样会使得验证过程非常简单，但是查找优先级最高任务的时间复杂度会变为 O(n) （n 为系统任务的个数）。

下面将简单介绍对 μC/OS-II 的验证工作。首先在第8.1节介绍 μC/OS-II 的抽象数据结构的定义；第8.2 节介绍如何定义全局资源不变式；第8.3节介绍验证过程中对 μC/OS-II 源码的修改；第8.4节介绍对优先级反转性质的研究以及 μC/OS-II 互斥信号量模块无优先级反转的验证工作。

8.1　对 uC/OS-II 的抽象

对 μC/OS-II 的抽象分为两个部分，一个是定义 μC/OS-II 的抽象内核状态，还有定义 μC/OS-II 代码的规范。

图8.2给出了对 μC/OS-II 内核状态的抽象。μC/OS-II 的抽象内核状态 Σ_μ 一共包含四个部分，分别是抽象任务表 α，抽象等待事件表 β，当前正在运行任务的任务号 t 以及系统时间 k。

抽象任务表 α 记录了系统中的每个任务的当前优先级（因为优先级可能动态变化）以及任务状态 *ts*。任务状态有两种，rdy 表示任务就绪随时可以调度执行（当前正在运行的任务状态也是 rdy）; 另一种是等待状态 $\mathsf{wait}(wt, \mathrm{k})$，$\mathrm{k}$ 是需要等待的时间，k 在每次时钟中断到来时都会减一，减至 0 时会将该任务状态重新设置为就绪。wt 表示等待的类型，tm 表示只是单纯的等待时间到来；$\mathsf{mtx}(eid)$ 表示在等待一个互斥信号量，eid 是等待的信号量的 ID，如果任务获得该信号量也会变成就绪状态；$\mathsf{sm}(eid)$ 表示等待一个普通信号量 eid；$\mathsf{mq}(eid)$ 表示等待一个消息队列 eid；$\mathsf{mbox}(eid)$ 表示等待一个消息邮箱 eid。

抽象等待事件表 β 保存了系统中的每个通信和同步事件，通信和同步事件 ed 一共有四种：（1）$\mathsf{mutex}(\mathrm{Q}, \mathrm{pr}, w)$ 表示该同步事件是一个互斥信号量，Q 是所有等待该互斥信号量的任务，pr 是该互斥信号量自身的优先级（为了防止优先级反转，μC/OS-II 给每个互斥信号量赋予一个优先级，下一章会具体介绍），w 是当前占有该互斥信号量的任务信息，包含占有该互斥信号量的任务号和该任务在占有时的优先级；（2）$\mathsf{sem}(\mathrm{Q}, \mathrm{k})$ 表示该同步事件是一个信号量，Q 是所有等待该信号量的任务，k 是该信号量目前可用的个数（因为信号量控制的资源个数可能不止一个）；（3）$\mathsf{mbox}(\mathrm{Q}, v)$ 表示该通信事件是个消息邮箱，Q 是所有等待该消息邮箱的任务，v 是目前该消息邮箱中保存的信息，如

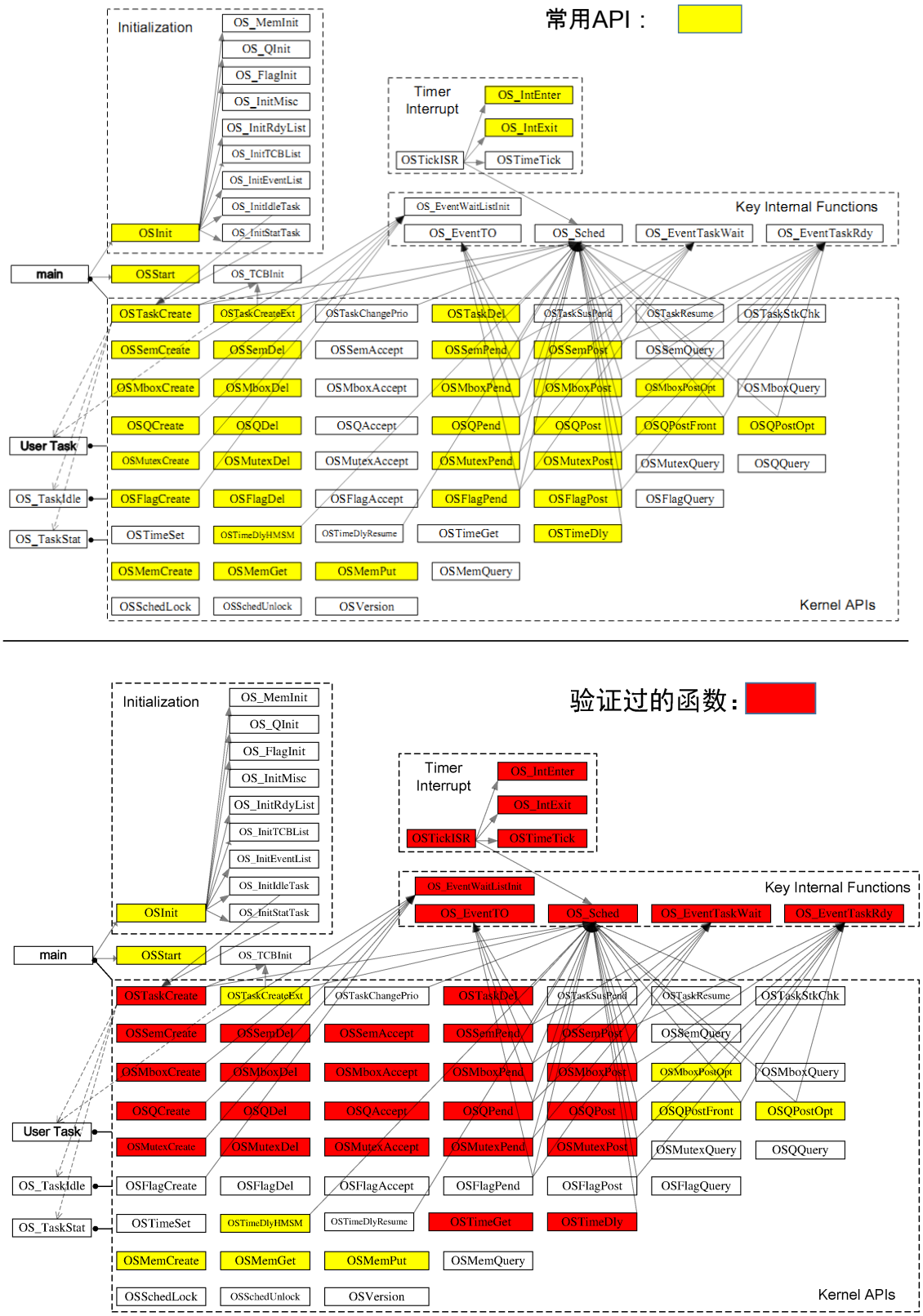

图 8.1　μC/OS-II 中验证过的函数

$$(\mathit{HAbsSt}) \quad \Sigma_\mu ::= \{\mathtt{tcbls} \rightsquigarrow \alpha,\ \mathtt{ecbls} \rightsquigarrow \beta,\ \mathtt{ctid} \rightsquigarrow \mathtt{t},\ \mathtt{time} \rightsquigarrow \mathtt{k}\}$$

$$(\mathit{HTCBLs}) \quad \alpha ::= \{\mathtt{t}_1 \rightsquigarrow (\mathtt{pr}_1, ts_1), \ldots, \mathtt{t}_n \rightsquigarrow (\mathtt{pr}_n, ts_n)\}$$
$$(\mathit{Priority}) \quad \mathtt{pr} \in \mathit{int32} \qquad (\mathit{HStatus}) \quad ts ::= \mathsf{rdy} \mid \mathsf{wait}(wt, \mathtt{k})$$
$$(\mathit{WaitType}) \quad wt ::= \mathsf{tm} \mid \mathsf{mtx}(\mathtt{eid}) \mid \mathsf{sm}(\mathtt{eid}) \mid \mathsf{mq}(\mathtt{eid}) \mid \mathsf{mbox}(\mathtt{eid})$$
$$(\mathit{HEvtId}) \quad \mathtt{eid} \in \mathit{Addr}$$

$$(\mathit{HECBLs}) \quad \beta ::= \{\mathtt{eid}_1 \rightsquigarrow \mathtt{ed}_1, \ldots, \mathtt{eid}_n \rightsquigarrow \mathtt{ed}_n\}$$
$$(\mathit{HECBDt}) \quad \mathtt{ed} ::= \mathsf{mutex}(\mathtt{Q}, \mathtt{pr}, w) \mid \mathsf{sem}(\mathtt{Q}, \mathtt{k}) \mid \mathsf{mbox}(\mathtt{Q}, v) \mid \mathsf{msgq}(\mathtt{Q}, \bar{v}, \mathtt{k})$$
$$(\mathit{WaitQ}) \quad \mathtt{Q} \in \mathtt{nil} \mid \mathtt{t} :: \mathtt{Q} \qquad (\mathit{MtxOwner}) \ w ::= \bot \mid (\mathtt{t}, \mathtt{pr})$$

图 8.2　μC/OS-II 抽象内核状态

果 $v = \mathtt{Vnull}$，则表示该消息邮箱是空的；（4）$\mathsf{msgq}(\mathtt{Q}, \bar{v}, \mathtt{k})$ 表示该通信事件是一个消息队列，消息队列允许缓存多个消息，Q 是所有等待该消息队列的任务，$\bar{v}$ 是目前该消息队列中保存的信息，k 表示该消息邮箱允许缓存的最大消息个数。

8.2　资源不变式定义

资源不变式 I 的定义对于操作系统验证是至关重要的，其精确地刻画了全局资源需要保持的性质。如果 I 定义得太弱，程序在进入临界区时可能获得的全局资源的性质可能无法让程序正确的推理下去；如果定义太强，程序在退出临界区时可能无法保证 I 继续成立。对于关系型程序逻辑来说，I 还需要描述出底层机器状态和高层抽象内核状态之间的关系。

操作系统内部存在大量数据结构，这些数据结构间的耦合度非常高。为了使得不变式的结构清晰且易于推理，对全局资源的不变式定义遵循如下原则：（1）对内存的描述和性质的描述分离。这样可以保证在用 hoare forward 等自动证明策略推理时，只需要访问描述内存的断言，而不需要展开描述性质的断言。这样会大大减少证明过程中前提的个数，从而提高 Coq 的证明效率。（2）不同数据结构间的性质尽量定义在高层。相对于底层描述内存的断言，描述高层抽象内核结构的断言清晰易读，性质定义在高层会大大提高不变式的可读性。

图8.3给出了不变式定义的总体结构，图中上方表示描述抽象内核结构的断言（例如：α 表示抽象任务表），下面是描述底层实际数据结构的断言。下面将以 μC/OS-II 中保存所有任务控制块（TCB）的双向链表（全局任务表）为例来看看如何定义抽象内核状态和对应的内存之间的关系（$\mathrm{R}_{\mathtt{tcbls}}$），以及不同抽象内核状态之间的关系（$\mathrm{RH}_{\mathtt{tcb\_ecb}}$）。

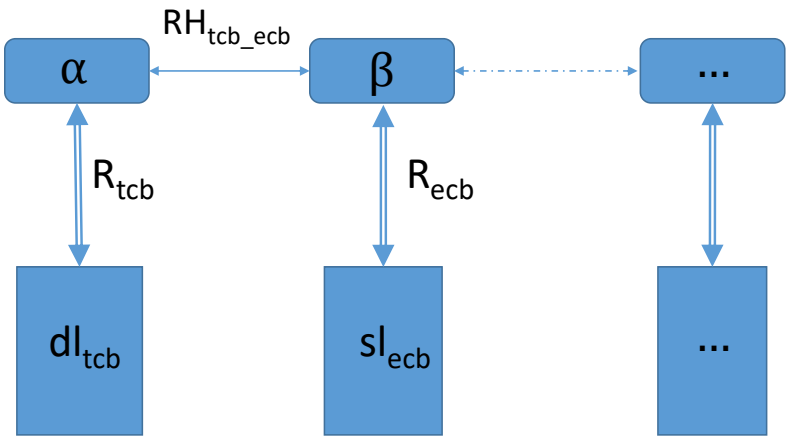

图 8.3　不变式结构

```
typedef struct os_tcb {
    struct os_tcb *OSTCBNext;
    struct os_tcb *OSTCBPrev;
    OS_EVENT      *OSTCBEventPtr;
    void          *OSTCBMsg;
    INT16U         OSTCBDly;
    INT8U          OSTCBStat;
    INT8U          OSTCBPrio;
    INT8U          OSTCBX;
    INT8U          OSTCBY;
    INT8U          OSTCBBitX;
    INT8U          OSTCBBitY;
} OS_TCB;
```

图 8.4　OS_TCB 结构体的定义

图8.4给出了 μC/OS-II 中 TCB 的类型。任务控制块 OS_TCB 共有 11 个域，其中 OSTCBNext 和 OSTCBPrev 分别是用于连接到全局任务控制块链表的指针；OSTCBEventPtr 用于指向等待的事件（如果该任务没有等待任何事件则该域等于 Vnull）；OSTCBMsg 用于保存该任务收到的消息；OSTCBDly 用于保存该任务还需要等待的时间；OSTCBStat 用于表示该任务当前的状态；OSTCBPrio 用于保存该任务当前的优先级；OSTCBX、OSTCBY、OSTCBBitX 和 OSTCBBitY 用于快速修改就绪任务表。验证框架抽象了任务栈，TCB 中用于保存任务栈的域也同样被抽象掉。

图8.5给出了描述全局任务链表的不变式 AOSTCBList。AOSTCBList 描述了四个部分：（1）指向全局任务表的全局变量 OSTCBList；（2）包含所有任务控制块的双向链表（双向链表 dl 的定义在图5.4中）；（3）抽象任务表 α；（4）全局任务表和抽象任务表需要满足的性质 R_{tcbls}。

$$\mathsf{AOSTCBList}(\bar{\mathtt{v}},\alpha) \stackrel{\text{def}}{=} \exists a_h, tail.\mathsf{OSTCBList}\mapsto \mathtt{Vptr}(a_h) * \mathtt{tcbls}\rightarrowtail\alpha * \mathsf{dl}(a_h, tail, \bar{\mathtt{v}}, \mathsf{TCB\_N}, \mathsf{TCB\_P}) * \langle \mathsf{R}_{\mathtt{tcbls}}(a_h,\bar{\mathtt{v}},\alpha)\rangle$$

$$\mathsf{R}_{\mathtt{tcbls}}(a_h,\bar{\mathtt{v}},\alpha) \stackrel{\text{def}}{=} \begin{cases} \alpha = \{a \rightsquigarrow (\mathtt{pr}, ts)\} \uplus \alpha' \wedge \mathsf{R}_{\mathtt{tcb}}(\bar{\mathtt{v}},(\mathtt{pr},ts)) \wedge \mathsf{RL}(\bar{\mathtt{v}}) \wedge \mathsf{R}_{\mathtt{tcbls}}(\mathsf{TCB\_N}(\bar{\mathtt{v}}),\bar{\mathtt{v}}',\alpha') & \text{if } \bar{\mathtt{v}} = \bar{\mathtt{v}} :: \bar{\mathtt{v}}' \\ \alpha = \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{R}_{\mathtt{tcb}}(\bar{\mathtt{v}},(\mathtt{pr},ts)) \stackrel{\text{def}}{=} \mathsf{TCB\_Prio}(\bar{\mathtt{v}}) = \mathtt{pr} \wedge \cdots$$

$$\mathsf{RL}(\bar{\mathtt{v}}) \stackrel{\text{def}}{=} (\mathsf{TCB\_Prio}(\bar{\mathtt{v}}) = \mathtt{Vint}(k) \wedge 0 \leqslant k \leqslant 63) \wedge \cdots$$

$$\mathsf{TCB\_N}(\bar{\mathtt{v}}) \stackrel{\text{def}}{=} \begin{cases} v & \text{if } \bar{\mathtt{v}} = v :: \bar{\mathtt{v}}' \\ \bot & \text{otherwise} \end{cases} \qquad \mathsf{TCB\_P}(\bar{\mathtt{v}}) \stackrel{\text{def}}{=} \begin{cases} v_2 & \text{if } \bar{\mathtt{v}} = v_1 :: v_2 :: \bar{\mathtt{v}}' \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{TCB\_Prio}(\bar{\mathtt{v}}) \stackrel{\text{def}}{=} \begin{cases} v_7 & \text{if } \bar{\mathtt{v}} = v_1 :: \cdots :: v_7 :: \bar{\mathtt{v}}' \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{RH}_{\mathtt{tcb\_ecb}}(\alpha,\beta) \stackrel{\text{def}}{=} \forall t, eid.\ \alpha(t) = (\_, \mathsf{wait}(\_(eid), \_)) \Leftrightarrow \exists Q.\ \beta(eid) = (\_(Q,\_)) \wedge t \in Q$$

图 8.5　描述 OSTCBList 的不变式

$\mathsf{R}_{\mathtt{tcbls}}$ 要求双向链表的每个节点和抽象任务表中的每个元素一一对应，并且二者满足关系 $\mathsf{R}_{\mathtt{tcb}}$，例如这里 $\mathsf{R}_{\mathtt{tcb}}$ 可以要求 OS_TCB 中的 OSTCBPrio 域的值和高层抽象 TCB 中的 pr 域相等；同时还要求双向链表的每个节点自身需要满足性质 RL，例如这里可以要求 OS_TCB 中的 OSTCBPrio 域的取值范围是从 0 到 63。在一开始的版本中，我们将 $\mathsf{R}_{\mathtt{tcbls}}$ 的定义融合在 dl 中（因为二者都是对 $\bar{\mathtt{v}}$ 归纳定义的），但是证明过程中发现每操作一个 TCB 节点，前提中就会展开得到其相关的性质，这样会使得前提变得庞大复杂，不仅仅影响了验证人员理解证明上下文的难度，还会使 Coq 的执行效率大大下降。

$\mathsf{RH}_{\mathtt{tcb\_ecb}}$ 表示对于一个任务 t，如果其当前状态为等待一个事件 eid，那么其一定会出现在 eid 的等待任务表 Q 中，反之亦然。可以看出 $\mathsf{RH}_{\mathtt{tcb\_ecb}}$ 的定义简单清晰，如果将该性质定义在底层会非常复杂。

8.3　对源码的修改

本文在验证过程中对源码作了一些不影响函数功能的修改，主要包含下列几种情况：

- 底层机器模型假设应用程序访问的资源 Δ 和内核程序访问的资源 Λ 是相互分离的。系统中存在如下 API：

  void OSSemPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)

  其中 err 是一个指向用户自己定义的变量的指针，OSSemPend 会将出错信息写入 err 指向的地址，应用程序代码通过访问 err 指向的内存来得到出错

信息。可以看出，此时内核代码访问并修改了用户资源 Δ。对于这种情况我们会将出错信息改到返回值中，例如 OSSemPend 修改后的代码如下：

INT8U OSSemPend (OS_EVENT *pevent, INT16U timeout)

- 系统对用户调用 API 时有很多默认的要求，例如，μC/OS-II 要求用户调用 OSSemPend 时传入的参数 pevent 指向的是一个已经被创建并初始化好的信号量，所以在 OSSemPend 的函数体中不再检查 pevent 是否指向合法的信号量（只检查 pevent 的值是否为 `Vnull`），而验证框架假设的是任意的用户，所以会在代码中加入一些参数检查。

- 为了保证资源不变式结构清晰，对不变式作了一些弱化。例如，对于当前正在运行的任务 t，其实系统能够保证除了在一些特殊的程序点，t 的状态始终是就绪的。较强的不变式可以如下定义：

$$(\neg P \Rightarrow (I \wedge \mathsf{RdyCur})) \wedge (P \Rightarrow (I \wedge \neg \mathsf{RdyCur}))$$

其中 P 用来表示是否在这些特殊程序点，RdyCur 表示当前运行的任务是否是就绪状态，I 是不变式的其他部分。可以看出这样会导致不变式的定义会比较复杂，而且为了判断这些特殊程序点，P 中会引入很多辅助变量。实际中，不变式就定义为 I，这样其实丢失了一些信息，但是会使得不变式简单清晰。当需要获得当前程序点 t 是就绪态的信息时，我们会插入如下代码:

```
if (OSTCBCur → OSTCBStat != OS_RDY){
    return NC_ERR;
}
```

实际中我们知道 NC_ERR 这种错误是不会发生的。

8.4　PIF 性质的证明

在多任务系统中，不同任务间存在共享资源，操作系统一般会提供信号量等同步机制来保证数据同步。有时低优先级的任务已经持有了某个共享资源，此时如果一个高优先级的任务如果想要访问该共享资源需要等待低优先级的任务释放该资源，这种现象被称为优先级反转（Priority-Inversion）。图8.6是一个发生优先级反转的例子，系统一开始任务 C 执行 P(S) 申请访问共享资源 S 并获得 S 的占有权，然后高优先级任务 A 被创建并调度执行，此时任务 A 也需要访问资源 S，但是因为 C 已经占有了资源 S，所以 A 被迫等待资源 S 直到任务 C 执行 V(S) 释放资源 S。

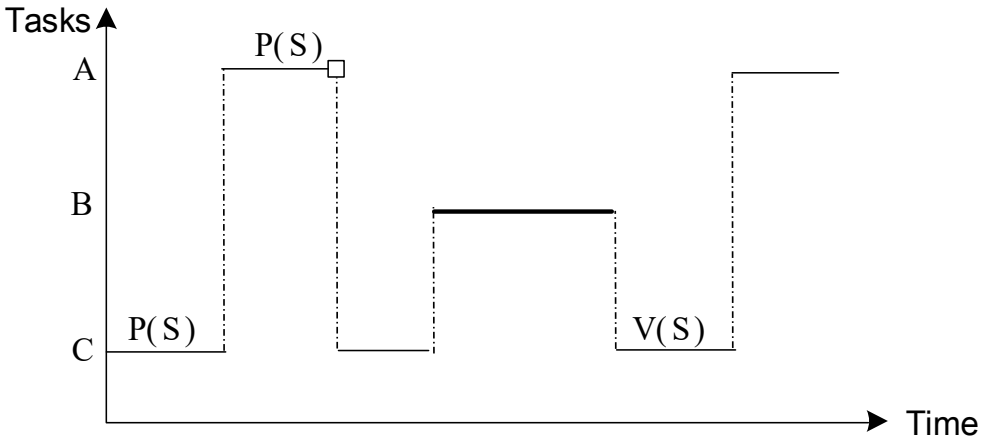

图 8.7　无边界优先级反转（unbounded-priority-inversion）

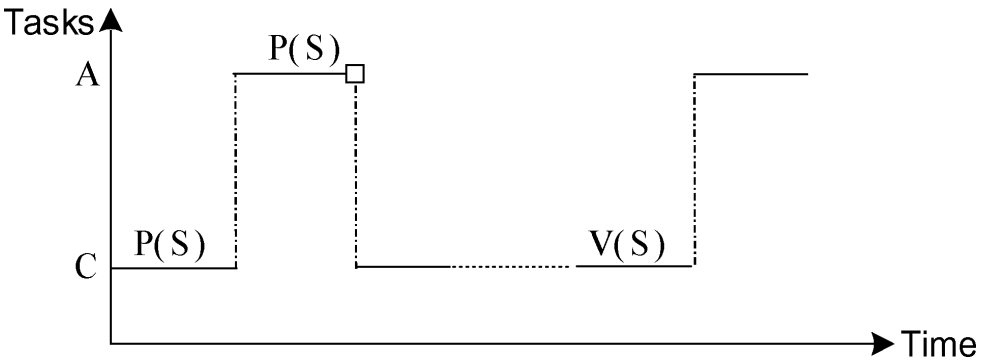

图 8.6　优先级反转

除非能保证所有任务在访问共享资源时，共享资源都未被其他低优先级的任务占有，否则为了保证数据同步，优先级反转的现象很难避免。一般能够接受系统出现图8.6 中所示的情况，因为任务对共享资源的访问一般都会在较短时间结束，图8.6中任务 A 虽然在等待一个低优先级的任务 C，但等待的时间是有限的。

图8.7给出了一个更糟糕的优先级反转的例子，假设任务优先级 A>B>C，一开始任务 C 执行 P(S) 申请访问共享资源 S 并获得 S 的占有权；然后任务 A 被创建并调度执行，此时任务 A 也需要访问资源 S，但是因为 C 已经占有了资源 S，所以 A 被迫等待资源 S 直到任务 C 执行 V(S) 释放资源 S；但是 C 在执行时，任务 B 被创建并调度执行，此时任务 A 不仅仅需要等待任务 C 还需要等待任务 B，但任务 B 不在临界区中，所以执行时间是不确定的。这种情况对于实时系统来说是致命的，因为高优先级的任务 A 可能永远无法被调度执行，这种高优先级任务可能无限期等待低优先级任务执行的现象被称为无边界优先级反转（unbounded-priority-inversion）。

存在许多协议和算法用于保证系统不发生无边界优先级反转 [56]，例如优

先级继承协议（priority-inheritance）和优先级上限协议（priority-ceiling），并且已有很多针对这些协议的分析和验证工作 [57–59]，但是这些工作都是针对某个具体协议，并且没有去验证实际运行的代码和协议之间的一致性。本节首先介绍一个用于验证系统不会发生无限期优先级反转的性质 PIF （priority-inversion-freedom），该性质不依赖于具体协议，并且具有方便验证且易于理解的特点，然后将介绍 μC/OS-II 中优先级反转相关的验证工作。

8.4.1 PIF 的定义

在讨论 PIF 的定义之前，先给出对系统的一些基本假设：（1）系统中的每个任务都有一个初始优先级，表示任务被创建时用户赋予的优先级，$\mathsf{OrgPr}(t, \Sigma)$ 表示系统状态 Σ 下任务 t 的初始优先级；（2）因为很多防止发生无边界优先级反转的协议都会临时改变任务优先级，这里用 $\mathsf{CurPr}(t, \Sigma)$ 表示系统状态 Σ 下任务 t 的当前优先级；（3）系统中同时处于临界区中的任务个数存在上限 N，且每个临界区的执行时间存在上限 T；（4）系统调度采用基于优先级的调度策略，也即所有的就绪态任务的优先级必须小于或等于当前正在运行的任务。

定义 8.4.1 (无优先级反转性质)**.** $\mathsf{PIF}(\Sigma)$ 成立，当且仅当，对于任意的 t，t_c，pr 和 pr_c，如果 $t \neq t_c$，$t_c = \mathsf{CurTask}(\Sigma)$，$pr = \mathsf{OrgPr}(t, \Sigma)$，$pr_c = \mathsf{OrgPr}(t_c, \Sigma)$，$\mathsf{IsWait}(t, \Sigma)$ 和 $\neg\mathsf{IsOwner}(t_c, \Sigma)$ 成立，那么 $pr \preceq pr_c$ 成立。

定义8.4.1表示，如果当前任务 t_c 不占有任何共享资源，那么该任务的原始优先级必须高于或等于所有处于等待状态的其他任务 t 的原始优先级。换句话说，如果当前正在执行的任务没有理由去让其他任务等待（该任务不占有共享资源），那么它必须比其他等待资源的任务更加急迫。$\mathsf{CurTask}(\Sigma)$ 表示获取 Σ 的当前任务号，$\mathsf{IsWait}(t, \Sigma)$ 表示 t 在等待某个共享资源。$\mathsf{IsOwner}(t_c, \Sigma)$ 表示 t_c 占有某个共享资源。该定义本质上是在说如果正在运行的任务占有了某个共享资源，那么其原始优先级可以比其他等待状态的任务的原始优先级低（这种情况下，我们希望尽快运行它，使得它可以尽快释放占有的资源）。如果系统中每个任务最后都能释放占有的共享资源，那么高优先级的等待状态任务将最终获得资源并执行。

下面来简单看下为什么一直满足定义8.4.1的系统能够保证高优先级的任务不会被低优先级任务无限期延迟。假设系统中存在任务 A_1 ,..., A_m, B_1,...,B_n 和 C，它们的原始优先级关系如下：$A_1 > \ldots > A_m > B_1 > \ldots > B_n > C$。$B_i$ 是正在运行的任务，A_m 在等待某个被 C 占有的资源。下面来看定义8.4.1如何保证任务 A_m 不会被低优先级任务 $(B_1, B_2, \ldots, B_n, C)$ 无限期的延迟。讨论当前任务 B_i 可能处于的两种情况：

（1） 如果任务 B_i 不占有任何资源，那么这种情况显然违背了 PIF 的定义，因

为此时当前任务不占有资源但是优先级却比一个等待状态的任务 A_m 的优先级高。

（**2**） 如果 B_i 占有一些资源，根据前述的假设（3），B_i 最终会退出其最外层临界区（因为临界区可能嵌套），然后会出现下列三种情况：

- 如果 B_i 仍然在运行，那么和情况（1）一样违背了 PIF 的定义；
- 如果切换到任务 B_j，那么下列两种情况之一成立：
  - B_j 不占有共享资源，那么和情况（1）一样违背了 PIF 的定义；
  - B_j 占有某些共享资源。然后又回到情况（**2**）一开始的状态，此时 B_j 变为和 B_i 一样的角色，但此时系统中处于临界区中的任务数减 1。通过对优先级低于 A_m 的任务个数 (n+1) 归纳可知，A_m 只会被优先级更低的任务 $(B_1, B_2, \ldots, B_n, C)$ 延迟有限的时间（延迟时间的上限为 (n+1)*T，此时 $(B_1, B_2, \ldots, B_n, C)$ 中每个任务都处于临界区中，且在 A_m 执行之前都被调度执行直到所有被占有的共享资源都被释放）。
- 如果切换到任务 A_k，因为 A_k 优先级高于 A_m，不属于优先级反转的状态。

可以看出如果一个机器模型一直满足定义8.4.1，则可以保证该机器模型不会发生无边界优先级反转。定义8.4.1是一个机器状态上的性质，如果要验证一个机器模型是否一直满足定义8.4.1，只需要对机器操作语义进行归纳证明即可。这比直接验证该机器模型满足无边界优先级反转的定义要容易得多。

8.4.2 μC/OS-II 嵌套使用互斥信号量的 bug

μC/OS-II 中的互斥信号量模块通过实现了一个简化版本的优先级上限协议来防止优先级反转。它主要提供两个 API 函数 `OSMutexPend(S)` 和 `OSMutexPost(S)` 分别用来获取和释放互斥信号量，下面分别简写为 P(S) 和 V(S)。在证明互斥信号量模块的过程中，我们发现在嵌套使用互斥信号量的时候其中存在 bug。下面先简单介绍下 μC/OS-II 中互斥信号量的实现，然后给出一个在嵌套使用互斥信号量时出现 bug 的例子。

μC/OS-II 中每个任务和互斥信号量 S 都存在一个优先级，且系统中每个优先级只允许存在一个任务或信号量。每个信号量 S 中会保存 S 自己的优先级、当前占有 S 的任务号 t，以及 t 在获取 S 时的当前优先级（因为 t 的当前优先级在占有 S 后可能发生变化，所以需要保存占有前的优先级用于在 t 释放 S 时恢复）。

当一个任务 t 执行 P(S) 时，它会执行下面几步：（1）t 的优先级要低于 S 的优先级；(2) 如果此时 S 未被占有，那么在 S 中存入任务号 t 和 t 的当前优先

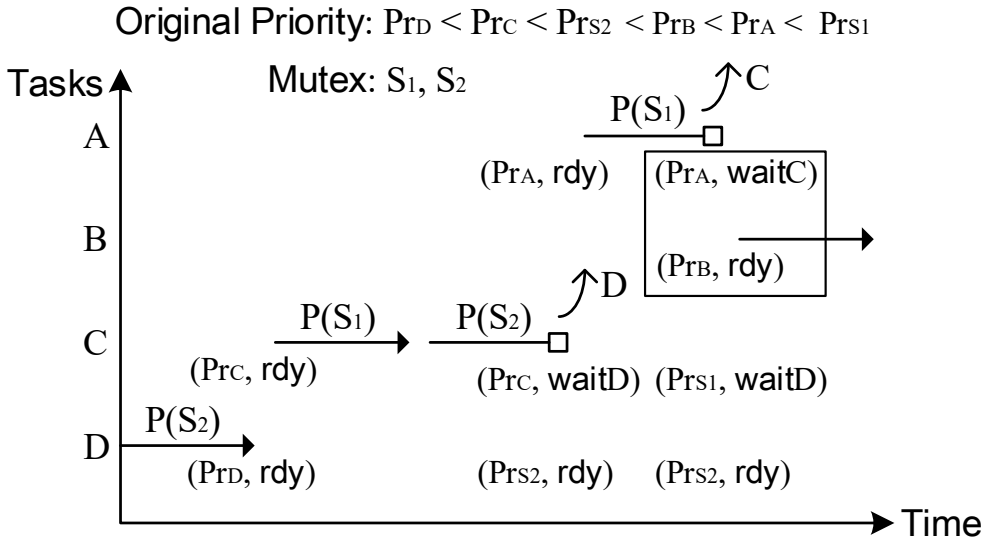

图 8.8　μC/OS-II 中嵌套使用互斥信号量的 bug

级；（3）如果 S 已经被 t′ 占有，此时 t 会被阻塞，如果 t 的当前优先级高于 t′ 的当前优先级，那么将 t′ 的当前优先级被设置为 S 的优先级。

当一个任务 t 执行 V(S) 释放信号量 S 时，它会执行下列几步：（1）t 必须占有 S；（2）如果没有任务正在等待 S，那么将 S 设置为可用状态；（3）如果有任务在等待 S，那么取等待 S 的任务中当前优先级最高的 t′，将 t 的当前优先级恢复为占有 S 之前的值（保存在 S 中），然后将 t′ 设置为 S 的占有者（在 S 中保存 t′ 及其当前优先级）。

图8.8展现了一个存在信号量嵌套使用时出现的 bug。图中包含 4 个任务 A、B、C、D 以及两个互斥信号量 S_1、S_2。用户设置的原始优先级满足下述关系：$Pr_D < Pr_C < Pr_{S_2} < Pr_B < Pr_A < Pr_{S_1}$。假设一开始任务 A、B、C 被延迟并且正在等待时钟中断唤醒，D 是唯一正在运行的任务。首先任务 D 获取信号量 S_2，然后 C 被唤醒并抢占了 D。C 接着获取了信号量 S_1, 然后尝试获取 S_2，但因为 S_2 已经被 D 占有，所以 C 会被阻塞，因为 $Pr_D < Pr_C$，所以根据协议将 D 的优先级提升到 Pr_{S2}，并且 D 被调度执行。然后任务 A 被唤醒，因为 A 的优先级最高，所以 A 抢占了 D。此时 A 尝试获取被 C 占有的 S_1，所以 A 被阻塞，并且 C 的优先级被提升到 Pr_{S_1}。最后任务 B 被唤醒并执行，但此时已经发生了无边界优先级反转，因为任务 A 可能无限期等待任务 B。并且从图中方框内的部分可以看出，此时定义8.4.1被破坏，因为任务 B 不占有任何资源，并且原始优先级低于 A。图中括号内的二元组表示该任务当前优先级和状态。

$$
\begin{array}{lll}
\mathsf{CurTask}(\Sigma) & \stackrel{\text{def}}{=} & \Sigma(\mathtt{ctid}) \qquad \mathsf{CurPr}(t,\Sigma) \stackrel{\text{def}}{=} \begin{cases} pr & \text{if } \Sigma(\mathtt{tcbls})(t) = (pr,\_) \\ \bot & \text{otherwise} \end{cases} \\
\mathsf{Own}(t,eid,\Sigma) & \stackrel{\text{def}}{=} & t \in dom(\Sigma(\mathtt{tcbls})) \wedge \Sigma(\mathtt{ecbls})(eid) = (\_,(t,\_)) \\
\mathsf{IsOwner}(t,\Sigma) & \stackrel{\text{def}}{=} & \exists eid.\mathsf{Own}(t,eid,\Sigma) \\
\mathsf{OrgPr}(t,\Sigma) & \stackrel{\text{def}}{=} & \begin{cases} pr & \text{if } \exists eid.\mathsf{Own}(t,eid,\Sigma) \wedge \Sigma(\mathtt{ecbls})(eid) = (\_,(\_,pr)) \\ pr & \text{if } (\neg\mathsf{IsOwner}(t,\Sigma)) \wedge \Sigma(\mathtt{tcbls})(t) = (pr,\_) \\ \bot & \text{otherwise} \end{cases} \\
\mathsf{Wait}(t,eid,\Sigma) & \stackrel{\text{def}}{=} & \Sigma(\mathtt{tcbls})(t) = (\_,\_,\mathsf{wait}(\mathsf{mtx}(eid),\_)) \\
t \xrightarrow{\Sigma} t' & \stackrel{\text{def}}{=} & \exists eid.\mathsf{Wait}(t,eid,\Sigma) \wedge \mathsf{Own}(t',eid,\Sigma) \wedge t \neq t' \\
\mathsf{IsWait}(t,\Sigma) & \stackrel{\text{def}}{=} & \exists t'.t \xrightarrow{\Sigma} t' \qquad \mathsf{IsRdy}(t,\Sigma) \stackrel{\text{def}}{=} \Sigma(\mathtt{tcbls})(t) = (\_,\mathsf{rdy}) \\
\mathsf{Init}(\Sigma) & \stackrel{\text{def}}{=} & \forall t \in dom(\Sigma(\mathtt{tcbls})).\neg\mathsf{IsWait}(t,\Sigma)
\end{array}
$$

图 8.9　PIF相关的辅助定义

8.4.3　互斥信号量满足 UPIF 的证明

上节已经介绍过在嵌套使用情况下存在的 bug，所以只证明在非嵌套使用的情况下 μC/OS-II 中互斥信号量模块满足 PIF。图8.9中给出了 μC/OS-II 互斥信号量模块相关的一些辅助定义。定理8.4.1是证明的 μC/OS-II 的互斥信号量模块满足的性质。

定理 8.4.1 (互斥信号量模块满足PIF)**.** 对于任意的 A、T、Σ、T'、Δ' 和 Σ'，如果 $\mathsf{Init}(\Sigma)$，$(A,\mathbb{O}_{\mu\text{C/OS-II}}) \vdash (T,\Delta,\Sigma) =\!\!\Rightarrow^{*}_{H} (T',\Delta',\Sigma')$，$\mathsf{NoNCR}(A,\Sigma,T,\Delta)$ 和 $\mathsf{SchedProp}(\Sigma')$ 成立，那么 $\mathrm{PIF}(\Sigma')$ 成立。

定理8.4.1表示，对于任意的应用程序 A，用户初始状态 Δ 和抽象内核状态初始状态 Σ，如果初始时每个任务都没有等待互斥信号量（$\mathsf{Init}(\Sigma)$），程序中不嵌套使用互斥信号量（$\mathsf{NoNCR}(A,\Sigma,T,\Delta)$），那么对于执行任意步之后的抽象内核状态 Σ'，如果 Σ' 满足基于优先级调度的要求（$\mathsf{SchedProp}(\Sigma')$），那么 Σ' 满足 PIF。

为了定义 $\mathsf{NoNCR}(A,\Sigma,T,\Delta)$，需要先定义出如果嵌套使用互斥信号量，抽象内核状态 Σ 会满足的性质：

$$
\begin{array}{rl}
\mathsf{Nest}(\Sigma) \stackrel{\text{def}}{=} & \exists t, eid.(t \in dom(\Sigma(\mathtt{tcbls})) \wedge \mathsf{Own}(t,eid,\Sigma)) \wedge \\
& (\mathsf{IsWait}(t,\Sigma) \vee \exists eid'.eid' \neq eid \wedge \mathsf{Own}(t,eid',\Sigma))
\end{array}
$$

Nest 表示，如果嵌套使用互斥信号量，那么系统中会存在一个任务 t 和一个互斥信号量 eid，t 在占有 eid 之后又尝试去占有另一个信号量 eid'（$\mathsf{IsWait}(t,\Sigma)$ 表示占有 eid' 失败）。

$\mathsf{NoNCR}(A,\Sigma,T,\Delta)$ 定义如下，它表示系统执行任意多步都不会出现 Nest 的情况：

定义 8.4.2 (无嵌套使用互斥信号量). $\mathsf{NoNCR}(A, \Sigma, T, \Delta)$ 成立，当且仅当，对于任意的 Σ'、T' 和 Δ'，如果 $(A, \mathbb{O}_{\mu C/OS\text{-}II}) \vdash (T, \Delta, \Sigma) =\!\!\Rightarrow^{*} (T', \Delta', \Sigma')$ 成立，那么 $\neg \mathsf{nest}(\Sigma')$.

定义8.4.3给出了对系统调度策略的要求 $\mathsf{SchedProp}(\Sigma)$ 的定义，其主要包含两个要求：（1）当前正在执行的任务一定是一个就绪态的任务；（2）当前任务是所有就绪态任务里面优先级最高的。

定义 8.4.3 (调度性质). $\mathsf{SchedProp}(\Sigma)$ 成立，当且仅当，对于任意的 t_c 和 pr_c，如果 $t_c = \mathsf{CurTask}(\Sigma)$ 和 $pr_c = \mathsf{CurPr}(t_c, \Sigma)$ 成立，那么 $\mathsf{IsRdy}(t_c, \Sigma)$ 成立，并且对于任意的 t 和 pr，如果 $t \neq t_c$，$pr = \mathsf{CurPr}(t, \Sigma)$ 和 $\mathsf{IsRdy}(t, \Sigma)$ 成立，那么 $pr \preceq pr_c$。

定理8.4.1的证明在 Coq 代码 [3] 的“/CertiOS/certiucos/spec/PIF.v”文件中，这里不再具体介绍。

第九章　总结

嵌入式实时操作系统内核被广泛用于各种嵌入式设备中，它通过任务抢占和多级中断来保证任务实时性，同时这也导致内核高度并发和复杂。本文设计并实现了第一个支持抢占和多级中断的操作系统内核验证框架，并应用该验证框架首次成功验证了商业实时操作系统内核 μC/OS-II 的核心功能模块。

操作系统内核的功能正确性被定义成为底层 API 具体实现和高层抽象规约之间上下文精化关系，即对于任意的应用程序，其在底层机器模型上执行产生的行为是高层抽象机器模型上执行产生的行为的子集。上述正确性定义是操作系统内核验证框架的验证目标，整个验证框架由三个部分构成：

- 操作系统内核建模：（1）对实现操作系统内核的底层 C 语言内嵌汇编建模；（2）对高层的规范语言建模。底层模型中 C 语言和汇编语言对机器的抽象层次不一样，如果要在统一的机器状态上描述 C 语言和汇编的语义会非常复杂，因此本文通过对 C 语言机器状态扩展，并将汇编代码抽象为原语的方式来完成底层内核语言的建模。高层模型中规范语言在抽象了很多不必要的实现细节的同时，可以准确的描述系统 API 中调度相关的行为，这对于在高层验证系统层面的各种调度相关的性质是至关重要的。应用程序代码只能通过调用系统 API 来访问内核资源，内核代码不会访问用户资源。在底层机器模型上，应用程序调用系统 API 会去执行实现该 API 的 C 语言内嵌汇编代码；而在高层机器模型上，应用程序调用系统 API 会去执行该 API 的规范代码。

- 并发精化程序逻辑 CSL-R：借鉴了并发分离逻辑 [43] 的占有权转移思想和 [34] 中关于中断处理程序的证明，并将这些工作扩展到多级中断和抢占式内核的验证中。为了保证该逻辑的正确性，参考了 RGSim [7, 8] 对于开放环境下精化关系的证明，通过程序模拟技术完成了逻辑正确性的证明。

- 自动证明策略：为了提高代码证明的效率，开发了一套自动证明策略，该自动证明策略借鉴了 [51] 中证明策略的设计思想，并将其扩展到基于关系型并发精化程序逻辑的代码验证中。扩展主要包含三个方面：（1）关系型分离逻辑断言蕴含关系的自动推理；（2）基于精化逻辑推理规则的验证条件自动生成；（3）特定领域数学性质的自动证明, 如：32 位机器整数。

本文成功应用上述验证框架验证了商业化实时嵌入式操作系统内核 μC/OS-II 的核心功能模块。这说明了验证框架具有很强的实用性。此外，本文在高层模型上证明了 μC/OS-II 中互斥锁在不被嵌套使用的情况下满足无优先级反转的性质（PIF），无优先级反转是实时抢占式操作系统中非常重要的系统性质，对 PIF 的证明也从侧面说明了高层规范语言的表达力。

进一步工作

- *规范语言支持对用户参数的约束* 由于操作系统正确性定义能够保证任意的应用程序调用底层具体 API 的实现所产生的行为不会比调用对应的高层抽象 API 规约产生的行为多，这个性质虽然能够为操作系统内核的正确性提供一个很强的保证，但是在实际的操作系统内核实现中，为了平衡内核性能，很多时候系统 API 在被应用程序调用时对应用程序是有一定的约束的。由于对任意应用程序的假设，导致目前这种情况不得不通过额外添加约束检查的代码来完成验证。作为下一步工作，可以通过扩展高层规范语言，引入一种叫“不满足调用系统 API 的要求”的错误，这种错误是用户没有正确调用系统 API 导致的，不应该看作是内核代码的错误。同时操作系统正确性定义可以弱化为，对所有满足系统 API 调用要求的应用程序，底层和高层之间满足精化关系。

- *支持内核代码访问用户状态* 应用程序可能会将指针作为参数传入系统 API，使得系统 API 可以通过该指针访问应用程序的资源。这种情况验证框架目前并不支持，下一步工作希望可以通过引入应用程序状态和内核程序状态之间的占有权转移来完成这类函数的验证。

- *支持更多的 C 语言特性* 验证框架目前仅支持一个 C 语言子集，虽然可以用来实现 μC/OS-II, 但是它并没有包含 C 语言的其他一些有用的特性（例如：共用体，break 语句等等），这些特性可能在新的操作系统实现中会被使用。下一步工作可以进一步扩展该 C 语言子集，使得验证框架可以用于更多的操作系统内核的验证。

- *保证编译过程的正确性* 构建一个可信的操作系统内核不仅需要验证源码和规范之间的一致性，还需要保证源码和目标机器码之间的一致性。下一步希望可以通过构建一个可信编译器来保证本文底层内核代码和真实目标机器码之间的一致性，该可信编译器需要支持并发 C 代码的编译，同时还应该支持对扩展的汇编原语的编译。

参考文献

[1] The Real-Time Kernel: μC/OS-II. http://micrium.com/rtos/ucosii/overview.
[2] The Coq Development Team: The Coq Proof Assistant. http://coq.inria.fr.
[3] Xu F, Fu M, Feng X, et al. A Practical Verification Framework for Preemptive OS Kernels (Coq Implementations), May, 2016. http://staff.ustc.edu.cn/~fuming/research/certiucos.
[4] Home page of Vxworks. www.windriver.com/products/vxworks.
[5] Stuxnet. From Wikipedia, 2014. http://en.wikipedia.org/wiki/Stuxnet.
[6] Heartbleed. From Wikipedia, 2014. https://en.wikipedia.org/wiki/Heartbleed.
[7] Liang H, Feng X, Fu M. A rely-guarantee-based simulation for verifying concurrent program transformations. Proceedings of POPL, 2012. 455–468.
[8] Liang H, Feng X. Modular Verification of Linearizability with Non-Fixed Linearization Points. Proceedings of PLDI, 2013. 459–470.
[9] Liang H, Feng X, Shao Z. Compositional Verification of Termination-preserving Refinement of Concurrent Programs. Proceedings of CSL-LICS, 2014. 65:1–65:10.
[10] Turon A, Thamsborg J, Ahmed A, et al. Logical relations for fine-grained concurrency. Proceedings of POPL, 2013. 343–356.
[11] Turon A, Dreyer D, Birkedal L. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. Proceedings of ICFP, 2013. 377–390.
[12] Liang H, Feng X. A program logic for concurrent objects under fair scheduling. Proceedings of ACM SIGPLAN Notices, volume 51. ACM, 2016. 385–399.
[13] Klein G, Andronick J, Elphinstone K, et al. Comprehensive formal verification of an OS microkernel. ACM Trans. Comput. Syst., 2014, 32(1):2.
[14] Klein G, Huuck R, Schlich B. Operating System Verification. J. Autom. Reasoning, 2009, 42(2-4):123–124.
[15] Hoffmann J, Ramananandro T, Shao Z. End-to-End Verification of Stack-Space Bounds for C Programs. Proceedings of PLDI, 2014. 270–281.
[16] Gu R, Koenig J, Ramananandro T, et al. Deep Specifications and Certified Abstraction Layers. Proceedings of POPL, 2015. 595–608.
[17] Chen H, Wu X N, Shao Z, et al. Toward compositional verification of interruptible OS kernels and device drivers. Proceedings of Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2016. 431–447.
[18] The eChronos RTOS. https://github.com/echronos/.
[19] Andronick J, Lewis C, Matichuk D, et al. Proof of OS scheduling behavior in the presence of interrupt-induced concurrency. Proceedings of International Conference on Interactive Theorem Proving. Springer, 2016. 52–68.
[20] Hohmuth M, Tews H. The VFiasco approach for a verified operating system. Proceedings of In Proc. the 2nd ECOOP Workshop on Programming Languages and Operating Systems, Glasgow, UK, 2005.
[21] Rieden, In T, Tsyban A. CVM–a verified framework for microkernel programmers. Electronic Notes in Theoretical Computer Science, 2008, 217:151–168.
[22] Daum M, Schirmer N, Schmidt M. Implementation Correctness of a Real-Time Operating System. Proceedings of SEFM, volume 9, 2009. 23–32.
[23] Baumann C, Bormer T. Verifying the PikeOS microkernel: First results in the Verisoft XT Avionics project. Proceedings of Doctoral Symposium on Systems Software Verification (DS SSV’09) Real Software, Real Problems, Real Solutions, 2009. 20.
[24] The Verisoft XT Project, 2007. http://www.verisoftxt.de.
[25] Hunt G C, Larus J R. Singularity: rethinking the software stack. ACM SIGOPS Operating Systems Review, 2007, 41(2):37–49.
[26] Cohen E, Dahlweid M, Hillebrand M, et al. VCC: A Practical System for Verifying Concurrent C. Proceedings of TPHOLs, 2009. 23–42.
[27] Lahiri S K, Qadeer S, Rakamarić Z. Static and precise detection of concurrency errors in systems code using SMT solvers. Proceedings of International Conference on Computer Aided Verification. Springer, 2009. 509–524.
[28] Chris H, Jean Y. Automated verification of a type-safe operating system, December 25, 2012. US Patent 8,341,602.

[29] Klein G, Elphinstone K, Heiser G, et al. seL4: Formal Verification of an OS Kernel. Proceedings of SOSP, 2009. 207–220.
[30] Sewell T A L, Myreen M O, Klein G. Translation validation for a verified OS kernel. Proceedings of ACM SIGPLAN Notices, volume 48. ACM, 2013. 471–482.
[31] Feng X, Shao Z. Modular verification of concurrent assembly code with dynamic thread creation and termination. ACM SIGPLAN Notices, 2005, 40(9):254–267.
[32] Feng X, Shao Z, Vaynberg A, et al. Modular verification of assembly code with stack-based control abstractions. Proceedings of ACM SIGPLAN Notices, volume 41. ACM, 2006. 401–414.
[33] Cai H, Shao Z, Vaynberg A. Certified self-modifying code. ACM SIGPLAN Notices, 2007, 42(6):66–77.
[34] Feng X, Shao Z, Dong Y, et al. Certifying Low-level Programs with Hardware Interrupts and Preemptive Threads. Proceedings of PLDI, 2008. 170–182.
[35] Leroy X. Formal verification of a realistic compiler. Commun. ACM, 2009, 52(7):107–115.
[36] Alkassar E, Paul W, Starostin A, et al. Pervasive Verification of an OS Microkernel: Inline Assembly, Memory Consumption, Concurrent Devices. Proceedings of VSTTE, 2010. 71–85.
[37] Yang J, Hawblitzel C. Safe to the last instruction: automated verification of a type-safe operating system. Proceedings of PLDI, 2010. 99–110.
[38] Moura L M, Bjørner N. Z3: An Efficient SMT Solver. Proceedings of TACAS, 2008. 337–340.
[39] 朱允敏, 张丽伟, 王生原, et al. 面向多核处理器的低级并行程序验证. 电子学报, 2009, 37(S1):1–6.
[40] 肖增良, 何镕, 康立山. 并行程序验证的调度策略. 计算机工程与应用, 2009, 45(11):39–41.
[41] Jones C B. Tentative steps toward a development method for interfering programs. ACM Transactions on Programming Languages and Systems (TOPLAS), 1983, 5(4):596–619.
[42] Feng X. Local rely-guarantee reasoning. Proceedings of POPL, 2009. 315–327.
[43] O'Hearn P W. Resources, Concurrency and Local Reasoning. Proceedings of CONCUR, 2004. 49–67.
[44] Labrosse J J. MicroC/OS-II: The Real Time Kernel. CMP Books, 2002.
[45] Reynolds J C. Separation Logic: A Logic for Shared Mutable Data Structures. Proceedings of LICS 2002, 2002. 55–74.
[46] Stewart G, Beringer L, Cuellar S, et al. Compositional CompCert. Proceedings of POPL, 2015. 275–287.
[47] Klein G, Andronick J, Elphinstone K, et al. seL4: Formal Verification of an Operating-system Kernel. Commun. ACM, 2010, 53(6):107–115.
[48] Sevcík J, Vafeiadis V, Nardelli F Z, et al. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. J. ACM, 2013, 60(3):22.
[49] Liang H, Feng X. Modular verification of linearizability with non-fixed linearization points. Technical report, USTC, March, 2013. http://kyhcs.ustcsz.edu.cn/relconcur/lin.
[50] 王勇朝, 李兆鹏, 冯新宇. 一个 C 语言子集上的程序逻辑. 小型微型计算机系统, 2014, 35(6):1258–1264.
[51] Cao J, Fu M, Feng X. Practical Tactics for Verifying C Programs in Coq. Proceedings of CPP, 2015. 97–108.
[52] McCreight A. Practical Tactics for Separation Logic. Proceedings of TPHOLs, 2009. 343–358.
[53] ClightTSO Syntax and Its Semantics. http://www.cl.cam.ac.uk/~pes20/CompCertTSO/doc/clightTSO.pdf.
[54] Boyland J. Checking interference with fractional permissions. Proceedings of International Static Analysis Symposium. Springer, 2003. 55–72.
[55] Appel A W. Tactics for Separation Logic, 2006. http://www.cs.princeton.edu/~appel/papers/septacs.pdf.
[56] Sha L, Rajkumar R, Lehoczky J P. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Transactions on Computers, 1990, 39:1175–1185.
[57] Zhang X, Urban C, Wu C. Priority Inheritance Protocol Proved Correct. Proceedings of ITP'12, volume 7406 of *Lecture Notes in Computer Science*, 2012. 217–232.
[58] Babaoglu O, Marzullo K, Schneider F B. A Formalization of Priority Inversion. Real-time Systems, 1993, 5:285–303.
[59] Dutertre B. The Priority Ceiling Protocol: Formalization and Analysis Using PVS. Proceedings of Proc. RTSS'00, 2000. 151–160.

附录 A　C 语言子集操作语义

附录中给出了 C 语言子集的操作语义。图A.1中给出了完整的程序后继的定义。其中表达式运算操作和语句执行操作将“$\eta \vdash (c, \kappa_e, m) \overset{\hat{\vartheta}}{\longmapsto} (c', \kappa_e', m')$”分为两种情况，分别是表达式计算操作“$(c, \kappa_e, m) -e\to (c', \kappa_e', m')$”和语句执行操作“$\eta \vdash (c, \kappa_s, m) -\overset{\hat{\vartheta}}{s}\to (c', \kappa_s', m')$”。注意只有当当前表达式计算完成后（$\kappa_e = \circ$）才能执行语句执行操作。表达式计算操作定义在图A.3中，语句执行操作定义在图A.4和图A.5 中。操作语义相关的辅助定义在图A.6、A.7、A.8和A.9中。注意这里 bop_eval 和 uop_eval，分别表示二元运算和一元运算，由于情况较多这里没有给出具体定义，感兴趣的读者可以参考 Coq 实现 [3]。

$$C \overset{\text{def}}{=} (c, K) \qquad K \overset{\text{def}}{=} (\kappa_e, \kappa_s)$$

$$c \overset{\text{def}}{=} e \mid sv \mid \mid \mathbf{s} \mid \textbf{\textit{fexec}}(f, \mathcal{T}, \bar{v}) \mid \textbf{\textit{alloc}}(f, \bar{v}, \mathcal{D}) \mid \textbf{\textit{free}}(\mathcal{O}, \hat{v}) \mid \textbf{\textit{skip}}$$

$$\begin{aligned}\kappa_e \overset{\text{def}}{=}\ & \circ \mid (\mathsf{offset}\ i) \cdot \kappa_e \mid (\mathsf{deref}\ \tau) \cdot \kappa_e \mid (\_[e]\ \tau) \cdot \kappa_e \mid (\mathsf{uop}\ \mathtt{uop}\ \tau) \cdot \kappa_e \mid \\ & (\mathsf{bop}\ \mathtt{bop}\ e_2\ \tau_1\ \tau_2) \cdot \kappa_e \mid (v[\_]\ \tau) \cdot K \mid (\mathsf{bopv}\ \mathtt{bop}\ v\ \tau_1\ \tau_2) \cdot \kappa_e\end{aligned}$$

$$\begin{aligned}\kappa_s \overset{\text{def}}{=}\ & \bullet \mid s \cdot \kappa_s \mid (\_ = v\ \tau) \cdot \kappa_s \mid (\mathsf{if}\ \_\ s_1\ s_2) \cdot \kappa_s \mid (\mathsf{while}\ \_\ e\ s) \cdot \kappa_s \mid (f(\bar{v}, \mathcal{T}, \bar{e})) \cdot \kappa_s \mid \\ & (\mathsf{call}\ f\ E) \cdot \kappa_s \mid (e = \_\ \tau) \cdot \kappa_s\end{aligned}$$

图 A.1　程序后继

$$\boxed{\eta \vdash (c, \kappa_e, m) \overset{\hat{\vartheta}}{\longmapsto} (c', \kappa_e', m')}$$

$$\frac{(c, \kappa_e, m) -e\to (c', \kappa_e', m')}{\eta \vdash ((c, (\kappa_e, \kappa_s)), m) \longmapsto ((c', (\kappa_e', \kappa_s)), m')}$$

$$\frac{\kappa_e = \circ \quad \eta \vdash (c, \kappa_s, m) -\overset{\hat{\vartheta}}{s}\to (c', \kappa_s', m')}{\eta \vdash ((c, (\kappa_e, \kappa_s)), m) \overset{\hat{\vartheta}}{\longmapsto} ((c', (\kappa_e, \kappa_s')), m')}$$

图 A.2　C 语言操作语义

$\boxed{(c, \kappa_e, m) \text{ —e→} (c', \kappa_e', m')}$

$$\frac{}{(\texttt{NULL}, \kappa_e, m) \text{ —e→} (\texttt{Vnull}, \kappa_e, m)} \qquad \frac{[\![x]\!]^r_m = v}{(x, \kappa_e, m) \text{ —e→} (v, \kappa_e, m)}$$

$$\frac{[\![\&x]\!]^r_m = v}{(\&x, \kappa_e, m) \text{ —e→} (v, \kappa_e, m)} \qquad \frac{[\![*e]\!]^t_m = \tau}{(*e, \kappa_e, m) \text{ —e→} (e, (\mathsf{deref}\ \tau) \cdot \kappa_e, m)}$$

$$\frac{[\![e]\!]^t_m = \texttt{Tstruct}(id', \mathcal{D}) \quad \mathsf{field\_off}(id, \mathcal{D}) = i}{(\&(e.id), \kappa_e, m) \text{ —e→} (\&e, (\mathsf{offset}\ i) \cdot \kappa_e, m)}$$

$$\frac{[\![e_1]\!]^t_m = \texttt{Tarray}(\tau, n)}{(\&(e_1[e_2]), \kappa_e, m) \text{ —e→} (\&e_1, (\_[e]\ \tau) \cdot \kappa_e, m)}$$

$$\frac{[\![e_1]\!]^t_m = \texttt{Tptr}(\tau)}{(\&(e_1[e_2]), \kappa_e, m) \text{ —e→} (e_1, (\_[e]\ \tau) \cdot \kappa_e, m)}$$

$$\frac{[\![e]\!]^t_m = \texttt{Tstruct}(id', \mathcal{D}) \quad \mathsf{field\_type}((, id), \mathcal{D}) = \tau'}{(e.id, \kappa_e, m) \text{ —e→} (\&(e.id), (\mathsf{deref}\ \tau) \cdot \kappa_e, m)}$$

$$\frac{[\![e_1]\!]^t_m = \texttt{Tarray}(\tau, n)}{(e_1[e_2], \kappa_e, m) \text{ —e→} (\&(e_1[e_2]), (\mathsf{deref}\ \tau) \cdot \kappa_e, , m)}$$

$$\frac{[\![e]\!]^t_m = \tau}{(\texttt{uop}\ e, \kappa_e, m) \text{ —e→} (e, (\mathsf{uop}\ \texttt{uop}\ \tau) \cdot \kappa_e, m)} \qquad \frac{\tau = \texttt{Tptr}(\tau') \quad [\![e]\!]^t_m = \texttt{Tptr}(\tau'')}{((\tau)e, \kappa_e, m) \text{ —e→} (e, \kappa_e, m)}$$

$$\frac{[\![e_1]\!]^t_m = \tau_1 \quad [\![e_2]\!]^t_m = \tau_2}{(e_1\ \texttt{bop}\ e_2, \kappa_e, m) \text{ —e→} (e_1, (\mathsf{bop}\ \texttt{bop}\ e_2\ \tau_1\ \tau_2) \cdot \kappa_e, m)}$$

$$\frac{m = (G, E, M) \quad \mathsf{load}(M, \texttt{Vptr}(b, i), \tau) = v}{([\texttt{Vptr}(b, i)], (\mathsf{deref}\ \tau) \cdot \kappa_e, m) \text{ —e→} ([v], \kappa_e, m)}$$

$$\frac{}{([\texttt{Vptr}(b, i)], (\mathsf{offset}\ i') \cdot \kappa_e, m) \text{ —e→} ([\texttt{Vptr}(b, i+i')], \kappa_e, m)}$$

$$\frac{}{([v], (\_[e]\ \tau) \cdot \kappa_e, m) \text{ —e→} (e, (v[\_]\ \tau) \cdot \kappa_e, m)}$$

$$\frac{}{([v], (\texttt{Vptr}(b, i)[\_]\ \tau) \cdot \kappa_e, m) \text{ —e→} ([\texttt{Vptr}(b, i+|\tau| * v)], \kappa_e, m)}$$

$$\frac{\mathsf{uop\_eval}(\texttt{uop}, v) = v'}{([v], (\mathsf{uop}\ \texttt{uop}\ \tau) \cdot \kappa_e, m) \text{ —e→} ([v'], \kappa_e, m)}$$

$$\frac{}{([v], (\mathsf{bop}\ \texttt{bop}\ e\ \tau_1\ \tau_2) \cdot \kappa_e, m) \text{ —e→} (e, (\mathsf{bopv}\ \texttt{bop}\ v\ \tau_1\ \tau_2) \cdot \kappa_e, m)}$$

$$\frac{\mathsf{bop\_eval}(\texttt{bop}, v, v', \tau_1, \tau_2) = v''}{([v], (\mathsf{bopv}\ \texttt{bop}\ v'\ \tau_1\ \tau_2) \cdot \kappa_e, m) \text{ —e→} ([v''], \kappa_e, m)}$$

图 A.3　表达式运算操作语义

$$\boxed{\eta \vdash (c, \kappa_s, m) \xrightarrow{\hat{\vartheta}}_s (c', \kappa_s', m')}$$

$$\frac{[\![e_1]\!]^t_m = \tau_1 \quad [\![e_2]\!]^t_m = \tau_2 \quad \tau_1 \propto \tau_2}{\eta \vdash (e_1 = e_2, \kappa_s, m) \xrightarrow{}_s (e_2, (e_1 =\_\ \tau_1) \cdot \kappa_s, m)}$$

$$\frac{}{\eta \vdash (s_1; s_2, \kappa_s, m) \xrightarrow{}_s (s_1, s_2 \cdot \kappa_s, m)}$$

$$\frac{}{\eta \vdash (\textbf{if}\ (e)\ \textbf{then}\ s_1\ \textbf{else}\ s_2, \kappa_s, m) \xrightarrow{}_s (e, (\mathsf{if}\ \_\ s_1\ s_2) \cdot \kappa_s, m)}$$

$$\frac{}{\eta \vdash (\textbf{while}\ (e)\ s, \kappa_s, m) \xrightarrow{}_s (e, (\mathsf{while}\ \_\ e\ s) \cdot \kappa_s, m)}$$

$$\frac{[\![e]\!]^r_m = v}{\eta \vdash (\textbf{print}\ e, s \cdot \kappa_s, m) \xrightarrow{v}_s (s, \kappa_s, m)} \qquad \frac{}{\eta \vdash (\textbf{\textit{skip}}, s \cdot \kappa_s, m) \xrightarrow{}_s (s, \kappa_s, m)}$$

$$\frac{}{\eta \vdash (\mathtt{f(nil)}, \kappa_s, m) \xrightarrow{}_s (\textbf{\textit{fexec}}(\mathtt{f}, \mathtt{nil}, \mathtt{nil}), \kappa_s, m)}$$

$$\frac{[\![e]\!]^t_m = \tau}{\eta \vdash (\mathtt{f}(e :: \bar{e}), \kappa_s, m) \xrightarrow{}_s (e, \mathtt{f}(\mathtt{nil}, \tau :: \mathtt{nil}, \bar{e}) \cdot \kappa_s, m)}$$

$$\frac{m = (G, E, M) \quad \mathsf{alloc}(x, E, M, E', M', \tau) \quad m' = (G, E', M')}{\eta \vdash (\textbf{\textit{alloc}}(\mathtt{f}, \mathtt{nil}, (x, \tau) :: \mathcal{D}), \kappa_s, m) \xrightarrow{}_s (\textbf{\textit{alloc}}(\mathtt{f}, \mathtt{nil}, \mathcal{D}), \kappa_s, m')}$$

$$\frac{\begin{array}{c} m = (G, E, M) \quad \mathsf{alloc}(x, E, M, E', M', \tau) \quad E'(x) = (b, \tau) \\ M'\{(b, 0) \overset{\tau}{\rightsquigarrow} v\} = M'' \quad m' = (G, E', M'') \end{array}}{\eta \vdash (\textbf{\textit{alloc}}(\mathtt{f}, v :: \bar{v}, (x, \tau) :: \mathcal{D}), \kappa_s, m) \xrightarrow{}_s (\textbf{\textit{alloc}}(\mathtt{f}, \bar{v}, \mathcal{D}), \kappa_s, m')}$$

$$\frac{}{\eta \vdash (\textbf{\textit{alloc}}(\mathtt{f}, \mathtt{nil}, \mathtt{nil}), (\mathsf{call}\ \mathtt{f}\ E) \cdot \kappa_s, m) \xrightarrow{}_s (s, \kappa_s, m)}$$

$$\frac{[\![e]\!]^t_m = \tau}{\eta \vdash (e = \mathtt{f}(\bar{e}), \kappa_s, m) \xrightarrow{}_s (\mathtt{f}(\bar{e}), (e =\_\ \tau) \cdot \kappa_s, m)}$$

$$\frac{m = (G, E, M) \quad \mathcal{O} = \mathsf{getaddr}(E)}{\eta \vdash (\textbf{return}, \kappa_s, m) \xrightarrow{}_s (\textbf{\textit{free}}(\mathcal{O}, \bot), \kappa_s, m)}$$

$$\frac{m = (G, E, M) \quad \mathcal{O} = \mathsf{getaddr}(E)}{\eta \vdash (v, (\mathsf{return}\ \_) \cdot \kappa_s, m) \xrightarrow{}_s (\textbf{\textit{free}}(\mathcal{O}, v), \kappa_s, m)}$$

$$\frac{}{\eta \vdash (\textbf{return}\ e, \kappa_s, m) \xrightarrow{}_s (e, (\mathsf{return}\ \_) \cdot \kappa_s, m)}$$

$$\frac{m = (G, E, M) \quad \mathsf{free}(\tau, b, M) = M' \quad m = (G, E, M')}{\eta \vdash (\textbf{\textit{free}}((b, \tau) :: \mathcal{O}, \hat{v}), \kappa_s, m) \xrightarrow{}_s (\textbf{\textit{free}}(\mathcal{O}, \hat{v}), \kappa_s, m')}$$

$$\frac{\lfloor \kappa_s \rfloor_c = (\mathsf{call}\ \mathtt{f}\ E') \cdot \kappa_s' \quad m = (G, E, M)}{\eta \vdash (\textbf{\textit{free}}(\mathtt{nil}, \hat{v}), \kappa_s, m) \xrightarrow{}_s ([\hat{v}], \kappa_s', (G, E', M))}$$

图 A.4　语句执行操作语义（I）

$$\boxed{\eta \vdash (c, \kappa_s, m) \xrightarrow[s]{\hat{\vartheta}} (c', \kappa_s', m')}$$

$$\frac{}{\eta \vdash ([v], (e = \_\ \tau) \cdot \kappa_s, m) \xrightarrow{s} (\&e, (\_ = v\ \tau) \cdot \kappa_s, m)}$$

$$\frac{m = (G, E, M) \quad \mathtt{Vptr}(b, i)\{v \overset{M}{\rightsquigarrow} \tau\} = M' \quad m = (G, E, M')}{\eta \vdash (\mathtt{Vptr}(b, i), (\_ = v\ \tau) \cdot \kappa_s, m) \xrightarrow{s} (\textbf{\textit{skip}}, \kappa_s, m')}$$

$$\frac{m = (G, E, M) \quad \eta(f) = (\tau, \mathcal{D}_1, \mathcal{D}_2, s) \quad m' = (G, \emptyset, M)}{\eta \vdash (\textbf{\textit{fexec}}(f, \bar{v}, \mathcal{T}), \kappa_s, m) \xrightarrow{s} (\textbf{\textit{alloc}}(f, \bar{v}, \mathsf{rev}(\mathcal{D}_1) +\!+ \mathcal{D}_2), (\mathsf{call}\ f\ E) \cdot \kappa_s, m')}$$

$$\frac{\mathsf{istrue}(v) = 1}{\eta \vdash (v, (\mathsf{if}\ \_\ s_1\ s_2) \cdot \kappa_s, m) \xrightarrow{s} (s_1, \kappa_s)}$$

$$\frac{\mathsf{istrue}(v) = 0}{\eta \vdash (v, (\mathsf{if}\ \_\ s_1\ s_2) \cdot \kappa_s, m) \xrightarrow{s} (s_2, \kappa_s)}$$

$$\frac{\mathsf{istrue}(v) = 1}{\eta \vdash (v, (\mathsf{while}\ \_\ e\ s) \cdot \kappa_s, m) \xrightarrow{s} (s, (\textbf{while}\ (e)\ s) \cdot \kappa_s, m)}$$

$$\frac{\mathsf{istrue}(v) = 0}{\eta \vdash (v, (\mathsf{while}\ \_\ e\ s) \cdot \kappa_s, m) \xrightarrow{s} (\textbf{\textit{skip}}, \kappa_s, m)}$$

$$\frac{[\![e]\!]^t_m = \tau}{\eta \vdash (v, f(\bar{v}, \mathcal{T}, e :: \bar{e}) \cdot \kappa_s, m) \xrightarrow{s} (e, f(v :: \bar{v}, \tau :: \mathcal{T}, \bar{e}) \cdot \kappa_s, m)}$$

$$\frac{}{\eta \vdash (v, f(\bar{v}, \mathcal{T}, \mathtt{nil}) \cdot \kappa_s, m) \xrightarrow{s} (\textbf{\textit{fexec}}(f, v :: \bar{v}, \mathcal{T},)\kappa_s, m)}$$

图 A.5　语句执行操作语义（II）

$$\mathsf{shorter}(\tau_1,\tau_2) \stackrel{\text{def}}{=} \begin{cases} \mathit{True} & \text{if} \quad (\tau_1 = \texttt{Tint8} \wedge \tau_2 = \texttt{Tint8}) \vee \\ & \qquad (\tau_1 = \texttt{Tint8} \wedge \tau_2 = \texttt{Tint16}) \vee \\ & \qquad \tau_1 = \texttt{Tint8} \wedge \tau_2 = \texttt{Tint32} \\ \mathit{True} & \text{if} \quad (\tau_1 = \texttt{Tint16} \wedge \tau_2 = \texttt{Tint16}) \vee \\ & \qquad (\tau_1 = \texttt{Tint16} \wedge \tau_2 = \texttt{Tint32}) \\ \mathit{True} & \text{if} \quad \texttt{Tint32} = \texttt{Tint32} \wedge \tau_2 = \texttt{Tint32} \\ \mathit{False} & \mathit{otherwise} \end{cases}$$

$$\mathsf{isint}(\tau) \stackrel{\text{def}}{=} \tau = \texttt{Tint8} \vee \tau = \texttt{Tint16} \vee \tau = \texttt{Tint32}$$

$$\mathsf{bop\_type}(\texttt{bop},\tau_1,\tau_2) \stackrel{\text{def}}{=} \begin{cases} \tau_1 & \text{if} \quad \mathsf{isint}(\tau_1) \wedge \mathsf{isint}(\tau_2) \wedge \mathsf{shorter}(\tau_1,\tau_2) \\ \tau_2 & \text{if} \quad \mathsf{isint}(\tau_1) \wedge \mathsf{isint}(\tau_2) \wedge \mathsf{shorter}(\tau_2,\tau_1) \\ \tau_1 & \text{if} \quad \tau_1 = \texttt{Tptr}(\_) \wedge \mathsf{isint}(\tau_2) \\ \bot & \mathit{otherwise} \end{cases}$$

$$\mathsf{field\_type}(\mathrm{id},\mathcal{D}) \stackrel{\text{def}}{=} \begin{cases} \mathsf{field\_type}(\mathrm{id},\mathcal{D}) & \text{if} \quad \mathcal{D} = (\mathrm{id}',\tau) :: \mathcal{D}' \wedge \mathrm{id} \neq \mathrm{id}' \\ \tau & \text{if} \quad \mathcal{D} = (\mathrm{id}',\tau) :: \mathcal{D}' \wedge \mathrm{id} = \mathrm{id}' \\ \bot & \mathit{otherwise} \end{cases}$$

$$\mathsf{field\_off}(\mathrm{id},\mathcal{D}) \stackrel{\text{def}}{=} \begin{cases} |\tau| + \mathsf{field\_off}(\mathrm{id},\mathcal{D}) & \text{if} \quad \mathcal{D} = (\mathrm{id}',\tau) :: \mathcal{D}' \wedge \mathrm{id} \neq \mathrm{id}' \\ 0 & \text{if} \quad \mathcal{D} = (\mathrm{id}',\tau) :: \mathcal{D}' \wedge \mathrm{id} = \mathrm{id}' \\ \bot & \mathit{otherwise} \end{cases}$$

图 A.6　表达式计算相关的函数

$$[\![e]\!]^t_m \stackrel{\text{def}}{=} \begin{cases} \texttt{Tnull} & \text{if} \quad e = \texttt{NULL} \\ \texttt{Tint32} & \text{if} \quad e = n \\ \tau & \text{if} \quad e = x \wedge m = (\_, E, \_) \wedge E(x) = (\_, \tau) \\ \tau & \text{if} \quad e = x \wedge m = (G, E, \_) \wedge x \notin \mathrm{dom}(E) \wedge G(x) = (\_, \tau) \\ \tau & \text{if} \quad e = \texttt{uop}\ e_1 \wedge [\![e_1]\!]^t_m = \tau \\ \tau & \text{if} \quad e = e_1\ \texttt{bop}\ e_2 \wedge [\![e_1]\!]^t_m = \tau_1 \wedge [\![e_2]\!]^t_m = \tau_2 \wedge \\ & \qquad \tau = \mathsf{bop\_type}(\texttt{bop},\tau_1,\tau_2) \\ \tau & \text{if} \quad e = *e_1 \wedge [\![e_1]\!]^t_m = \texttt{Tptr}(\tau) \\ \texttt{Tptr}(\tau) & \text{if} \quad e = \&e_1 \wedge [\![e_1]\!]^t_m = \tau \\ \tau & \text{if} \quad e = e_1.\mathrm{id} \wedge [\![e_1]\!]^t_m = \tau' \wedge \tau' = \texttt{Tstruct}(\mathrm{id}',\mathcal{D}) \wedge \\ & \qquad \mathsf{field\_type}(\mathrm{id},\mathcal{D}) = \tau \\ \tau & \text{if} \quad e = (\tau)e' \wedge \\ & \qquad ([\![e']\!]^t_m = \texttt{Tptr}(\tau') \wedge \tau = \texttt{Tptr}(\tau'')) \vee (\mathsf{isint}(\tau') \wedge \mathsf{isint}(\tau'')) \\ \tau & \text{if} \quad e = e_1[e_2] \wedge ([\![e_1]\!]^t_m = \texttt{Tarray}(\tau,n) \vee [\![e_1]\!]^t_m = \texttt{Tptr}(\tau)) \wedge \\ & \qquad ([\![e_2]\!]^t_m = \texttt{Tint8} \vee [\![e_2]\!]^t_m = \texttt{Tint16} \vee [\![e_2]\!]^t_m = \texttt{Tint32}) \\ \bot & \mathit{otherwise} \end{cases}$$

图 A.7　计算表达式的类型

$$
[\![e]\!]^{r}_{m} \stackrel{\text{def}}{=} \begin{cases}
\texttt{Vnull} & \text{if } e = \texttt{NULL} \\
n & \text{if } e = n \\
v & \text{if } e = x \wedge m = (G, E, M) \wedge E(x) = (b, \tau) \wedge \\
 & \quad \mathsf{load}(M, \texttt{Vptr}(b, 0), \tau) = v \\
v & \text{if } e = x \wedge m = (G, E, M) \wedge x \notin \mathrm{dom}(E) \wedge G(x) = (b, \tau) \wedge \\
 & \quad \mathsf{load}(M, (b, 0), \tau) = v \\
v & \text{if } e = \texttt{uop}\ e' \wedge [\![e']\!]^{r}_{m} = v' \wedge v = \mathsf{uop\_eval}(\texttt{uop}, v') \\
v & \text{if } e = e_1\ \texttt{bop}\ e_2 \wedge [\![e_1]\!]^{r}_{m} = v_1 \wedge [\![e_2]\!]^{r}_{m} = v_2 \wedge \\
 & \quad [\![e_1]\!]^{t}_{m} = \tau_1 \wedge [\![e_2]\!]^{t}_{m} = \tau_2 \wedge v = \mathsf{bop\_eval}(\texttt{bop}, v_1, v_2, \tau_1, \tau_2) \\
v & \text{if } e = \&e' \wedge [\![e']\!]^{l}_{m} = v \\
v & \text{if } e = *e' \wedge [\![e']\!]^{r}_{m} = \texttt{Vptr}(b, i) \wedge [\![e]\!]^{t}_{m} = \tau \wedge l = (b, i) \\
 & \quad m = (G, E, M) \wedge \mathsf{load}(M, l, \tau) = v \\
v & \text{if } e = e'.\mathrm{id} \wedge [\![e'.\mathrm{id}]\!]^{l}_{m} = \texttt{Vptr}(b, i) \wedge [\![e]\!]^{t}_{m} = \tau \wedge l = (b, i) \\
 & \quad m = (G, E, M) \wedge \mathsf{load}(M, l, \tau) = v \\
v & \text{if } e = (\tau)e' \wedge [\![e']\!]^{r}_{m} = v' \wedge \mathsf{castv}(v, \tau) = v \\
v & \text{if } e = e_1[e_2] \wedge [\![e]\!]^{l}_{m} = \texttt{Vptr}(b, i) \wedge [\![e]\!]^{t}_{m} = \tau \wedge l = (b, i) \\
 & \quad m = (G, E, M) \wedge \mathsf{load}(M, l, \tau) = v \\
\bot & \quad \textit{otherwise}
\end{cases}
$$

图 A.8　计算表达式的值

$$
[\![e]\!]^{l}_{m} \stackrel{\text{def}}{=} \begin{cases}
\texttt{Vptr}(b, 0) & \text{if } e = x \wedge m = (G, E, M) \wedge E(x) = (b, \tau) \\
\texttt{Vptr}(b, 0) & \text{if } e = x \wedge m = (G, E, M) \wedge x \notin \mathrm{dom}(E) \wedge G(x) = (b, \tau) \\
v & \text{if } e = *e' \wedge [\![e']\!]^{r}_{m} = v \\
\texttt{Vptr}(b, i + i') & \text{if } e = e'.\mathrm{id} \wedge [\![e']\!]^{l}_{m} = \texttt{Vptr}(b, i) \wedge \\
 & \quad [\![e']\!]^{t}_{m} = \texttt{Tstruct}(\mathrm{id}, \mathcal{D}) \wedge \mathsf{field\_off}(\mathrm{id}, \mathcal{D}) = i' \\
\texttt{Vptr}(b, i + |\tau| * i') & \text{if } e = e_1[e_2] \wedge [\![e_1]\!]^{l}_{m} = \texttt{Vptr}(b, i) \wedge [\![e_2]\!]^{r}_{m} = i' \wedge \\
 & \quad [\![e_1]\!]^{t}_{m} = \texttt{Tarray}(\tau, n) \wedge [\![e_2]\!]^{t}_{m} = \tau_i \wedge \mathsf{isint}(\tau_i) \wedge i' < n \\
\texttt{Vptr}(b, i + |\tau| * i') & \text{if } e = e_1[e_2] \wedge [\![e_1]\!]^{r}_{m} = \texttt{Vptr}(b, i) \wedge [\![e_2]\!]^{r}_{m} = i' \wedge \\
 & \quad [\![e_1]\!]^{t}_{m} = \texttt{Tptr}(\tau) \wedge [\![e_2]\!]^{t}_{m} = \tau_i \wedge \mathsf{isint}(\tau_i) \\
\bot & \quad \textit{otherwise}
\end{cases}
$$

图 A.9　计算表达式的地址

致　谢

有人说读博很容易失去对未来生活的一颗积极乐观的心。不停地失败和自我质疑的确很容易消磨人的信心，但好在有周围老师和同学的帮助，我的求学过程快乐并充满收获。在即将结束读博生涯时我虽然迷茫，但依然对未来拥有信心。

首先要感谢导师冯新宇教授。他的智慧和经验，他极高的学术素养，以及做事认真负责的态度都深深影响着我。从他身上我不仅学到了大量的专业知识，更学习到了处理问题的方法以及对待学术问题的严谨态度。我觉得这是读博过程中最大的收获，也是我对未来拥有信心的重要原因。

感谢付明导师，能完成学业很大程度上得益于付明老师耐心的帮助，感谢他不厌其烦地跟我讨论各种问题，感谢他兄长般的指导和爱护。虽然经常争得面红耳赤，但和他一起讨论问题的时光我觉得非常美好和难忘。

还要感谢陈意云老师的关心和爱护。感谢实验室的小伙伴们，感谢郭宇、“烤猫”和我讨论各种乱七八糟的问题，感谢什么都懂的庄重，感谢张扬带我吃喝玩乐，感谢梁红瑾维护的“吃货 list”，感谢张紫鹏，感谢何春晖，感谢蒋翰如，感谢经常喂大家“狗粮”的张晖同学，感谢实验室的所有同学，感谢大家的陪伴。

我在科大度过了九年多的时光，感谢这一路走来的兄弟姐妹，在生命中最美好的时光里，是你们陪伴着我成长。虽然大家未来可能因为工作和生活渐行渐远，但我永远珍惜和你们一起的美好和快乐。

本论文谨献给我的父亲，每当论文难以下笔时我的脑中始终是你的笑容。我内心充满悔恨和懊恼，懊恼我的任性和无知，懊恼我自己当时不理解你的痛苦和无奈。愿你在另一个世界没有痛苦，开心快乐。

感谢妈妈，我爱你。

许峰唯

2016 年 10 月 12 日

在读期间发表的学术论文与取得的研究成果

研究工作：

1. *2012 年 -2013 年 C 语言内嵌汇编建模：* 试图对 C 内嵌汇编的行为在统一的机器中进行形式化建模。

2. *2013 年 -2016 年嵌入式操作系统内核验证：* 作为核心技术骨干参与了实验室关于嵌入式操作系统内核 μC/OS-II 的项目，设计并实现了一个抢占式操作系统内核的验证框架，并成功应用该验证框架验证了 μC/OS-II 的核心功能模块。其中本人的主要工作包括：验证框架的设计和实现；全局不变式的定义；时钟中断处理程序、调度程序、消息队列模块以及内部函数的验证；部分自动证明策略的设计和实现。累计撰写 Coq 代码 8-10 万行。

已发表论文：

1. **Fengwei Xu**, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang and Zhaohui Li. A Practical Verification Framework for Preemptive OS Kernels. In Proceedings of 28th International Conference on Computer Aided Verification (**CAV'16**), Toronto, Ontario, Canada, Pages 59-79, July, 2016. （CCF A 类会议）