摘要

L4 微内核系统经历了 20 年的发展与演进。它拥有活跃的用户和开发者社区，并且已有商业版本被大规模部署在各种系统中，包括安全性关键系统。本文总结了这 20 年来关于微内核设计与实现的经验教训。我们回顾了早期的 L4 设计论文，并分析了从最初的 L4 到最新一代 L4 微内核，尤其是 seL4 的设计与实现演变。seL4 将 L4 模型推进到了最远的程度，是第一个操作系统内核，其实现通过了完整的形式化验证，并完成了最坏情况执行时间（WCET）的可靠分析。我们将展示，尽管发生了许多变化，最小化原则和高性能的进程间通信（IPC）仍然是设计与实现决策的主要驱动力。

1 简介

20 年前，Liedtke [1993a] 通过其 L4 内核展示了微内核的 IPC 可以非常快速，其性能比其他同时代微内核快了 10 到 20 倍。微内核将内核的职责压缩到最低限度：仅保留一组通用机制，而将真正的操作系统服务交由用户态的服务程序来实现 [Levin et al., 1975]。用户代码通过进程间通信（IPC）机制（通常是消息传递）与服务器通信以获取系统服务。因此，IPC 处于任何服务调用的关键路径上，降低 IPC 的开销至关重要。

在 90 年代初，IPC 性能成为了微内核的致命弱点：一次单向消息传递的典型成本大约为 100 微秒，这对于构建高性能系统来说代价太高，进而导致了将核心服务迁回内核的趋势 [Condict et al., 1994]。也有观点认为高 IPC 成本是微内核系统结构的固有结果 [Chen and Bershad, 1993]。

在这种背景下，Liedtke 展示的 IPC 成本数量级的改进显得尤为突出。随后，一系列工作探讨了 L4 的设计哲学与机制 [Liedtke, 1995, 1996]，展示了在 L4 上运行的半虚拟化 Linux，其性能开销仅为几个百分点 [Härtig et al., 1997]；L4 微内核被部署在数十亿台移动设备和安全性关键系统中，并最终实现了世界上第一个操作系统内核的功能正确性形式化验证 [Klein et al., 2009]。并且，它还对其他操作系统的研究产生了深远影响，例如 Barrelfish [Baumann et al., 2009]。

本文考察了 L4 在过去 20 年的发展历程。我们重点分析现代 L4 微内核的核心特征，它们与 Liedtke 最初的设计和实现之间的联系，以及哪些微内核“基本要素”经受住了时间的检验。我们特别关注过去的经验教训如何影响了以 seL4 [Klein et al., 2009] 为代表的最新一代 L4 微内核的设计，同时指出当前其他的 L4 版本在设计上所做的不同选择。

2 背景

2.1 L4 微内核家族

L4 起源于 Liedtke 在 1980 年代初期在 i386 平台上开发的早期系统 L3 [Liedtke, 1993b]。L3 是一个具备内建持久性支持的完整操作系统，并且已经具备用户态驱动程序，这一点至今仍是 L4 微内核的典型特征。L3 被商业化部署在几千个安装场景中，主要是学校和律师事务所等机构。像当时所有微内核一样，L3 的进程间通信（IPC）开销大约为 100 微秒。

Liedtke 最初利用 L3 进行新思想的尝试，在其早期出版物中所称的 "L3" 实际上是一次彻底重新设计的中间版本。他从 1995 年起将“L4”这一名称用于社区中广泛传播的“V2” ABI 版本。本文中我们将这一版本称为“初代 L4”。Liedtke 在基于 i486 的 PC 上使用汇编语言完全实现了该内核，并很快将其移植到 Pentium 上。

这项初始工作启动了一个为期二十年的发展历程，包括多个 ABI 修订版本和从头开始的重新实现，如图 1 所示。最初是德累斯顿工业大学（TU Dresden）和新南威尔士大学（UNSW）分别在 64 位 Alpha 和 MIPS 处理器上重新实现了该 ABI（并作了必要调整），MIPS 实现中所有运行时间较长的操作均用 C 语言完成。这两个内核都实现了亚微秒级的 IPC 性能 [Liedtke et al., 1997a]，并以开源方式发布。UNSW 的 Alpha 版本是第一个支持多处理器的 L4 内核。

当时已从 GMD 转至 IBM Watson 的 Liedtke，继续在所谓的 Version X 中探索 ABI 的演进。GMD 和 IBM 实施的知识产权限制对于其他研究者来说过于严格，因此促使德累斯顿团队从头实现了一个新的 x86 版本，命名为 Fiasco，以此表达他们在处理 IP 问题上的挫折经历。Fiasco 是第一个几乎完全用高级语言（C++）编写的 L4 内核版本，也是至今仍在维护的最早 L4 代码库。它也是第一个获得较大规模商业部署的 L4 内核版本（估计出货量达 10 万台）。

Liedtke 后来转至卡尔斯鲁厄理工学院，他和他的学生基于 C 语言从零开发了 Hazelnut，这是第一个被移植（而非重新实现）到另一架构（从

Pentium 到 ARM）的 L4 内核。

卡尔斯鲁厄团队基于 Version X 和 Hazelnut 的经验提出了一个主要的 ABI 修订版本 V4，旨在提升内核和应用程序的可移植性、支持多处理器，并解决其它已知问题。在 Liedtke 于 2001 年不幸去世后，他的学生们将这一设计落实为一个新的开源内核 L4Ka::Pistachio（简称 Pistachio）。该内核使用 C++ 实现，最初支持 x86 和 PowerPC，UNSW /NICTA 的研究团队随后将其移植到了 MIPS、Alpha、64 位 PowerPC 和 ARM。在大多数移植中，修改的代码量不到 10%。

在 NICTA，我们很快将 Pistachio 重新定位到用于资源受限的嵌入式系统，形成了一个分支版本 NICTA::Pistachio-embedded（又称 L4-embedded）。当高通公司将其作为受保护模式的实时操作系统，用于无线调制解调器处理器的固件后，该版本得到了大规模商业部署。NICTA 的衍生公司 Open Kernel Labs 继续支持并开发该内核，并将其更名为 OKL4 微内核。另一个被广泛部署的版本是由 Sysgo 公司开发的商业 V2 克隆版——PikeOS。它通过了用于安全关键航空电子设备的认证，并部署于飞机和列车中。

在日益增长的安全需求和 EROS [Shapiro et al., 1999] 的广泛影响下，访问控制机制开始转向能力模型 [Dennis and Van Horn, 1966]。OKL4 在 2008 年发布的 2.1 版本中首次引入能力机制，随后 Fiasco（后改名为 Fiasco.OC）也采用了这一机制。为了实现形式化验证，我们选择了从零开发一个专为能力机制设计的 seL4 内核，因为已有的代码库并未考虑可验证性。

近年来还出现了两个主要以虚拟化为核心目标的新设计：来自德累斯顿的 NOVA [Steinberg and Kauer, 2010] 和 OK Labs 的 OKL4 Microvisor [Heiser and Leslie, 2010]。

贯穿这二十年的主线是第 3.1 节中提出的“最小化”原则，以及对关键 IPC 操作性能的高度关注。如表 1 所示，内核开发者通常致力于尽可能的接

近微架构所能实现的性能极限。因此，L4 社区倾向于以 CPU 周期而非微秒来衡量 IPC 延迟，因为这更贴近硬件限制。事实上，第 3.1 节还提供了关于硬件对上下文切换友好程度的有趣观点（可以比较 Pentium 4 和 Itanium 在高度优化 IPC 实现中的周期开销差异）。

2.2 现代代表

我们将分析 L4 设计演进的基础建立在 seL4 上——这是我们最熟悉的版本，也是在多个方面偏离原始设计最远的系统。我们会指出其他现代版本在设计上的不同取向，并尝试理解背后的原因，以此反映 L4 社区在微内核设计理念上的一致程度。

与其他系统不同的是，seL4 是唯一一个从一开始就为了支持安全性与可靠性形式化验证而设计的系统，同时仍秉承 L4 在最小性、高性能以及支持几乎所有系统架构方面的传统。这一目标促使我们设计出一种全新的资源管理模型，所有空间资源的分配均由用户态代码显式控制，包括内核内存 [Elkaduwe et al., 2008]。它也是目前文献中第一个具有完整且可靠的最坏情况执行时间（WCET）分析的受保护的操作系统内核 [Blackham et al., 2011]。

另一个值得关注的系统是 Fiasco.OC，它是唯一跨越了 L4 大部分发展历史的代码库，最初是原始 ABI 的一个克隆（甚至并未面向性能优化），但后来逐步发展为一款高性能内核，具备新一代内核的各种特征，包括基于能力的访问控制。Fiasco 也曾作为许多设计探索的测试平台，特别是在实时支持方面 [Härtig and Roitzsch, 2006]。

此外还有两个从零开发的较新设计：专为 x86 平台上的硬件辅助虚拟化而设计的 NOVA [Steinberg and Kauer, 2010]；以及面向 ARM 处理器的高效准虚拟化商业平台 OKL4 Microvisor [Heiser and Leslie, 2010]（简称 OKL4）。

3 微内核设计

Liedtke [1995] 概述了驱动最初 L4 设计的原则与机制。我们将探讨这些原则是如何随着时间演进的，并重点将他们与当前一代微内核的进行比较。

3.1 最小化

Liedtke 设计的主要驱动力是最小性与 IPC 性能，并且他坚信前者有助于后者。具体来说，他提出了微内核最小化原则如下：

仅当将某个概念保留在微内核中是系统实现所必需的，即如果将其移出内核从而允许多种实现方式将无法满足系统所需功能时，才可接受其存在于内核中 [Liedtke, 1995]。

该原则是“仅在内核中保留最基本机制，而不包含任何策略”这一思想的更明确表述，并持续成为 L4 微内核设计的基石。接下来的章节的讨论将展示社区持续努力地去除功能或将其替换为更通用（且功能更强大）机制的过程。

如表 2 所示，这一原则的坚持可以从源代码规模的对比中看出：尽管系统随着时间推移往往会变得更大，但作为家族中最新成员的 seL4（尽管在设计上与传统模型的偏离最为显著）其代码规模仍基本与早期版本相当。形式化验证对最小化具有特别强的推动力，因为即使是 9,000 行代码的 SLOC 也已经接近可验证性的极限。

保留项：最小化作为关键设计原则。

尽管如此，迄今为止尚无任何 L4 微内核设计者声称其开发了严格遵循最小化原则的“纯粹”微内核。例如，所有 L4 内核都包含一个调度器，并在内核中实现了一定的调度策略（通常为固定优先级轮转调度）。目前还没有人提出一种真正通用的内核内调度器，或者一种能够将所有调度策略委托给用户态、且不会带来高开销的可行机制。

3.2 IPC

3.2.1 同步 IPC

最初的 L4 仅支持同步（回合式）IPC 作为唯一的通信、同步与信号传递机制。同步 IPC 避免了内核中的缓冲以及缓冲所附带的管理与拷贝开销，

同时也是后续若干实现优化的前提条件，如惰性调度（第 4.2 节）、直接进程切换（第 4.3 节）以及临时映射（第 3.2.2 节）等。

尽管这一机制在概念与实现上都极其简洁且符合最小化原则，我们仍在实践中发现该模型也存在显著缺陷：它将多线程设计强加于原本可以保持简单的系统，进而带来了同步方面的复杂性。例如，缺乏类似 UNIX select() 的功能，导致每个中断源都需要一个独立线程，单线程服务器无法同时等待客户端的请求与中断。

在 L4-embedded 中，我们通过引入异步通知机制来解决上述问题，这是一种非常简洁的异步 IPC 形式。我们在 seL4 中进一步发展了这一模型，引入了异步端点（AEP，端点将在第 3.2.3 节中详细介绍）：发送操作是非阻塞的，且对接收者而言是异步的，接收者可以选择阻塞等待或轮询接收消息。逻辑上，异步端点类似于将多个二进制信号量打包进一个字中：每个异步端点具有一个单字的通知字段，发送操作指定一个掩码位（通常为一个比特位），该比特位将与通知字段进行按位“或”操作。等待操作实际上相当于对通知字段的 select()。

在我们当前的设计中，一个异步端点可以绑定到某个特定线程。如果线程正在同步端点上等待，而此时有通知到达，该通知将像同步消息一样被交付（但带有标识表明他实际上是一个通知）。

总而言之，seL4 与大多数其他 L4 内核一样，保留了同步 IPC 模型，但通过异步通知进行了增强。OKL4 完全放弃了同步 IPC，转而使用虚拟中断（本质上即异步通知）。NOVA 通过计数信号量机制扩展了同步 IPC [Steinberg and Kauer, 2010]，而 Fiasco.OC 也通过引入虚拟中断增强了同步 IPC。

总结：seL4、NOVA 和 Fiasco.OC 在同步 IPC 基础上引入了异步通知机制；OKL4 则完全以异步机制取代了同步 IPC。

引入两种 IPC 机制违反了“最小化”原则，因此在这一点上 OKL4 的做法更为“纯粹”（尽管其通道抽象在另一个层面上违背了该原则，详见第 3.2.2 节）。此外，在多核环境中，同步 IPC 的效用也变得更加可疑：类似 RPC 的服务器调用会将客户端与服务器强行串行化，而如果它们运行在不同

核心上，这种串行化应尽量避免。因此，我们认为基于异步通知的通信协议将会更加普及，而且未来转向完全异步 IPC 的可能性仍然存在。

3.2.2 IPC 消息结构

最初的 L4 IPC 拥有丰富的语义。除了寄存器内（“短”）消息外，还支持几乎任意大小的消息（包括字对齐的缓冲区以及多个非对齐的字符串）。结合完全同步的设计，该机制避免了冗余拷贝。

寄存器参数支持零拷贝：内核始终在发送方上下文中启动 IPC，然后切换到接收方上下文，期间不触碰消息寄存器。这一机制的缺点在于其具有架构依赖性，且在某些平台（如 x86-32）上支持的零拷贝消息大小较小。实际上，随着 ABI 的演变，可用寄存器数量频繁变化，因为系统调用参数的改变会占用或释放寄存器。

Pistachio 引入了虚拟消息寄存器的概念，其数量在 16 到 64 之间可配置。在实现上，一部分虚拟寄存器映射到物理寄存器，其余部分存储于每个线程的地址空间中一段“固定”的内存区域。该区域不会触发页面错误，因此具有与寄存器相似的语义。内联访问函数隐藏了物理寄存器与内存寄存器之间的差异。seL4 和 Fiasco.OC 延续了这一做法。

此方案的动机主要有两个：其一，虚拟消息寄存器提升了跨架构的可移植性；其二，对于超过物理寄存器数量的中等大小消息，它们的性能开销较小，仅需少量拷贝操作而不需要“长消息”IPC 所需的临时映射。

随着时间推移，上下文切换的架构开销成为 IPC 性能的主导因素，因此寄存器内消息传输的优势逐渐减弱。例如，在 ARM11 上，使用寄存器传输 4 字消息可带来约 10% 的性能提升；而在 Cortex A9 上，该提升下降至约 4%。在 x86-32 上，为消息传输保留寄存器反而会妨碍编译器的优化能力。

总结：物理消息寄存器被虚拟消息寄存器所取代。

“长消息”可在一次 IPC 中指定多个缓冲区以均摊模式切换与上下文切换的开销。在发送方上下文中，内核建立一个临时映射窗口指向接收方地址空间，并直接将消息复制过去。该过程可能在源或目的地址空间中触发页面错误，进而要求内核处理嵌套异常。内核需调用用户级页面错误处理程序，同时必须模拟异常发生于普通用户态执行过程中。恢复后，需重建原始系统调

用上下文。这一流程导致内核实现极为复杂，且充满边界情况，容易引入错误。

尽管“长消息 IPC”提供了一些用户态难以仿真的功能，实际上其使用场景极少：共享缓冲区几乎总是更优选择，适用于大多数批量数据传输。同时，异步接口也可实现批量传输，而无需内核提供显式支持。“长消息”主要用于传统 POSIX 的 read/write 接口，需要将任意缓冲区数据传给无法访问客户端内存的服务器。然而，随着 Linux 成为 POSIX 中间件，其与应用程序共享内存，这一用例已被按引用传递（pass-by-reference）所取代。其他剩余用例也要么接口灵活、要么可通过共享内存实现。

“长消息 IPC”违背了“最小化原则”（此原则关注的是功能，而非性能）。因此，由于内核复杂性以及存在用户态替代方案，我们在 L4-embedded 中移除了长消息 IPC，NOVA 与 Fiasco.OC 也不再提供此功能。对于 seL4，放弃长消息的理由更加充分：seL4 的形式化验证明确避免了任何形式的内核并发 [Klein et al., 2009]，而嵌套异常将引入并发。同时，这还打破了 C 语言的语义，引入额外控制流。虽然理论上可以对嵌套异常进行形式化推理，但这会显著增加验证工作的难度。当然，内核可以通过增加检查来避免页错误，但这不仅降低了最佳性能，还需要更复杂的形式模型来证明其完整性与正确性。

总结：长消息 IPC 被废弃。

3.2.3 IPC 目标

最初的 L4 将线程作为 IPC 操作的目标。这种设计的目的是为了避免间接机制而导致的缓存和 TLB 污染，尽管 Liedtke [1993a] 指出，端口可以以大约 12% 的开销实现（主要是额外的 2 次 TLB 未命中）。该模型要求线程 ID 是唯一标识符。这个模型的缺点是信息隐藏性差。

一个多线程服务器必须向客户端暴露其内部结构，以实现客户端负载均衡，或者使用一个网关线程，而这可能成为瓶颈，并引入额外的通信与同步开销。虽然曾提出一些缓解方法，但都存在不足。此外，现代 CPU 对大页的支持降低了由于间接机制带来的 TLB 污染，因为大页更容易使不同实体驻留在同一页中。而且，全局 ID 还引入了隐蔽信道 [Shapiro, 2003]。

受 EROS [Shapiro et al., 1999] 的影响，seL4 和 Fiasco.OC

[Lackorzynski and Warg, 2009] 采用了 IPC 端点作为 IPC 的目标。seL4 中的（同步）端点本质上类似端口：挂起发送者或接收者队列的根现在是一个独立的内核对象，而不再是接收线程的线程控制块（TCB）的一部分。与 Mach 的端口不同，IPC 端点不提供任何缓冲。

替代方案：将线程 ID 替换为类似端口的 IPC 端点作为消息目标。

3.2.4 IPC 超时

阻塞式 IPC 机制可能导致拒绝服务（DOS）攻击。例如，一个恶意（或有缺陷的）客户端可能向服务器发送请求但永远不接收回复；由于是会合式 IPC，发送者将会无限期阻塞，除非实现看门狗机制进行中止和重启。L4 的长 IPC 还可能实现更复杂的攻击方式：恶意客户端可以向服务器发送一条长消息，确保其触发页面错误并阻止其分页器处理该错误。

为了防范此类攻击，原始的 L4 IPC 操作中包含超时机制。具体来说，一次 IPC 系统调用中可以指定 4 个超时：一个用于限制进入发送阶段前的阻塞时间，一个用于接收阶段的阻塞时间，以及两个用于发送和接收阶段页面错误时的阻塞限制（适用于长 IPC）。

超时值使用浮点格式编码，支持零、无限大以及从 1 毫秒到数周的有限值。它们增加了管理唤醒列表的复杂性。

然而，实际上超时机制在防御 DOS 攻击中作用微乎其微。对于非平凡系统，并不存在理论或良好的启发式方法来选择超时时间。实际上，系统中实际上只使用了零和无限两种超时值：客户端使用无限超时进行发送和接收，服务器在接收请求时使用无限超时，在发送回复时使用零超时。（客户端使用一种类似 RPC 的调用操作，即先发送后原子切换到接收阶段，从而保证客户端在服务器回复时处于可接收状态。）传统的看门狗计时器代表了一种更好的方式，用于检测未响应的 IPC 交互（如死锁造成的）。

L4-embedded 已经放弃了长 IPC，因此我们用一个标志位取代了原来的超时机制，该标志位支持轮询（零超时）或阻塞（无限超时）两种选择。仅需两个标志位，分别用于发送和接收阶段。seL4 延用了这一模型。OKL4 的完全异步模型与超时机制不兼容，也不需要它来防御 DOS 攻击。

超时机制还可以通过等待来自一个不存在线程的消息来实现定时休眠，这对实时系统很有用。Dresden 曾尝试扩展模型，包括绝对超时机制，即超

时在某个墙上时钟的时间点而不是相对系统调用开始时间到。我们的做法是让用户态访问（物理或虚拟）定时器。

被废弃：seL4 和 OKL4 中废弃了 IPC 超时机制。

3.2.5 通信控制

在原始的 L4 中，内核会将发送者无法伪造的 ID 传递给接收者。这使得服务器可以实现一种可自由支配的访问控制方式，即忽略不希望接收的消息。然而，恶意客户端仍可以用大量虚假消息（尤其是大消息）轰炸服务器。即使复制操作由内核执行，接收这些消息所耗费的时间也会阻止服务器执行有用的工作，而判断是否丢弃这些消息本身也消耗时间。因此，这种消息构成了一种 DOS 攻击，只有通过内核支持阻止不希望的消息发送才能避免这种攻击 [Liedtke et al., 1997b]。强制访问控制策略同样需要一种机制作为中介并授权通信。

原始的 L4 通过一种叫做“氏族与首领”（clans and chiefs）的机制来实现这一点：进程被组织成“氏族”层级结构，每个氏族有一个指定的“首领”。氏族内部的消息可以自由传递，并由内核保证消息完整性。而跨越氏族边界的消息（无论是发出还是接收）都会被重定向到该氏族的首领，从而由其控制消息流。该机制还支持对不可信子系统的隔离 [Lipner, 1975]。

Liedtke [1995] 认为该模型每次 IPC 操作仅增加两个周期开销，因为氏族 ID 可以被编码进线程 ID 以实现快速比较。然而，这种低开销仅适用于可直接通信的情况。一旦发生重定向，每次重定向都会为一次（逻辑上）单轮 IPC 添加两个消息，造成显著开销。此外，严格的线程层级结构在实践中非常笨拙（大概是构建基于 L4 系统时最令人痛苦的功能之一）。在强制访问控制场景中，该模型最终迅速演变为“每个进程一个首领”。这正是内核强制策略（地址空间层级）限制设计空间的典型例子。

由于这些缺点，许多 L4 实现根本没有实现氏族与首领机制（或在构建时将其禁用），这也意味着无法控制 IPC。后来也曾尝试使用更通用的 IPC 重定向模型 [Jaeger et al., 1999]，但并未能获得广泛采用。最终该问题通过灵活的、基于能力的端点访问控制机制得以解决。

被废弃：氏族与首领机制。

3.3 用户级设备驱动

L4（或者更准确地说，它的前身 L3）最具创新性的理念之一，是将所有设备驱动程序都作为用户态进程来实现。这一设计是其“最小性原则”的核心体现，也成为所有 L4 内核的标志性特征之一。将驱动移出内核的一个重要原因是为了支持形式化验证：一旦将未经验证的代码（如驱动）加入内核，将彻底破坏内核的安全性与正确性保证，而现实系统中庞大的驱动代码量又使得对其进行形式化验证在当前阶段几乎不可能实现。

当然，仍有极少数驱动程序最好保留在内核中。在现代 L4 内核中，这通常是定时器驱动，用于在时间片结束时抢占用户进程、中断控制器驱动，用于将硬件中断安全地分发给用户态驱动程序。

L4 的用户态驱动模型与将中断建模为 IPC 消息这一机制紧密耦合：当发生中断时，内核会向相应的驱动程序发送一条 IPC 消息。这个机制的具体细节（比如虚拟线程发出的 IPC 与上行调用的差异），以及中断消息的绑定与确认协议，多年来曾多次调整，甚至反复变化，但整体思路始终一致。最显著的变化是将中断传递方式从同步 IPC 转为异步 IPC，原因在于同步方式需要模拟内核中的虚拟线程作为中断消息的源头，这在实现上比较复杂，而异步传递大大简化了设计。

用户态驱动程序受益于硬件虚拟化带来的改进。IOMMU 的出现允许用户态驱动程序安全地访问硬件设备。现代网络接口还支持中断合并，进一步降低了中断处理带来的系统开销。在 x86 架构下，TLB 标记等机制大幅降低了上下文切换的成本，也让用户态驱动变得更高效。

如今，用户态驱动程序已经成为主流做法，在 Linux、Windows 和 macOS 等操作系统中都得到了支持，尽管这些系统中并未特别鼓励这种做法。相比之下，由于 L4 拥有高度优化的 IPC 机制，其用户态驱动模型的系统开销要小得多。我们过去也在 Linux 上展示过，即使在较为通用的系统中，只要硬件支持良好，用户态驱动程序依然可以实现极低的系统开销。但在实际应用中，真正对性能有极高要求的设备只是极少数，因此大多数驱动放在用户态对整体系统性能影响并不大。

3.4 资源管理

初代 L4 的资源管理方式，与其通信控制策略类似，严重依赖于进程层

级结构。这种方式既适用于进程管理，也适用于虚拟内存管理。层级结构在资源管理和回收方面是一种有效手段，并且能够约束子系统（系统机制限制子进程的权限为其父进程的子集），但其缺点是灵活性差。然而，这种层级结构本质上是一种策略，因此与微内核的理念（如第 3.2.5 节所述）不相匹配。能力机制可以打破层级结构的限制，这也是所有现代 L4 内核都采用基于能力的访问控制的原因之一。以下我们将分析初代 L4 模型中最重要的资源管理问题，以及我们现在是如何处理这些问题的。

3.4.1 进程层级结构

一个进程（在 L4 中本质上就是一个页表加上一些关联线程）会消耗内核资源，而不受控制地分配 TCB 和页表很容易导致拒绝服务攻击。初代 L4 通过进程层级结构来处理这个问题：“任务 ID”本质上是对地址空间的能力，允许创建和删除。它们的数量是有限的（数量级为几千），由内核按先到先得的方式分配。它们可以被委托，但只能沿着层级向上或向下传递。（它们还与用于 IPC 控制的线程层级密切相关，见第 3.2.5 节。）在典型的设置中，初始用户进程会在创建任何子进程之前先占用所有的任务 ID。可以预料，这种模型被证明是僵化且受限的；最终它被完整的能力机制所取代。

已废弃：层级式进程管理。

3.4.2 递归页映射

初代 L4 将对物理内存帧的权限与现有的页映射绑定在一起。一个进程如果在其地址空间中拥有一个页帧的有效映射，就拥有将该页映射到另一个地址空间的权利。进程也可以将某个页面“授予”出去，这会将该页（以及对其的权限）从授予者那里移除。映射（但不是授予）可以通过 unmap 操作撤销。地址空间在创建时为空，需要通过映射原语填充。

递归映射模型以原始地址空间 σ0 为基础，该空间在内核启动后获得所有剩余空闲帧的一对一映射。σ0 是所有启动时创建进程的缺页处理器，并会将其页面首次映射给第一个请求该页（通过地址缺页）的进程。

值得注意的是，尽管 L4 的内存模型从每个帧出发构建了一个映射层级，它并不强制地址空间视图是层级化的：映射是通过 IPC 建立的（类似于通过 IPC 消息传递能力），一个进程可以将其某页映射给任何它允许发送

IPC 的进程（前提是接收方同意接收该映射）。与 Mach 不同，L4 没有内存对象语义，只有更接近 Mach 内核内部 pmap 接口的底层地址空间管理机制，而不是用户可见的内存对象抽象 [Rashid et al., 1988]。内存对象、写时复制（copy-on-write）和影子链（shadow-chain）都只是用户级构造的抽象或实现方式。

递归映射模型在概念上简洁优雅，Liedtke 对此非常自豪——它在许多论文中都占据重要位置，包括最早的那篇 [Liedtke, 1993a]，以及他所有的演讲中。然而，实践表明它存在显著缺陷。

为了支持页粒度的撤销，递归地址空间模型需要以映射数据库的形式进行大量记录和管理。而且，L4 内存模型的普适性也使得两个串通的进程可以通过将相同的帧递归映射到彼此地址空间中的不同页面来迫使内核消耗大量内存，这是在 64 位硬件上可能发生的拒绝服务攻击，仅能通过控制 IPC（依赖令人诟病的 clans-and-chiefs）来阻止。

在 L4-embedded 中，我们移除了递归映射模型，因为我们发现即使在没有恶意进程的情况下，映射数据库就占用了 25–50% 的内核内存。我们用一种更贴近硬件的模型取而代之，其中映射总是来源于物理内存帧的区间。

这种方式的代价是失去了内存的细粒度委托和撤销能力（除非对页表进行暴力扫描），因此我们只将它视为一个过渡性的缓解手段。OKL4 在这个最小模型的基础上做了一些扩展，但并未达到原模型的通用性和细粒度控制。

映射控制在基于能力的系统中非常容易实现，使用标准的 grant-take 模型的变体 [Lipton and Snyder, 1977]。这正是 seL4 的做法：映射权限是通过对物理帧的能力来传递的，而不是通过对以该帧为后备的虚拟页的访问权限来传递的，因此 seL4 的模型不是递归的。即使拥有帧能力，映射仍然严格受限于内核用于记录映射的显式内存模型，如下所述。

Xen 提供了一个有趣的对比点。其 grant table 允许在拥有有效映射的基础上创建一种类似帧能力的机制，可传递给其他域以建立共享映射 [Fraser et al., 2004]。最近有一个扩展 proposal，允许撤销帧 [Ram et al., 2010]。其内存映射原语的语义与 seL4 大致相似，但不包括缺页传播。在 Xen 的场景下，只有在共享时才会承担细粒度委托和撤销所带来的开销。

NOVA 和 Fiasco.OC 仍然保留递归地址空间模型，映射权限由有效映射

的持有决定。Fiasco.OC 通过每个任务的内核内存池机制，解决了无法限制映射和记录开销分配的问题。

现有的 L4 地址空间模型（尤其是细粒度委托与撤销）在机制的通用性与最小性之间，以及可能更节省空间的领域特定方法之间做出了不同权衡。

多种方法：一些 L4 内核保留了递归构建地址空间的模型，而 seL4 与 OKL4 则从物理帧出发建立映射。

3.4.3 内核内存

虽然能力机制为权限委托提供了一种简洁而优雅的模型，但单靠它们无法解决资源管理的问题。一个拥有映射授予权限的恶意线程，仍然可以利用这一点创建大量映射，迫使内核为元数据消耗大量内存，从而可能导致系统遭受拒绝服务（DoS）攻击。L4 内核传统上使用一个固定大小的堆为其数据结构分配内存。初代 L4 使用一个称为 σ1 的内核分页器，允许内核通过它向用户空间请求额外内存。然而，这并不能解决恶意（或存在缺陷的）用户代码强迫系统消耗不合理内存的问题，它只是将问题转移了。因此，大多数 L4 内核并不支持 σ1。

这个根本问题，也存在于多数操作系统中，其本质是通过共享内核堆导致的用户进程隔离不足。一个令人满意的方案必须能够提供完全的隔离。即使在一个以能力表示权限的系统中，如果存在不受能力系统控制的资源，那么也无法对系统的安全状态进行完整推理。那些将内存仅作为用户级内容缓存来管理的内核，仅能部分缓解这个问题。虽然基于缓存的方法可以避免因内存耗尽引发的 DoS 攻击，但它们并不支持内核内存的严格隔离，而这种隔离是性能隔离或实时系统的前提条件，而且还可能引入隐蔽通道（covert channels）。

Liedtke 等人 [1997b] 研究了这个问题，并提出了每个进程独立内核堆的机制，并辅以在内存耗尽时向内核捐赠额外内存的方案。NOVA、Fiasco 和 OKL4 都采用了这一方案的不同变体。每进程内核堆简化了用户级别的管理（因为用户不再控制分配），代价是失去了在不销毁整个进程的情况下撤销分配的能力，以及无法直接推理已分配内存的状态（只能对其进行限制）。这一权衡仍在学术界持续探索中。

我们在 seL4 中采用了一种根本不同的做法，其内核内存管理模型是

seL4 对操作系统设计的主要贡献之一。我们希望能够对资源使用和隔离进行精确推理，因此我们将所有内核内存（除了内核启动时使用的固定数量内存，包括其严格受限的栈）都纳入由能力控制的权限系统中。具体而言，我们彻底移除了内核堆，并提供给用户空间一种机制，使其能在内核分配数据结构时识别获得授权的内存。

这一机制的副作用是显著减少了内核的体积和复杂性，这对形式化验证是一个重大优势。关键在于：使所有内核对象都显式化，并受能力机制控制访问。这种设计灵感来源于基于硬件的能力系统，特别是 CAP 系统[Needham 和 Walker, 1977]，其中硬件解释的能力直接对应内存。HiStar [Zeldovich 等, 2011] 也使所有内核对象显式化，不过它采用的是基于缓存的内存管理方式。

当然，使内核对象对用户可见，并不意味着拥有相关能力的用户可以直接读写该对象。能力仅提供调用该对象特定方法（的子集）的权限，其中包括销毁该对象。（对象一旦创建，其大小就不会再改变。）尤其重要的是，seL4 中的内核对象类型包括“未类型化内存”（Untyped），它可以被用于创建其他对象。对 Untyped 唯一允许的操作是将其部分区域重新类型化为某种对象类型。新对象与原始 Untyped 的关系被记录在能力派生树（capability derivation tree）中，该树还记录其他能力派生形式，如创建权限减少的能力副本。

一旦某块 Untyped 被重新类型化，对应区域的唯一允许操作就是撤销（revoke）所派生的对象（见下文）。在 seL4 中，重新类型化是唯一创建对象的方式。因此，系统可以通过限制对 Untyped 内存的访问来控制资源分配。重新类型化也可以产生更小的 Untyped 对象，这些对象可以独立管理——这是资源管理委托的关键所在。Untyped 的派生过程还确保了内核的完整性属性，即任何两个已类型化对象不会重叠。

表 3 给出了 seL4 所有对象类型及其用途的完整列表。用户空间只能直接访问（加载/存储/获取）映射到其地址空间中的 Frame 所对应的内存（这是通过将 Frame 能力插入页表实现的）。

这个模型具有以下属性：

1. 所有权限都是通过能力显式赋予的；

2. 数据访问和权限可以被限制（confined）；

3. 内核本身（用于其数据结构）也遵循分配给应用程序的权限，包括对物理内存的消耗；

4. 每个内核对象都可以独立于其他对象被回收；

5. 所有操作的执行时间，或可被抢占的时间，都是“短”的（常数时间，或与对象大小线性相关，但不超过一个页）。

属性 1–3 保证了我们能够对系统资源及安全性进行推理。尤其是属性 3，在形式化证明内核可确保完整性、权限限制和保密性时至关重要 [Murray 等, 2013；Sewell 等, 2011]。属性 5 保证了所有内核操作的延迟有界，从而支持其用于硬实时系统 [Blackham 等, 2011]。属性 4 确保了内核完整性。任何持有合适能力的实体都可以随时回收一个对象（从而使原始 Untyped 区域可再次用于创建对象）。例如，在不销毁整个地址空间的情况下，可以单独回收页表所占用的内存。为了实现这一点，内核必须能够检测并使对该对象的所有引用失效。

这一需求通过能力派生树得以满足。撤销对象是通过在派生树上较高位置的 Untyped 对象上调用 revoke() 方法实现的；这将移除所有从该 Untyped 派生出的对象的所有能力。当一个对象的最后一个能力被移除时，该对象也会被删除。这将移除它在内核中与其他对象之间的所有依赖，使该内存区域可以重新使用。移除最后一个能力的行为很容易检测，因为这会清除能力树中指向某个特定内存位置的最后一个叶节点。

撤销操作需要用户空间在更高层抽象（如进程）上进行 Untyped 能力与对象的关联管理。这一机制的具体语义，以及 Untyped 与用户级账本的关系，目前仍在研究探索中。

补充：seL4 实现了用户级对内核内存的控制；Fiasco.OC 则实现了内核内存配额机制（kernel memory quota）。

3.4.4 多核

L4 社区很早就开始研究多处理器相关问题。除 L4/Alpha 和 L4/MIPS 外，大多数工作都是在 x86 平台上完成的，这些平台是最早可负担的多处理器系统。早期的 x86 多处理器和多核系统具有较高的核间通信成本，并且没有共享缓存。因此，标准做法是使用每个处理器独立的调度队列（核心之间的内核数据共享最小化），线程迁移只在用户空间显式请求时发生。Uhlig

[2005] 探讨了在多核平台上的锁机制、同步与一致性问题，并基于 RCU（Read-Copy-Update）[McKenney 等, 2002] 提出了可扩展的内核数据结构并发控制方法。NOVA 和 Fiasco.OC 广泛采用了 RCU。

随着研究重点从高端服务器转向嵌入式和实时平台，多处理器问题一度被搁置，直到近期嵌入式处理器开始支持多核，这一问题才被重新关注。这类处理器通常具有较低的核间通信成本和共享的 L2 缓存，因此与 x86 平台的取舍不同。在这类平台上，线程迁移的成本通常不足以支撑额外的系统调用开销，因此全局调度器具有更大的意义。

在此背景下，形式化验证引入了新的限制。如第 3.2.2 节所述，并发性给验证带来巨大挑战，因此我们尽可能地将并发排除在内核之外。对于多核系统，这意味着要么采用大内核锁（big kernel lock），要么采用多内核结构（multikernel）[Baumann 等, 2009]。对于微内核而言，由于系统调用执行时间较短，大锁策略并不像看起来那么荒谬，至少对于共享 L2 缓存的少量核心而言，其锁争用是可以接受的。

我们目前正在探索的策略是“簇状多内核”（clustered multikernel）——这是一种混合方式，在共享 L2 缓存的核心之间使用大锁，在不同簇之间则采取限制性多内核方式（不允许在内核之间迁移内存）[von Tessin, 2012]。簇状多内核避免了大部分内核代码中的并发问题，因此可以在一定假设条件下继续维持形式化保证。主要的假设包括：（1）锁机制本身以及其之外的代码都是正确且无竞争的；（2）内核对与用户空间共享内存（对于 seL4，仅为虚拟 IPC 消息寄存器块）中的并发修改是健壮的。

该方法的优势在于只需做出较小的修改即可保留现有的单核形式化证明。我们已经在形式上完成了多个单核自动机的并行组合，并证明该组合仍然符合系统的行为精化。然而缺点是：形式化保证不再涵盖整个内核，并且 lifting 框架所使用的大步语义（large-step semantics）限制了我们进一步扩展该形式框架，使其无法对锁的正确性、用户-内核并发以及资源迁移放宽等方面进行更细致的形式化推理。

尽管我们不作强烈断言，但某种变体的簇状多内核可能最终会成为实现多处理器微内核完全形式化验证的最佳方式。不过，在形式化方面仍需大量工作，才能对微内核级别的细粒度交错行为进行严密的推理。

4 微内核实现

Liedtke [1993a] 总结了一系列设计决策与实现技巧，使得最初 i486 版本中的 IPC 十分高效，尽管其中一些带有过早优化的气息。

其中一些技巧前文已提到，比如（现已废弃的）long IPC 中使用的临时映射窗口。还有一些是无争议的，例如将发送-接收操作组合为一次系统调用（用于类似 RPC 的客户端调用和服务端的“回复-等待”模式）。我们将在下文详细讨论剩余内容，包括一些传统的 L4 实现方法，这些方法虽较少公开讨论，但在社区中早已成为理所当然的默认实践。

4.1 严格的进程导向与虚拟 TCB 数组

最初的 L4 为每个线程分配了独立的内核栈，该栈位于该线程 TCB（线程控制块）之上的同一页内。TCB 的基地址因此可以通过内核栈指针进行掩码运算获得。这种设计只需要一个 TLB 项即可覆盖线程的 TCB 和栈。

此外，所有 TCB 被分配在一个稀疏的虚拟地址数组中，通过线程 ID 进行索引。在 IPC 期间，这允许非常快速地查找目标线程的 TCB，而无需事先验证 ID 的合法性：如果调用者提供了无效 ID，查找操作可能会访问未映射的 TCB，从而触发页错误，由内核处理并终止该 IPC。如果未发生页错误，则通过将调用者提供的 ID 与 TCB 中存储的实际 ID 进行比较来验证其合法性。（原始 L4 的线程 ID 包含版本号，每次线程销毁并重建时版本号都会变化，从而保证线程 ID 在时间上唯一。将当前 ID 存储在 TCB 中可用于检测过时的线程 ID。）

这两种机制都存在代价：大量的内核栈导致每个线程的内存开销较大，也增加了内核的缓存占用。虚拟 TCB 数组则扩大了内核的虚拟内存使用以及 TLB 占用，但避免了额外查找表带来的缓存开销。在仅支持单页大小和非标记 TLB 的处理器上，优化空间有限，只能尽量将数据结构分组以减少页面使用。然而，RISC 处理器通常支持大页（或物理内存寻址）和带标签的 TLB，使得取舍方案有所不同。

当 L4 开始在嵌入式领域获得应用时，内核的内存使用成为一个重大问题，因此设计必须重新审视。

在 Pentium 上采用单栈内核的初步实验显示内核内存使用减少，且在微基准测试中 IPC 性能有所提升 [Haeberlen, 2003]。Warton [2005] 在 ARMv5

处理器上对 Pistachio 进程式内核与基于事件（单栈）内核进行了全面性能评估。他发现两者在微基准中性能相当（相差一般在 1% 以内），但在多任务负载（如 AIM7）下，事件式内核性能高出 20%。尽管事件内核的 TCB 大小是进程内核的两倍多（因为需要存储 continuation），其每线程内存开销却只有进程内核的四分之一。

与此同时，Nourai [2005] 分析了虚拟地址和物理地址方式的 TCB 的权衡。他在 Pistachio（MIPS64 架构）中实现了物理寻址版本。尽管在微基准测试中几乎看不出差异，但在 TLB 压力较大的工作负载下，物理寻址版本表现更优。MIPS 较为特殊，它即便启用了 MMU 也支持物理寻址，而大多数其他架构通常通过幂等大页映射以及“全局”映射来模拟物理寻址。不过 Nourai 的结果足以表明虚拟寻址的 TCB 并无明显性能优势。

事件驱动内核避免了内核中的页错误异常，因而可以保持 C 语言语义。如第 3.2.2 节所讨论，保持在 C 的语义范围内可以简化形式化验证的复杂度。

综合这些结果，我们在 L4embedded 中选择了事件驱动设计与物理地址的内核数据结构，seL4 也采用了这一策略。尽管这一决定最初是出于嵌入式系统资源紧张和形式化验证的需要，但其优势并不局限于这些场景，我们认为这在现代硬件上是更优的通用方法。

4.2 延迟调度（Lazy scheduling）

在会合 IPC 模型中，线程状态频繁在“可运行”与“阻塞”之间切换。这导致频繁的队列操作：将线程加入或移出就绪队列，可能在一个时间片内反复多次发生。

Liedtke 提出了“延迟调度”的技巧以最小化这些队列操作：当线程在执行 IPC 时阻塞，内核只更新其 TCB 中的状态，但不立即从就绪队列中移除，假设它很快就会被唤醒。当因为时间片耗尽而调用调度器时，调度器会遍历就绪队列，直到找到真正处于可运行状态的线程，并将不可运行的线程移除。该方法还辅以对唤醒队列的延迟更新。

延迟调度将原本在高频操作中的工作，延迟到较少发生的调度阶段来处理。然而，我们在分析 seL4 的最坏情况执行时间（WCET）时发现了它的一个问题 [Blackham 等，2012]：调度器的执行时间仅受系统中线程数量的

限制！

为了解决这个问题，我们采用了另一种优化方法，称为 Benno 调度，它不会出现上述异常的时序行为：该策略中，就绪队列中包含所有“可运行”的线程（除了当前正在执行的那个）。因此，在 IPC 过程中线程被阻塞或解除阻塞时，通常无需修改就绪队列。在时间片用尽时，内核只需将仍然“可运行但已停止执行”的线程重新插入就绪队列中。由于移除了超时机制，因此也不再需要管理唤醒队列。尽管端点等待队列仍需严格维护，但在常见场景中（例如服务器通过一个端点响应客户端请求），这些数据结构通常处于缓存中，因此操作开销很低。

这种方法具有与延迟调度相似的平均性能，同时也具有可界定的最坏执行时间。

4.3 直接进程切换

L4 传统上尽量避免在 IPC（进程间通信）期间运行调度器。如果一个线程在发起 IPC 调用时被阻塞，内核会切换到一个可以明确识别的可运行线程，并让该线程在原线程的时间片上运行，通常忽略优先级。这种方式称为直接进程切换（direct process switch）。

这种方法比一开始看起来更合理，尤其是在假设服务器线程优先级不低于客户端线程的情况下。一方面，如果一个（客户端）线程发起一个调用操作（到服务器），调用方显然会阻塞直到被回复。既然线程能够执行系统调用，说明它是当前优先级最高的可运行线程，而保持该优先级最好的方式是让被调用方尽快完成操作（而被调用方通常本身也具有较高优先级）。

另一方面，如果服务器对一个等待的客户端执行“回复并等待”（reply-and-wait）操作，并且服务器还有来自其他客户端的请求在等待，那么继续运行服务器线程是合理的，可以利用缓存的预热优势执行 IPC 的接收阶段。

现代的 L4 版本为了保证实时行为的正确性，在直接进程切换符合优先级时保留该机制，否则调用调度器。事实上，直接进程切换是一种时间片捐赠（time-slice donation）的形式，而 Steinberg 等人 [2005] 表明这可以用于实现优先级继承和优先级上限协议。Fiasco.OC 和 NOVA 通过允许用户按调用设置是否捐赠时间片来支持这一点。

替代方案：seL4 中的直接进程切换受优先级控制，在 Fiasco.OC 和 NOVA 中为可选。

4.4 抢占

传统的 L4 实现会在内核执行期间禁用中断，尽管有些（如 L4/MIPS）在长时间运行的操作中设置了抢占点，在这些点会短暂启用中断。这种做法能显著简化内核实现，因为大多数内核部分无需并发控制，通常能带来更好的平均性能。

然而，原始的 L4 ABI 包含一些运行时间较长的系统调用，早期的 Fiasco 工作将内核做成了完全可抢占，以提升实时性能 [Hohmuth 和 Härtig, 2001]。后来版本的 ABI 移除了大部分长时间运行的操作，Fiasco.OC 又回到了最初的、基本不可抢占的方式。

对于 seL4 来说，不可抢占内核还有一个额外的理由：避免并发使形式化验证更可行 [Klein 等人, 2009]。鉴于 seL4 针对的是许多属于硬实时系统的安全关键系统，我们需要对中断响应延迟有严格的上限。因此必须避免长时间运行的内核操作，并在无法避免的情况下（例如对象删除这种几乎无法界定时间的操作）使用抢占点。我们在抢占点的设置以及最小化其需求的数据结构和算法设计上投入了大量精力 [Blackham 等人, 2012]。

需要注意的是，基于 continuation（延续）的事件内核天然支持抢占点（通过将其作为 continuation 点来实现）。

4.5 不可移植性

Liedtke [1995] 指出，微内核的实现不应追求可移植性，因为硬件抽象层会带来额外开销，并掩盖对特定硬件的优化机会。他还举例说即便是“兼容”的 i486 和 Pentium 处理器之间也有微妙的架构差异，导致最优实现方式出现显著变化。

但这个论点被 Liedtke 自己推翻了，他开发了高性能但可移植的 Hazelnut 内核，尤其是 Pistachio。通过仔细的设计和实现，使得最终版本的实现有 80–90% 与架构无关。

在 seL4 中，架构无关的代码（在 x86 和 ARM 之间）仅占约 50%。大约一半的代码用于虚拟内存管理，这部分本质上就是架构相关的。架构无关

代码比例较低的原因是 seL4 整体体积较小，大部分（架构无关的）资源管理代码被移到了用户态。除了 IPC 快路径，架构相关的优化也很少。Steinberg [2013] 估算，将 NOVA 移植到 ARM 也需重写大约 50% 的代码。

4.6 非标准调用约定

原始的 L4 内核完全由汇编语言实现，因此函数的调用约定在内核内部并不重要。在 ABI 层面，所有非系统调用参数使用的寄存器都被指定为消息寄存器。库接口使用内联汇编生成函数封装，将编译器的调用约定转换为内核的 ABI（希望编译器能优化掉这些转换的开销）。

下一代 L4 内核（从 L4/MIPS 开始）至少部分使用 C 语言实现。在进入 C 代码时，这些内核必须重新建立 C 编译器的调用约定，返回时再切换回内核约定。这使得调用 C 函数代价较高，因此只在处理本身开销较大的操作时才使用 C。

后来的内核几乎完全用 C（Hazelnut）或 C++（Fiasco、Pistachio）编写。调用约定不匹配的成本（以及缺乏 Liedtke 式地“虐待自己”以微优化每一行代码的意愿）导致这些 C 代码的性能比不上旧的汇编内核。因此这些内核的开发者开始引入手写的汇编快速路径，从而获得与原始 L4 可比的 IPC 性能（见表 1）。

这种传统方式并不适用于 seL4，因为其验证框架只能处理 C 代码 [Klein 等人, 2009]，而我们希望尽可能全面地验证内核功能。这要求将汇编代码限制到最小，并避免调用约定转换，迫使我们采用工具链的标准调用约定。

4.7 实现语言

seL4 也极度依赖 fast-path 代码来获得具有竞争力的 IPC 性能，但这些 fast-path 现在必须用 C 语言实现。由于汇编代码的高维护成本，汇编 fast-path 已经在商业的 OKL4 内核中被废弃，因为在商业环境中，这种成本超过了任何性能上的收益。对于 seL4，我们容忍的 IPC 性能下降不超过 10%，相对于在相同架构上最快的内核。

幸运的是，事实证明，只要精心手工构建 fast-path，就能实现高度竞争力的 IPC 延迟 [Blackham 和 Heiser，2012]。具体而言，这意味着手动重新排列语句，并利用编译器无法通过静态分析得出的（已验证的）不变量。

实际上，在 ARM11 处理器上，一个单向 IPC 的最终延迟为 188 个周期，比我们在同一硬件上测量到的任何其他内核的最快 IPC 性能还要好 10%。这部分归功于简化的 seL4 ABI 和 IPC 语义，以及事件驱动内核不再需要在堆栈切换时保存和恢复 C 调用约定。此外，我们还得益于改进的编译器，特别是它们对常见情况条件分支进行注解的支持，这有助于提升代码局部性。

无论如何，这一结果表明，汇编实现已不再具有性能优势。德累斯顿的团队事实上发现，即使不使用 fast-path，他们也能实现具有竞争力的性能。

第一个完全用高级语言编写的 L4 内核是 Fiasco，它选择了 C++ 而非 C（后者早些年用于 MIPS 内核的部分实现）。鉴于当时 C++ 编译器的状况，这一决定看似大胆，但至少部分原因是 Fiasco 最初并未以性能为目标。后来情况发生了变化，最近的 Fiasco 经验证明，只要使用得当，C++ 代码不会造成性能损失。

卡尔斯鲁厄团队也为 Pistachio 选择了 C++，主要是为了支持可移植性。尽管德累斯顿和卡尔斯鲁厄对 C++ 抱有高度热情，但我们从未看到 C++ 在微内核实现中提供了任何令人信服的优势。此外，OK Labs 发现，在嵌入式领域中，优秀的 C++ 编译器难以获得，因此他们将微内核版本转回为纯 C。

对于 seL4，形式化验证的需求迫使我们选择 C。尽管德累斯顿的 VFiasco 项目试图验证用 C++ 编写的内核 [Hohmuth 和 Tews，2005]，但它从未完成对 Fiasco 所用 C++ 子集语义的形式化。相反，通过用 C 实现 seL4，我们能够基于已有的 C 语言形式化 [Norrish，1998] 构建，这是实现验证的关键推动力。

5 结论

很少有一个研究型操作系统同时拥有庞大的开发者社区、显著的商业部署，并经历了长期的演进。而 L4 正是这样一个系统：其 API、设计与实现原则已演进了 20 年，期间还诞生了十多个从零开发的实现版本。我们认为这是一个极佳的机会，去反思那些经受住时间考验的设计原则与技术诀窍，以及哪些理念因更深入的理解、应用场景的变化和 CPU 架构的演进而被淘汰。

在这期间，许多设计选择与实现技巧曾被采纳，也有不少被放弃，其中一些甚至是最初设计者尤为珍视的。然而，L4 背后最核心的通用原则仍然是最小化设计（包括让设备驱动在用户态运行）和对性能的高度关注。这两个原则至今仍在开发者心中占据首要地位。尤其值得注意的是，作为微内核关键性能指标的 IPC 延迟（进程间通信延迟），在不同的指令集架构和微架构之间对比时，其时钟周期基本保持不变，这与 Ousterhout 在 L4 出现前不久所观察到的性能趋势形成鲜明对比。更令人惊讶的是，L4 的代码规模基本保持稳定，这在软件系统中是极其罕见的现象。

形式化验证的引入进一步强化了“最小化”这一核心理念，并推动了实现的简化。许多设计决策，如简化的消息结构、用户级控制内核内存、多核处理策略等，都深受可验证性的影响。同时，它也改变了一些实现方法，例如使用事件驱动的内核结构、采用标准的调用约定、选择 C 语言作为实现语言。然而，我们并不认为这些为了可验证性而做出的取舍在忽略验证需求的前提下是更差的选择。

正是通过形式化验证，L4 兑现了微内核当年最核心的承诺之一：健壮性。我们认为，L4 在保持其初衷的同时实现了这一目标，这本身就是对 Liedtke 原始设计思想的最大肯定。这也许花费了非常漫长的时间，但最终，Brinch Hansen 在 1970 年提出的那些曾被视为激进的想法被时间所证实。

不过，有一个领域至今仍未找到令人满意的抽象方式：时间。当前的 L4 内核仍然实现了一种具体的调度策略——通常是基于优先级的轮转调度，这是目前内核中唯一剩下的策略性内容，也正是 L4 通用性受限的最大因素。然而，Dresden 和 NICTA 的研究表明，一个可参数化的通用调度器有可能支持所有标准的调度策略。我们相信，这一次不需要再等上 20 年了。