开源操作系统的 LoongArch 移植
—seL4 微内核

项目成员：刘庆涛　雷洋　陈洋

指导老师：张福新　高燕萍

学　　校：中国科学技术大学

全国大学生计算机系统能力大赛

操作系统赛

功能挑战赛道

2022 年 8 月

摘要

本项目为 2022 全国大学生计算机系统能力设计大赛-操作系统设计赛功能挑战赛道的 97 号赛题：la-seL4：开源操作系统的 LoongArch 移植-seL4 微内核。

seL4 是经典 L4 系列微内核的一种实现，有着高安全性、高性能等特点，经常被用作可靠安全系统的基础部件。它采用了基于 capability 的访问控制方式，还通过了严格的形式化验证。seL4 官方已经实现 x86，arm 和 RISC-V 版本的 seL4 微内核。为促进操作系统教学，推动龙芯生态建设，扩大 seL4 开源社区影响力，本团队在 LoongArch 平台上移植了 seL4 微内核和相关程序库。

项目主要工作：

1. 移植 elfloader 内核引导程序。配置虚拟内存相关寄存器，初始化 TLB，建立内核临时页表，引导微内核启动。

2. 移植 seL4 微内核。结合 LoongArch 指令集、内存管理、中断与例外规范和 seL4 内核特点设计并实现 LoongArch 版本 seL4 微内核。

3. 移植用户测试程序。程序通过 seL4test-driver 加载 seL4test-test，然后在系统调用、IPC、进程管理、调度等方面测试内核功能。微内核已通过 seL4-test 的 15 个测试样例。

4. 移植 Cmake 编译框架。使用龙芯交叉编译工具把 elfloader、kernel 和用户程序编译为可执行 elf 程序，然后使用龙芯 qemu 和 gdb 调试并优化程序。

5. 移植自动化测试程序库。首先移植测试程序依赖的 docker 镜像，然后移植 compile workflow、C parser workflow、CI workflow 自动化测试程序并测试微内核功能。

6. 开源全部资料。分享项目文档、代码注释、docker 镜像等资源，整理龙芯文档和 seL4 文档，以便学习和交流。

本文分为六个章节：

第一章，项目背景。本章介绍项目背景，L4 发展历史，seL4 微内核特点，分析在 LoongArch 平台上移植 seL4 微内核和相关程序库的意义。

第二章，seL4 简介。本章调研 seL4 微内核设计理念，介绍 seL4 的设计模式，简单分析基于 seL4 的现有项目。

第三章，项目特点和难点。本项目特点：项目涉及知识面广，移植仓库数量多。本项目难点：工程量大，seL4 架构相关资料少，龙芯学习资料少。

第四章，seL4-test 的 LoongArch 移植。本章首先介绍 LoongArch 平台上移植 seL4 微内核和相关程序库的具体工作：内存管理，用户上下文和通信消息，中断与例外，Cmake 编译框架，自动化测试程序（github workflow）和其他移植工作。然后展示程序的运行结果。

第五章，开源资源分享。本章介绍该开源项目的获取和使用方式，整理龙芯文档和 seL4 官方资源，以便交流学习。

第六章，项目回顾与展望。本章梳理了项目开发思路，回顾了初赛和决赛阶段的进展，并规划了后续移植和优化工作。

目录

图目录

表目录

第1章 项目背景

本章介绍项目背景，L4 发展历史，seL4 微内核特点，分析在 LoongArch 平台上移植 seL4 微内核和相关程序库的意义。

1.1 L4 微内核家族

微内核最小化了内核提供的功能。微内核仅提供一组服务机制，而实际的操作系统服务由在微内核基础上构建的用户模式服务器提供[1]。应用程序通过进程间通信（IPC）机制（通常是消息传递）与服务器通信来获取系统服务。所以，IPC 是调用任何服务的关键基础，降低 IPC 成本非常重要。

20 世纪 90 年代，IPC 性能已经成为微内核的致命弱点：一次正常的单向通信成本约 100 μs，这对于构建高性能系统代价过于沉重，同时也导致了微内核设计模式逐渐衰落[2]，甚至有人认为这是微内核设计的缺陷[3]。

但是在 1993 年，Liedtke 就用他的 L4 内核[4]证明了通过恰当的设计，微内核的 IPC 可以比当代的微内核快 10 到 20 倍。在 L4 基础上的半虚拟化 Linux 演示也证实了 L4 极佳的性能[5]。除此之外，还有学者实现了对 L4 全面的形式验证，包括微内核功能的正确性证明和二进制执行程序级别的高安全性完整证明[6]。L4 对其他的研究工作也产生了较大的影响，如 EROS[7]。

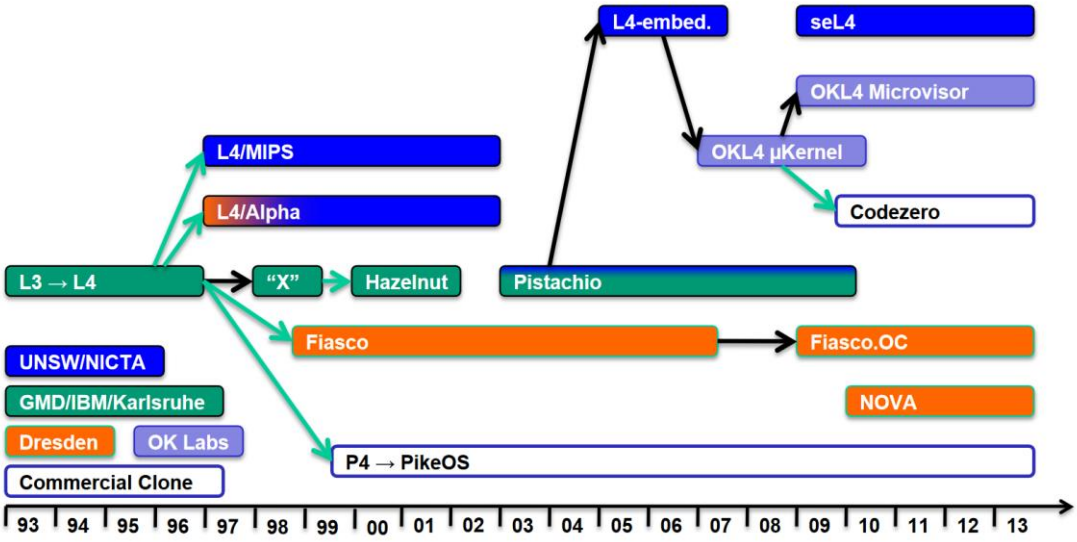

图 1-1 L4 家族

L4微内核家族经历了20年的使用和发展,产生了很多迭代版本(图1-1[11]),已经拥有大量用户和活跃的开发者社区,并且在安全关键商业系统进行了大规模应用。

1.2 seL4 微内核的发展

由于人们对操作系统安全性的关注日益密切，受 EROS 和 KeyKOS[8]系统设计模式的启发，L4 采用 capabilities[9]的概念实现了访问控制。进一步地，为了对支持访问控制的 L4 进行形式化验证, Gernot Heiser 等人在 2004 年着手开始实现基于 capabilities 的 L4 微内核——seL4。

2009 年，seL4 团队完成了引导程序以外代码的形式化验证；2011 年，证明了 seL4 内核能保证数据完整性，即在没有 capabilities 授权的情况下不能修改数据，同时也分析了 seL4 的最差情况执行时间，证明了 seL4 是混合临界系统（mixed-criticality systems、MCS），即在同时运行可靠和不可靠代码的情况下，可以保证关键可靠代码的硬实时性,可信代码实时性不会被非可信代码的异常行为破坏；2013 年证明了 seL4 在 separation-kernel 配置下可以保证数据保密性（confidentiality），即在没有 capabilities 授权的情况下不能读取数据（但是证明过程没有考虑时间，不能杜绝 timing channel）；2013 年证明了 seL4 产生的二进制代码是 seL4 C 语言子集的正确翻译，从而不必依赖可信 C 编译器。在 2014 年，seL4 源代码开源[12]；2020 年 4 月 7 日，seL4 加入 Linux 基金会。

seL4 是一个高安全性、高性能的操作系统微内核,也是世界上最先进最可靠的操作系统内核。它的独特之处在于其全面的形式验证，而不影响性能，经常被用作可靠安全系统的基础部件[10]。seL4 通过 capabilities 提供细粒度的访问控制,并控制系统组件之间的通信——这是系统最关键的部分,它以特权模式运行。作为微内核, seL4 被简化为系统最小核心,可以为服务于不同应用场景的任意系统提供可靠的基础。seL4 为运行中的应用程序提供了最高的隔离保证,且在数学上被证明了正确性。

seL4 是 L4 家族发展 20 年研究的成果，而且还在持续发展中。seL4 设计原则主要有以下几点[13]:

1. 可验证性

可形式化验证是 seL4 最主要的出发点。

2. 极简性

这也是 L4 微内核的核心原则，而且因为 seL4 的验证成本几乎与代码量的平方成正比，所以极简性原则对 seL4 也十分重要。

3. 通用性

这是 L4 乃至所有微内核的最初设计目标：成为其他系统设计的基础。出于通用性的考虑，极简性并没有走向极端化。

4. 自由性

极简性和通用性导致了微内核必须专注于基本机制，由此给用户模式极大的自由度。

5. 高效性

性能，尤其是 IPC 操作的性能一直是 L4 发展的核心驱动力[1]。

6. 安全性

这也是在 seL4 之前 OKL4 微内核采用 capabilities 机制的原因。安全性是核心原则，内核从根本上是为提供尽可能强的隔离机制而设计的。

7. 反常性

seL4 与其他内核的一些原则差距甚远。

1) seL4 尽最大可能不限制用户的权限，因为系统设计人员应当保证：只有被信赖的用户才能执行危险的操作。

2) seL4 不会试图构建易用的 API，因为如何方便地构建实际系统不是 seL4 的设计原则。

3) seL4 不会抽象硬件层，比如多级页表。试图将不同体系结构下不同的页表格式置于同一个抽象层会不可避免地丢失细节。

8. 平衡性

[1] seL4 团队不允许 seL4 与最佳的内核对比时的损失超过 10%，而早期的 seL4 性能损失最多为 10%，随着进一步的优化，seL4 逐渐成为性能最佳的微内核。

除安全性和可验证性外，其他原则并不是绝对的。seL4 的各个原则也会存在冲突的情况，这就需要设计者有所取舍，比如，可验证性和高效性之间的取舍。

1.3 项目意义

世界早已步入智能设备时代，硬件+软件的产业结构对于基于芯片的智能设备设计、生产和推广都至关重要。以 windows+intel x86 架构生态体系为代表的产业，其基本在 PC 市场内形成了垄断。在此垄断的背景下，2021 年新推出的龙芯架构要想站稳脚跟，必须发展基于 LoongArch 的软件生态。

作为国产芯片的象征，龙芯自创立以来就坚持自主研发，在二十余年的 CPU 研制和生态建设积累经验的基础上融合吸收 x86、ARM 等架构的特点，推出了龙芯指令系统 LoongArch。自此，龙芯生态的硬件部分基本成型，但软件生态的发展较为落后。将各类软件移植到 LoongArch 指令集架构是龙芯走向开放市场的一个重要切入点。

操作系统软件作为应用软件和硬件的桥梁，对于龙芯生态的发展起着很大作用。除了官方的 Loongnix 和 LoongOS 外，来自第三方的统信操作系统、麒麟操作系统、龙蜥操作系统目前也已经在 LoongArch 架构上成功移植。除此之外，还有更多操作系统等待着被移植到 LoongArch 架构之上。seL4 作为目前最优秀的微内核，在未来的应用前景十分广大。将 seL4 移植到 LoongArch 架构，对于推动龙芯生态的建设，提高龙芯生态的影响力有很大的帮助。

第2章 seL4简介

本章调研 seL4 微内核设计理念，介绍 seL4 的设计模式，简单分析基于 seL4 的现有项目。

2.1 seL4 是什么[2]

2.1.1 seL4 与主流操作系统的区别

seL4 作为典型的微内核，与主流操作系统，如 Linux 的区别十分明显。

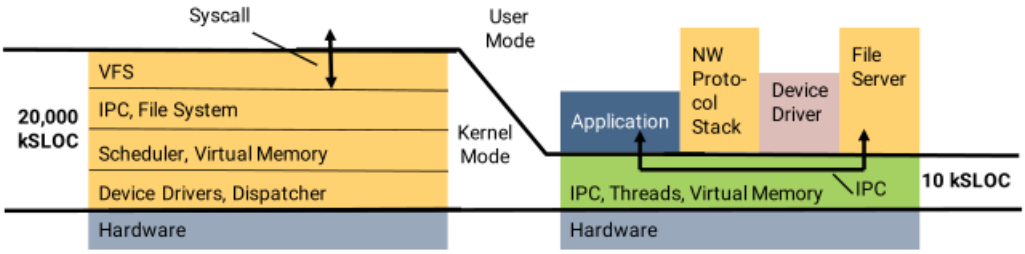

图 2-1 微内核和宏内核[15]

图 2-1 左侧为主流操作系统的框架，其中的黄色部分代表操作系统内核，为应用程序提供文件存储、网络等服务。实现这些服务的所有代码都将在特权模式(内核模式)运行。应用程序将在非特权模式(用户模式)下运行,不能直接访问某些硬件资源。操作系统在内部有多个层级，每一层都对底层进行抽象，并为上层提供服务。这样宏内核模式中，如果特权态代码有一个漏洞被发现，攻击者就可以利用该漏洞在特权模式下运行恶意代码。

Linux 内核包含大约 2000 万行源代码（20 MSLOC），里面的漏洞成千上万[16]。因为 Linux 有一个庞大的可信计算库 (TCB),它被定义为整个系统的子集，该库只有在被完全信任的情况下才能正确运行，由此实现整个系统的安全。

出于以上原因，微内核减少了 TCB 的规模，以此减少受攻击的概率。图 2-1 右侧显示,绿色部分的特权模式代码相比于左侧黄色部分要小得多。在 seL4 中，

[2] 本节和 2.2 节描述的内容来自于 seL4-whitepaper[15]和 seL4-manual[17]

它的源代码量级为一万行（10 kSLOC）。这比 Linux 内核小了三个数量级，被攻击的可能性大大降低。

如此小的代码库中不可能提供和 Linux 相同的功能。微内核几乎不提供任何服务，它只是硬件的一个薄包装，但足以安全地复用硬件资源。微内核主要的功能是隔离，即作为程序执行的安全沙箱。除此之外，它提供了一种受保护的过程调用机制，由于历史原因该机制被称为 IPC。IPC 允许一个程序安全地调用不同程序中的函数，其中微内核在程序之间传输函数的输入和输出，并且接口函数只能被明确授权的客户端（被赋予适当的 capability）调用。

微内核系统使用以下的方法来提供单片操作系统在内核中实现的服务。在微内核中，这些服务只是程序，与应用程序没有区别，它们运行在自己的沙箱中，并提供 IPC 接口供应用程序调用。如果某服务受到攻击，那么这个攻击仅限于被攻击的服务，沙箱会保护系统的其余部分。这与 Linux 形成鲜明对比，Linux 服务受到攻击就会危害整个系统。有研究表明，在已知的被归类为关键（即最严重）的 Linux 漏洞中，29%的漏洞能被微内核设计完全消除，还能额外减轻 55%的安全漏洞隐患[16]。

2.1.2 seL4 也是管理程序

seL4 是一个微内核，但是同时也是一个管理程序，可以在 seL4 上运行虚拟机，并在虚拟机上运行主流操作系统，如 Linux。也就是说，除了自己提供服务以外，还可以通过虚拟化技术使用其他操作系统提供的服务。

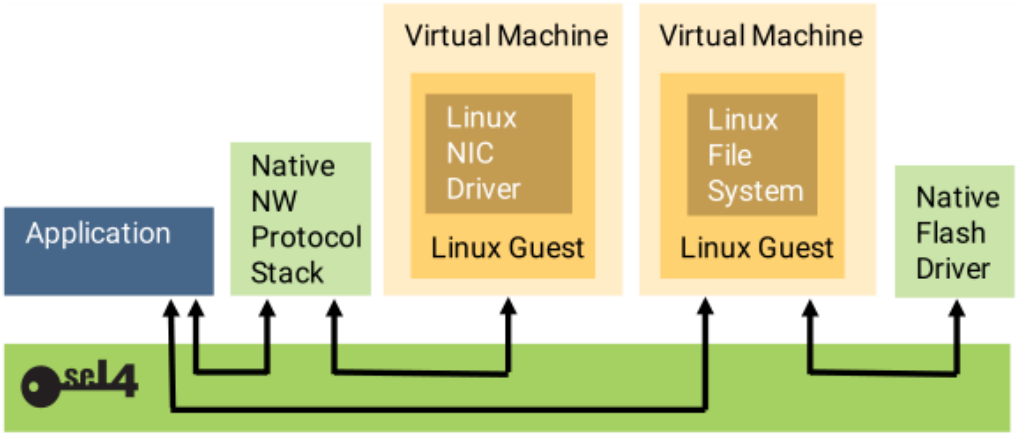

图 2-2 虚拟化技术将 Linux 提供的服务集成到本机

2.2 seL4 设计模式

由于项目关注内容有限,这里不介绍包括 seL4 的混合临界系统 MCS(mixed-criticality systems)、IPC、NOTIFICATION 等内容，如有需求请查阅 seL4-manual 手册或官方网站。

2.2.1 基于 capabilities 的访问控制

seL4 微内核提供了一个基于 capabilities 的访问控制模型。Capabilities 是向特定对象传达特定权限的访问令牌。所有的内核服务都会受到访问控制；为了执行特定操作，应用程序必须调用对所请求服务具有足够权限的 capability。通过这种机制，系统可以配置为彼此隔离的软件组件，也可以通过有选择地授予特定的 capability，在组件之间实现授权的、受控的通信。这使得软件组件隔离具有高度的安全性，因为只有那些由 capability 拥有者明确授权的操作才被允许执行。

图 2-3 capability 是传递权限的密钥

一个 Capabilitity[3]也可以看作是一个特定的内核对象(如线程控制块 TCB)的引用，类似于指针(capability 的实现通常被称为胖指针)，但是这个指针指向的地址不可变，始终引用固定的对象。除指针功能外，capability 还对访问权限进行编码，即携带着控制可能调用哪些方法的访问权限。capability 驻留在应用程序的 capability 空间中(见 2.2.3 节),此空间的地址指向一个可能包含或不包含 capability 的插槽(slot)。应用程序通过引用 capability(使用 capability 所在插槽的地址)就可以对系统对象执行操作。Capability 可以在权限空间内复制和移动，也可以通过 IPC 发送。

[3] 这里的 capability 与 Linux 中的 capability 概念不同，Linux 中的 capability 是具有系统调用粒度的访问控制列表（ACL）。

seL4 实现了 10 种引用不同对象的 capability:

1. **Endpoints:** 执行受保护的函数调用;
2. **Reply Objects:** 表示来自受保护调用过程的返回路径;
3. **Address Spaces:** 提供各种组件的沙箱(抽象硬件页表的薄包装);
4. **Cnodes:** 存储代表组件访问权限的 capability;
5. **Thread Control Blocks(TCBs):** 表示执行的线程;
6. **Scheduling Contexts:** 在内核上访问特定执行时间段的权限;
7. **Notifications:** 同步对象(类似信号量);
8. **Frames:** 表示可以映射到地址空间的物理内存;
9. **Interrupt Objects:** 提供对中断处理的访问;
10. **Untypeds:** 可以转换为任何其他类型的未使用(空闲)的物理内存。

seL4 采用 capability 作访问控制的原因为:

1. 细粒度访问控制

Capability 可以提供细粒度的访制，符合最小权限的安全原则(最小权限原则 POLA)。与传统的访问控制列表(ACL)不同，后者主要用于 Linux、Windows 等主流系统，在商业和安全系统中也有应用，如 INTEGRITY、PikeOS 等。

在 Linux 中，文件具有一组关联的访问模式位，其中的一些位决定该文件的所有者对文件操作的权限,其他位代表着文件所在组的各成员对文件操作的权限。这种粗粒度的访问控制模式存在着很多问题,比如如果用户想要运行不受信任的程序处理某些文件，但是不想让这个程序访问其他文件，Linux 完全不能胜任这种情况。虽然人们提出了可以通过容器等方案来解决这类问题，但是这类问题完全可以通过设计来避免。

Capability 提供了一种面向对象的访问控制形式，而不是和用户绑定。具体来说，当且仅当请求操作的主体提供了 capability 来授权此次操作时，内核才允许本次操作继续进行。如果执行了不受信任的程序，该程序只能访问被授予 capability 的文件。

2. 打桩和授权机制

Capability 支持打桩机制，因为 capability 是不透明的对象引用。被赋予 capability 的对象不知道 capability 引用的到底是什么，他只去调用引用对象上的方法。比如某个对象被赋予了一个引用文件的 capability，但是实际上这个 capability 是指向安全监视器的通信通道，后者又持有实际文件的 capability。当这个对象调用这个 capability 时，从对象的角度来看只是对一个文件的调用，但是实际上是由一个安全监视器检查了该对象的操作，验证合法后由监视器对文件执行的操作。

Capability 还支持授权机制。如果 A 想让 B 访问 A 的某个对象，A 就创建(mint 操作)一个新的 capability 交给 B。当然，新创建的 capability 所拥有的权限不会超出原有的范围，只可能缩小；A 也可以随时通过销毁 capability 来撤销授权。这种机制在 ACL 中很难安全地实现。

Capability 对象支持四种访问权限：Read、Write、Grant 和 GrantReply，其中 Grantreply 是 Grant 的一种较弱的形式。每一项权限的含义都是相对于各种对象类型进行解释的，如表 2-1 所示。

表 2-1 各种对象的权限

对象类型	Read	Write	Grant	GrantReply
Endpoint	Recv	Send	发送 caps	发送 reply caps
Notification	Wait	Signal	N/A	N/A
Page	页可读	页可写	N/A	N/A
Reply	N/A	N/A	发送 reply caps	N/A

2.2.2 seL4 内核服务

微内核提供了有限的服务原语。更复杂的服务可以作为应用程序用这些原语实现。这样可以不增加特权模式代码以及复杂性，还能扩展功能，从而支持各种应用程序域的大量服务。**注意，只有在内核配置为 MCS（mixed-criticality system）支持时一些服务才可用，这里不介绍这类服务。**

seL4 在非 MCS 配置下提供以下基本服务：

1. **Threads：**线程是支持软件运行的 CPU 执行的抽象;
2. **Address spaces:** 地址空间是虚拟内存空间，每个地址空间包含一个应用程序。应用程序只能访问地址空间中的内存;
3. **Inter-process communication：**通过 Endpoints 的进程间通信(**IPC**)允许线程使用消息传递进行通信;
4. **Notifications：Notifications：**提供了一种类似于二进制信号量的非阻塞信号机制。
5. **Device primitives：**设备原语允许将设备驱动程序实现为非特权应用程序。内核通过 IPC 消息导出硬件设备中断;
6. **Capability spaces：**Capability 空间存储对内核服务的权限(即访问权限)及其记录信息。

2.2.3 Capability 空间

seL4 实现了一个基于 capability 的访问控制模型。每个用户空间线程都有一个相关的 capability 空间(CSpace)，它包含线程拥有的 capability，从而控制线程可以访问哪些资源。capability 驻留在称为 Cnode 的内核管理对象中。CNode 是一个插槽(Slot)组成的表，每个槽可以包含一个 capability，也可以包括另一个 Cnode 的 capability，形成一个有向图。从概念上讲，一个线程的 CSpace 是有向图中从 CNode 的 capability 开始可到达的部分，CNode capability 是它的 CSpace 的根 capability。

CSpace 地址指的是一个单独的槽位(在 CSpace 中的某些 CNode 中)，它可能包含也可能不包含某个 capability。线程在它们的 Cspace 中引用 capability (例如进行系统调用时)，使用存放该 capability 的槽的地址。CSpace 中的地址是 CNode capability 的索引的连接，形成了到目的槽的路径。

Capability 可以在 Cspace 中复制和移动，也可以在消息中发送。此外，新 capability 可以从旧 capability 的权利子集中创建出来。seL4 维护了一个 capability 派生树(CDT)，它在其中跟踪这些复制的 capability 与原始 capability 之间的关系。revoke 方法删除所有从选定 capability 派生出来的 capability (在所有 Cspace 中)。

这种机制可以被服务器用来恢复对客户端可用的对象的唯一 capability，也可以被无类型化内存(untyped memory)的管理者用来销毁内存中的对象，以便 retype。

seL4 要求程序员从用户空间管理所有内核数据结构，包括 CSpaces。这意味着用户空间程序员要负责构造 Cspace 以及在 Cspace 中对 capability 进行寻址。更详细的内容查阅官方文档。

2.3 基于 seL4 的项目

seL4 官方提供了一个单独的技术文档网站 Docsite[14]，其内容包含了 seL4 的基础知识、内核和核心库的 API、用户级框架等。新南威尔士大学的高级操作系统课程也提供了一个基于 seL4 的作业，教授如何在 seL4 上构建操作系统。

seL4 官网提供了 seL4-tutorials 和 seL4-test 项目，前者用于学习 seL4 设计思想和技术细节，后者用于测试 seL4 是否可以在机器上成功运行。经过综合考虑，本团队拟在 LoongArch 平台移植 seL4-test 项目，该项目的测试程序能够评估微内核是否移植成功。另外，seL4-test 项目由 seL4 微内核库和相关程序库组成，项目扩展性强，团队拟首先移植 seL4 微内核，再移植相关程序库，例如简单示例程序、seL4-tutorials 等项目。

第3章 项目特点和难点

本项目不仅要求学习 seL4 设计思想和原理，还需要学习龙芯指令集、交叉工具链和 qemu 使用方法，综合运用操作系统、汇编语言、软件工程等专业知识设计具体移植方案。

seL4 有 55 个官方仓库，包含微内核、测试程序、形式化验证等仓库。为实现 seL4 微内核移植和用户程序测试，本团队 fork 并修改了其中 10 个官方仓库，还使用了官方的 seL4_projects_libs 和 projects_libs 这 2 个仓库，这些仓库的介绍如下表 3-1 所示。

表 3-1 项目仓库介绍

仓库名	仓库介绍
la-seL4	seL4 微内核代码
la-sel4test	seL4 测试代码（用户空间程序）
la-seL4_tools	seL4 构建工具，如 cmake，elfloader 等
la-musllibc	轻量级 C 语言库
la-sel4runtime	运行 C 语言兼容程序最小 runtime 系统
projects_libs	用户程序库（使用官方仓库）
seL4_projects_libs	用户程序库（使用官方仓库）
la-util_libs	用户程序库
la-seL4_libs	用户程序库
la-seL4-ci-actions	自动化测试程序仓库
la-l4v	seL4 形式化证明工具仓库
la-seL4-CAmkES-L4v-dockerfiles	docker 镜像构建仓库

项目难点：①工程量大。seL4 官方有 55 个仓库，编写或运行程序往往需要关联多个仓库。团队工作不仅涉及 elfloader、cmake 等程序或工具的移植，还涉及微内核、测试程序、形式化验证等相关仓库的移植。②seL4 移植资料少。相比于 xv6 教学操作系统，seL4 的官方指导资料和技术博客较少，官方只提供了架构无关的设计思想，团队需要充分理解 riscv 等架构代码才能移植 seL4 到龙芯平台。③龙芯资料少。龙芯指令集在 2021 年发布，教学资料和技术博客较少，可供参考示例程序不多。

第4章 seL4-test的LoongArch移植

本章主要介绍将 seL4-test 移植的过程中与 LoongArch 架构密切相关的设计工作，包括内核设计、cmake 设计、自动化测试程序移植和其他设计工作。本章最后一节还介绍了程序运行情况。

4.1 内核设计

4.1.1 内存管理相关设计

龙芯架构的 MMU 支持两种虚实地址翻译模式：直接地址翻译模式和映射地址翻译模式，通过配置 CSR.CRMD 来决定 MMU 的工作模式。

在直接地址翻译模式下，物理地址默认等于虚拟地址的[PALEN-1:0]位（不足补 0），处理器复位结束后将进入直接地址翻译模式，seL4 微内核加载时只有起始地址处几条指令是在此模式下执行的。

在映射地址翻译模式下，具体又可以分为直接映射地址翻译模式（简称“直接映射模式”）和页表映射模式，具体模式在后续章节介绍，此模式下翻译地址时优先按照直接映射模式进行翻译，否则按照页表映射模式进行翻译。在 seL4 中只有在启动阶段的 elfloader 中使用了直接映射窗口，其他部分都使用页表映射模式。

4.1.1.1 虚拟地址结构

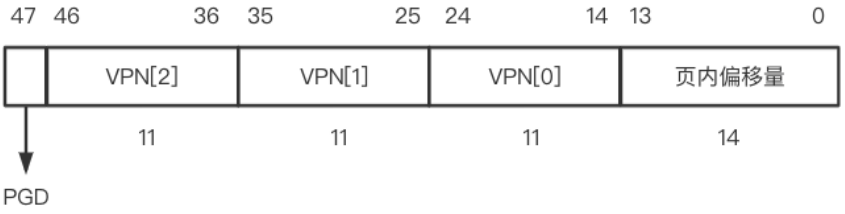

图 4-1 虚拟地址结构

LoongArch 物理地址空间范围是：$0\sim2^{PALEN}-1$。在 3A5000 上 PALEN=48。

页表最顶层目录的基地址 PGD 需要根据被查虚地址的第（PALEN－1）位决定。当该位为 0 时，PGD 来自于 CSR.PGDL;当该位为 1 时，PGD 来自于 CSR.PGDH。

在 3A5000 上虚拟地址共 48 位，最高位为 PGD。本文设计了三级页表，每一级页表索引位为 11 位，页内偏移量为 14 位。也就是说，基本页大小为 16KB，可用的大页为 32MB 或 64GB。

4.1.1.2 虚拟地址空间

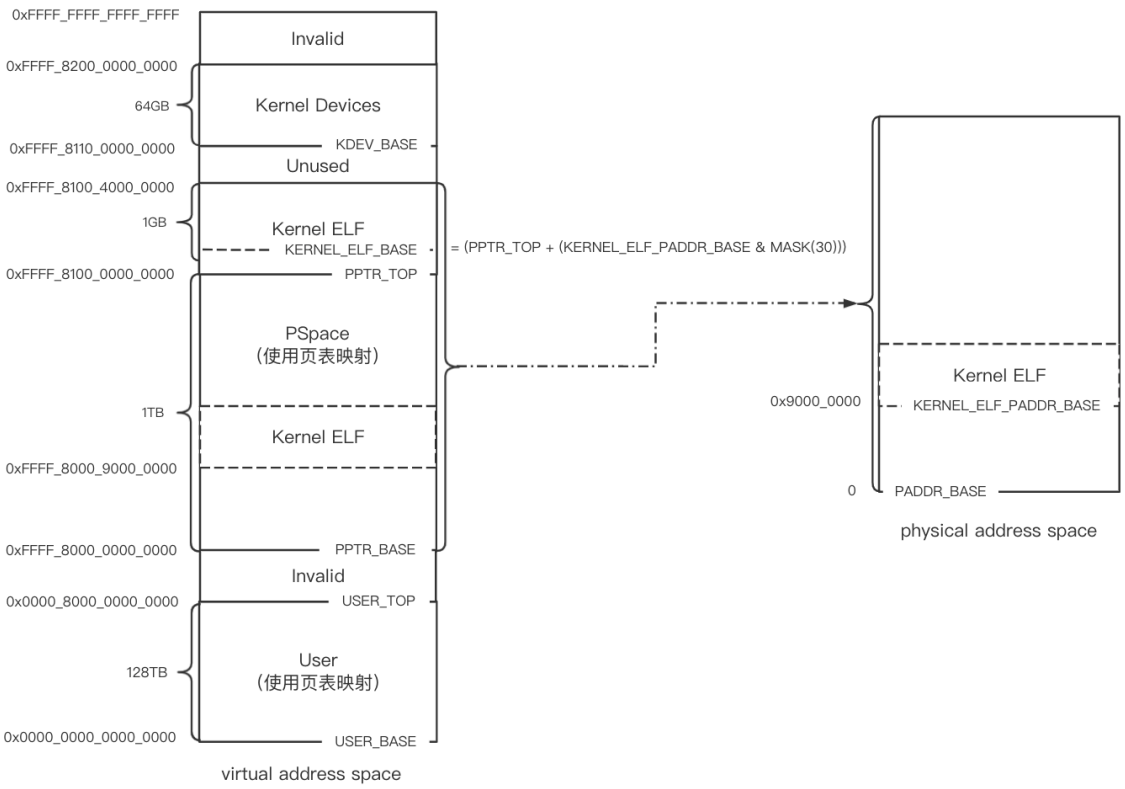

图 4-2 虚拟地址空间

LoongArch 虚拟地址映射的两种方式：

- 直接映射窗口

直接映射窗口 VESG 对应虚拟地址[63:60]位，访问时先匹配 VESG 域，若命中，取[PALEN-1:0]位转换得到物理地址。

- 页表映射

采用页表映射高位必须和 PALEN - 1 位相同，即符号拓展。这是 seL4 使用的主要虚拟地址映射方式。

1） 用户空间

当 PGD = 0 时，即为用户空间，大小为 128TB。seL4 作为微内核，用户虚拟空间由用户级代码自行管理。

2) 内核空间

当 PGD = 1 时，为内核空间。其中：

0xffff_8000_0000_0000~0xffff_8100_0000_0000 位 1TB 的物理地址空间映射窗口（PSpace），和物理地址空间一一对应，因为 3A5000 芯片+7A1000 桥片的物理地址空间范围为 0x0~0xfc_ffff_ffff，因此采用 1TB 的窗口足够包含整个物理地址空间。PSpace 采用 64GB 大页做页表映射。

0xffff_8100_0000_0000~0xffff_8110_0000_0000（Kernel ELF）这 64GB 虚拟空间（刚好为一整个二级页表空间的大小）用来做 Kernel Image 的映射，实际只使用了其中的 1GB，剩余保留可做未来拓展使用。这部分采用 32MB 大页做页表映射。这里需要注意的是，PSpace 因为映射了整个物理地址空间，因此 PSpace 中有一段和这里的 Kernel ELF 映射了相同的一段物理空间，其中装载了 Kernel Image。之所以要在 PSpace 以外另外开辟一段虚拟空间单独做映射是为了适应像 KernelImageClone 这样的新 API，其功能是在不同物理内存区域复制 Kernel Image，这样 PSpace 中的 Kernel ELF 段也会随着移动，而在 PSpace 之外单独开辟的 Kernel ELF 段保证了虚拟空间的唯一性。

0xffff_8110_0000_0000~0xffff_8200_0000_0000（Kernel Devices）用来映射内核设备。内核加载过程中，通过解析 dtb（得到对应的数组为 kernel_device_frames）对需要用的设备 io 地址区域做页表映射。seL4 作为一个微内核，遵循的是极简主义，内核没有多余的设备驱动程序，只需要中断控制器和计时器，此外我们还需要串口做输出。

4.1.1.3 内核装载流程

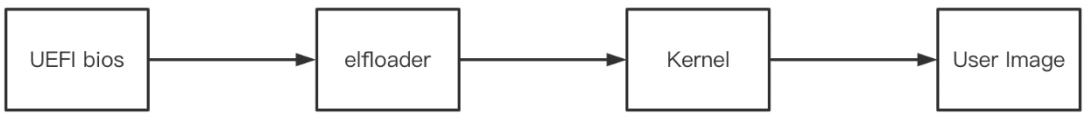

图 4-3 seL4 在 LoongArch 上的内核启动流程

UEFI BIOS 装载内核时，从内核 elf 文件获取入口点地址，然后使用抹去高 32 位的地址。例如，elf 链接的地址是 0x9000_0000_0103_4804，实际 BIOS 跳转的地址将是 0x103_4804，代码装载的位置也是物理内存 0x103_4804。BIOS 逻辑上相当于用物理地址去访问内存，高的虚拟地址空间没有映射不能直接用。

编译生成最终运行在 3A5000 上的 elf 文件是 sel4test-driver-image-loongarch-3A5000，其实质上是 elfloader，而把 kernel image、dtb 和 user image 用 cpio 打包作为 elfloader 的数据部分，生成了最终的 sel4test-driver-image-loongarch-3A5000，因此我们将此作为最终运行的 image 实际上是运行 elfloader，目前其装载地址定位 0x0200_0000（参考图 4-4），先在用物理地址运行，然后配置了一个 0x9000 开头的直接映射窗口，将地址跳转到直接映射窗口中继续运行，然后开始解压其数据中的 cpio 包，逐个读出 kernel image、dtb 和 user image，根据各自的 elf 头信息装载到特定的物理内存中，然后做了一个 kernel image 部分的临时页表映射，初始化 tlb 后跳转到 kernel 的虚拟地址入口开始运行 kernel，本文将 kernel 的装载地址定位 0x9000_0000（参考图 4-4），dtb 和 user image 随后依次装载。

7A MEM 高地址空间	MEM_UP_LIMIT ~ 0xfc,ffff,ffff
内存高地址空间	0x8000,0000 ~ MEM_UP_LIMIT
7A MEM 低地址空间	0x4000,0000 ~ 0x7fff,ffff
处理器配置空间	0x3000,0000 ~ 0x3fff,ffff
保留	0x2000,0000 ~ 0x2fff,ffff
处理器低速设备空间	0x1f00,0000 ~ 0x1fff,ffff
处理器 LPC MEM 空间	0x1c00,0000 ~ 0x1dff,ffff
7A I/O 空间和配置空间	0x1800,0000 ~ 0x1bff,ffff
7A 设备固定地址空间	0x1000,0000 ~ 0x17ff,ffff
内存低地址空间	0x0000,0000 ~ 0x0fff,ffff

图 4-4 龙芯 3A5000 处理器+7A1000 桥片地址空间划分

4.1.1.4 TLB 的使用

1） LoongArch 的 TLB

龙芯架构下 TLB 分为两个部分，一个是所有表项的页大小相同的单一页大小 TLB（Singular-Page-Size TLB，简称 STLB），采用多路组相联的组织形式；另一个是支持不同表项的页大小可以不同的多重页大小 TLB（Multiple-Page-Size TLB，简称 MTLB），采用全相联查找表的组织形式。在虚实地址转换过程中，STLB 和 MTLB 同时查找。

2） seL4 实现中的 TLB 相关配置

在 seL4 内核中，本文使用了 64GB 和 32MB 两种规格的页，同时用到 STLB 和 MTLB，其中 32MB 页映射的是 Kernel Image 和 Kernel Devices，64GB 页映射的是 PSpace（物理地址空间映射窗口）。本文将 CSR.STLBSIZE 配置为 0x19，这样保证 32MB 的页直接进入 STLB，不需要管 TLB 表项中的 PS 域；CSR.TLBREHI 的 PS 域设置为 64GB，使得 64GB 的页进入 MTLB，填入 TLB 表项的 PS 域值为 0x24。此外还要设置 CSR.TLBRENTRY 中 TLB 重填例外的处理函数入口地址。

3） TLB 重填例外的实现

见 4.1.3.2

4.1.2 用户上下文和通信消息设计

4.1.2.1 用户上下文设计

seL4 微内核中，线程由线程控制块对象（Thread Control Block，TCB）描述。TCB 定义了线程的状态，优先级，可用时间片长度，用户上下文等信息。不同架构的用户上下文信息需要参考架构特点进行特殊设计。对于 LoongArch 架构，本文定义用户上下文信息为这样的集合：{通用寄存器，特殊控制状态寄存器，seL4 特殊字段}

seL4-LoongArch 架构相关的用户上下文如下表 4-1 所示。

表 4-1 seL4-LoongArch 用户上下文信息描述

寄存器或字段	seL4 别名	功能描述
r1/ra	LR	返回地址
r2/tp	TP,TLS_BASE	线程指针

r3/sp	SP	栈指针
r4/a0	capRegister,badgeRegister	传参、返回值寄存器
r5/a1	msgInfoRegister	传参、返回值寄存器
r6/a2	—	传参寄存器
r7/a3	—	传参寄存器
r8/a4	—	传参寄存器
r9/a5	—	传参寄存器
r10/a6	replyRegister	传参寄存器
r11/a7	—	传参寄存器
r12/t0	nbsendRecvDest	临时寄存器
r13/t1	—	临时寄存器
r14/t2	—	临时寄存器
r15/t3	—	临时寄存器
r16/t4	—	临时寄存器
r17/t5	—	临时寄存器
r18/t6	—	临时寄存器
r19/t7	—	临时寄存器
r20/t8	—	临时寄存器
r21	—	保留
r23/s0	—	静态寄存器
r24/s1	—	静态寄存器
r25/s2	—	静态寄存器
r26/s3	—	静态寄存器
r27/s4	—	静态寄存器
r28/s5	—	静态寄存器
r29/s6	—	静态寄存器
r30/s7	—	静态寄存器
r31/s8	—	静态寄存器
r22/s9	—	栈帧指针/静态寄存器

csr_era	FaultIP	例外程序返回地址
csr_badvaddr	—	出错虚地址
csr_prmd	—	例外前模式信息
csr_euen	—	扩展部件使能
csr_ecfg	—	例外配置
csr_estat	—	例外状态
NextIP	—	例外返回的地址

表 4-1 中，r1～r31 寄存器是 LoongArch 的通用寄存器，以“csr_”为前缀的寄存器是控制状态寄存器,NextIP 是 seL4 的特殊字段,保存例外返回的地址。seL4 微内核的架构无关部分使用别名来操作 LoongArch 寄存器，本文结合寄存器功能和别名含义，设计出别名和寄存器映射关系如第二列所示，各寄存器或字段的功能如第三列所示。

4.1.2.2 通信消息设计

seL4 的线程间通信消息可以划分为普通消息和错误消息。

- 普通消息

msgRegisters，frameRegisters 和 gpRegisters 寄存器集合是 seL4 普通消息的重要组成部分。微内核通过调用 LoongArch 架构的消息接口来操作具体寄存器。

seL4 的 msgRegisters 寄存器集合用于传递消息，例如，线程间使用该寄存器集合通信。结合 LoongArch 架构寄存器功能，本文设计 msgRegisters 集合为：

msgRegisters = { a2, a3, a4, a5 }

seL4 的 frameRegisters 寄存器集合存储栈帧信息。结合 seL4 特点和其他架构的设计，本文设计 frameRegisters 集合为：

frameRegisters = { FaultIP, ra, sp, s0, s1, s2,

s3, s4, s5, s6, s7, s8, s9 }

seL4 的 gpRegisters 表示其他通用寄存器集合。结合 seL4 特点和其他架构的设计，本文设计 gpRegisters 集合为：

gpRegisters = { a0, a1, a2, a3, a4, a5, a6, a7,

t0, t1, t2, t3, t4, t5, t6, t7, t8, tp }

- 错误消息

seL4 线程间传递的错误消息由 fault_messages 组成，fault_messages 是 SYSCALL_MESSAGE，EXCEPTION_MESSAGE 和 TIMEOUT_REPLY_MESSAGE 三种消息的组合。

SYSCALL_MESSAGE 是系统调用信息。结合 seL4 的 seL4_UnknownSyscall_Msg 消息特点，本文设计 SYSCALL_MESSAGE 为：

SYSCALL_MESSAGE = { FaultIP, SP, LR, a0,

a1, a2, a3, a4, a5, a6 }

EXCEPTION_MESSAGE 是例外信息。结合 seL4 的 seL4_UserException_Msg 消息特点，本文设计 EXCEPTION_MESSAGE 为：

EXCEPTION_MESSAGE = { FaultIP, SP }

TIMEOUT_REPLY_MESSAGE 是由于线程间通信超时产生的信息。结合 seL4 的 seL4_TimeoutReply_Msg 消息特点，本文设计 TIMEOUT_REPLY_MESSAGE 为：

TIMEOUT_REPLY_MESSAGE = { FaultIP, LR, SP, s0, s1, s2,

s3, s4, s5, s6, s7, s8, s9, a0,

a1, a2, a3, a4, a5, a6, a7, t0,

t1, t2, t3, t4, t5, t6, t7 }

4.1.3 中断与例外处理

4.1.3.1 LoongArch 中断与例外简介

LoongArch 架构定义了 3 个例外入口，分别是 TLB 重填例外入口（CSR.TLBRENTRY），机器错误例外入口（CSR.MERRENTRY）和普通例外入口（CSR.EENTRY）。参考 Linux（LoongArch 版）等操作系统，绝大多数例外是

TLB 重填例外或普通例外，因此本项目只填入了 TLB 重填例外入口和普通例外入口，没有处理机器错误例外。

下文介绍中断与例外处理涉及的重要寄存器。关于通用寄存器和控制状态寄存器的具体信息见本项目 readme 中的参考网址。

TLB 重填例外处理流程涉及的寄存器如下表 4-2 所示。

表 4-2 TLB 重填例外涉及的重要寄存器

寄存器名	寄存器功能
CSR.TLBRENTRY	TLB 重填例外入口地址
CSR.TLBRSAVE	供 TLB 重填例外处理程序暂存数据
CSR.PGD	当前上下文中出错虚地址对应的全局目录基址
CSR.TLBRELO0/1	TLB 指令操作时，存放 TLB 表项低位部分信息
CSR.TLBREHI	TLB 指令操作时，存放 TLB 表项高位部分信息
CSR.STLBPS	STLB 页大小信息

普通例外处理流程涉及的寄存器如下表 4-3 所示。

表 4-3 普通例外涉及的重要寄存器

寄存器名	寄存器功能
CSR.ECFG	配置例外和中断入口计算方式，局部中断使能
CSR.ESTAT	记录中断状态，例外的一二级编码
CSR.CRMD	保存当前模式信息，包括全局中断使能位
CSR.PRMD	保存例外前模式信息，包括全局中断使能位
CSR.ERA	保存例外程序返回地址。
CSR.BADV	在触发地址错误相关例外时，记录出错的虚地址
CSR.SAVE	数据保存寄存器

4.1.3.2 TLB 重填例外

- 配置 TLB 重填例外

将 TLB 重填例外入口（handle_tlb_refill）填入 CSR.TLBRENTRY。除了配置 TLB 重填例外入口，其他 TLB 相关的寄存器的配置方法见 4.1.1.4。

- TLB 重填例外（handle_tlb_refill）流程

将 t0 寄存器中的值保存到 CSR.TLBRSAVE；

将 CSR.PGD 的值读到 t0；

两个 lddir 从顶级目录开始读到大页页表项或者末级页表地址为止；

两个 ldpte 将大页页表项折半或者基本页的相邻奇偶页分别填入到 CSR.TLBRELO0 和 CSR.TLBRELO1 中；

tlbrefill 根据 CSR.TLBREHI、CSR.TLBRELO0、CSR.TLBRELO1 填入 TLB，当被填入的页表项的页大小（根据 CSR.TLBREHI 的 PS 域）与 STLB 所配置的页大小（CSR.STLBPS）相等时将被填入 STLB，否则将被填入 MTLB；

从 CSR.TLBRSAVE 恢复 t0 的值。

- TLB 重填例外返回

ertn 指令进行例外处理返回。

4.1.3.3 普通例外

- 配置和初始化

关闭全局中断，配置 CSR.CRMD.IE=0；

配置 CSR.ECFG.VS=0，即普通例外使用同一个入口（未使用向量入口的形式）；

将普通例外入口 trap_entry 写入 CSR.EENTRY，把 trap_entry 代码段定义在 k_eentry 段中，在链接脚本中将 k_eentry 段 16K 对齐；

LS3A5000 芯片例外与中断初始化：清除 CSR.ESTAT.IS 所有中断状态，配置 CSR.ECFG.LIE，使能 LoongArch 核内部 13 个中断源。使能 3A5000 的 EXT_IOIen 的 UART0 中断和 KEYBOARD 中断，配置 EXT_IOImap0 将 EXT_IOI[31:0] 中断信号路由到 1 号引脚，配置 EXT_IOImap_Core0 - EXT_IOImap_Core7，将前 8 个外部中断路由到 0 号节点类型的 0 号核上，配置

EXT_IOI_node_type0 将 0 号节点类型设置为 0 号节点，从而将要开启的 UART0 和 KEYBOARD 外部中断映射到 node 0 core 0 的内部硬中断引脚 HW1 上；

LS7A1000 桥片中断初始化：使能 UART0 和 KEYBOARD 的 INT_MASK 的中断位，配置 INTEDGE 中断触发方式为边沿触发，设置 HTMSI_VECTOR0 偏移地址为 0x202 和 0x203 的寄存器，将 2 号 UART 中断路由到扩展中断 2 号引脚，将 3 号 KEYBOARD 中断路由到扩展中断 3 号引脚。

设置各中断的状态为 IRQInactive，配置 seL4 的 IRQTimer 宏为 LoongArch 的时钟中断；

调用 cap_irq_control_cap_new() 函数创建 irq_control_cap，写入 root_cnode_cap 的 seL4_CapIRQControl=4 位置；

开启全局中断，配置 CSR.CRMD.IE=1。

- 陷入及处理

交换 t0 和 SAVE0 寄存器的值。SAVE0 寄存器的宏定义是 LOONGARCH_CSR_KS3，它指向当前线程上下文的结构体（类似于 riscv 的 sscrartch 寄存器）；

以 t0 作为基址，以用户上下文寄存器或特殊字段的顺序作为偏移值，保存用户上下文。用户上下文的寄存器顺序见表 4-1 seL4-LoongArch 用户上下文信息描述；

从用户栈切换至内核栈，即 sp 指向内核栈栈顶，内核栈栈顶位置是 kernel_stack_alloc + 1<<CONFIG_KERNEL_STACK_BITS；

提取 CSR.ESTAT.Ecode 到 s1 寄存器。①如果 CSR.ESTAT.Ecode≥64，表明当前是中断，设置 nextIP=csr.era，并跳转到中断处理部分，根据具体中断号调用对应处理函数；②如果 CSR.ESTAT.Ecode==11，表明当前是系统调用，设置 nextIP=csr.era+4，将 a7 内的系统调用号填入 a2 参数寄存器，跳转到系统调用处理部分，根据系统调用号调用对应处理函数；③当前是除中断和系统调用外的其他普通例外，设置 nextIP=csr.era，跳转到相应的普通例外处理部分，根据普通例外号调用对应处理函数；

普通例外处理后，跳转到 restore_user_context 恢复上下文。

- 恢复上下文并返回

获取当前线程上下文指针，写入 t0;

以 t0 作为基址，以用户上下文寄存器或特殊字段的顺序作为偏移值，恢复用户上下文到寄存器中。用户上下文的寄存器顺序见表 4-1 seL4-LoongArch 用户上下文信息描述;

ertn 返回。

4.2 Cmake 设计

seL4-test 采用 cmake 工具为项目生成 build.ninja 文件，执行 ninja 编译出镜像文件。整个 seL4-test 内的架构如图 4-5 所示。

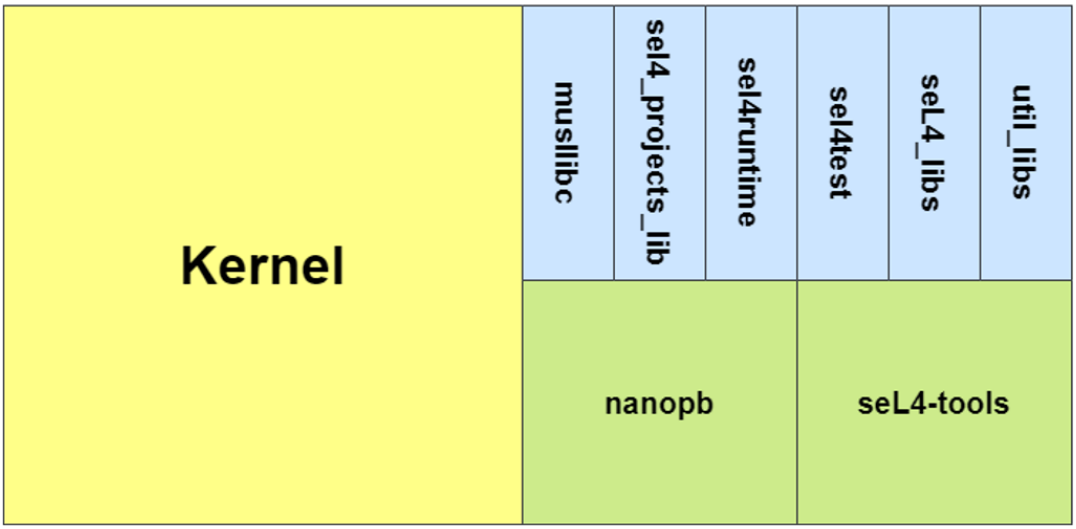

图 4-5 seL4-test 项目结构

seL4-test 项目结构如上图，其中黄色部分代表 seL4 微内核，蓝色部分代表构建 seL4-test 需要用到的 6 个组件，绿色部分是构建和运行过程中的工具和 nanopb[4]。各部分解释如下：

表 4-4 seL4-test 结构描述

组件	描述
Kernel	seL4 微内核

[4] 实际上绿色部分还包含 opensbi，但是因为移植到 LoongArch 没有用到 opensbi，所以这里并没有提及。

sel4test	seL4 用户空间程序及构建入口
seL4_tools	构建 seL4-test 的工具，如 cmake-tool，elfloader-tool 等
musllibc	提供代码静态链接和动态链接的轻量级 C 语言库
sel4runtime	运行 C 语言兼容程序的最小 runtime 系统
seL4_libs	在 seL4 微内核上编写用户程序的程序库
util_libs	seL4 微内核使用的程序库，包括 pci 驱动库、驱动程序库、设备树库等
sel4_projects_lib	seL4 的库集合，与 seL4_libs 兼容。
nanopb	seL4 采用 nanopb 来实现协议缓冲区

seL4-test 项目由 cmake 工具构建，其构建入口在 sel4test 组件内的 CMakeLists。cmake 构建过程分为预设置、kernel 设置和组件设置三个部分。

4.2.1 预设置

预设置主要是为 kernel 准备全局变量和文件路径。cmake 构建过程中涉及许多变量，这些变量最后将通过 cmake 语法及脚本为具体构建的项目生成配置头文件[5]。seL4 现有的 cmake 结构扩展性较强，容易添加 LoongArch 架构的设置，实现细节不再详述。

预设置主要设置的一些 cmake 变量见表 4-5。

表 4-5 预设置的主要变量

变量名	值
project_dir	sel4-test 的路径
project_modules	projects 下 6 个组件路径
CMAKE_MODULE_PATH	kernel,sel4-tools/cmake-tool/helpers、sel4-tools/elfloader-tool 及 6 个组件的路径
NANOPB_SRC_ROOT_FOLDER	nanopb 路径

[5] 在 cmake 定义在 kernel/tools/helpers.cmake 里以 config 开头的函数或宏实现这样的功能，如 config_string、config_option。

SEL4_CONFIG_DEFAULT_ADVANCED	ON
KernelSel4Arch	loongarch64
KERNEL_PATH	kernel 路径
KERNEL_HELPERS_PATH	kernel/tools/helpers.cmake 路径
KERNEL_CONFIG_PATH	kernel/configs/seL4Config.cmake 路径
CROSS_COMPILER_PREFIX	loongarch64-unknown-linux-gnu-
sel4_arch	loongarch64
KernelSel4Arch	loongarch64
KernelPlatformSupportsMCS	OFF
Sel4testHaveTimer	OFF
CMAKE_BUILD_TYPE	DEBUG
BAMBOO	OFF
DOMAINS	OFF
KernelNumDomains	1
SMP	OFF
LibSel4TestPrintXML	OFF
KernelMaxNumNodes	1
MCS	OFF
KernelIsMCS	OFF
KernelRootCNodeSizeBits	13
KernelDomainSchedule	sel4test/domain_schedule.c 路径
KernelArch	loongarch
Kernel64	ON
Kernel32	OFF
KernelPlatform	3A5000
KernelPlatform3A5000	ON
KernelVerificationBuild	OFF
KernelLoongarchPlatform	3A5000
CALLED_declare_default_headers	1(ON)
TIMER_FREQUENCY	100000000

HW_MAX_NUM_INT	8
KernelWordSize	64
INTERRUPT_CONTROLLER	loongarch_extio_dummy.h 路径

注：项目移植目前没有考虑 seL4 的 MCS 和 SMP 多核配置。

除了配置通用架构变量，添加龙芯交叉编译器与架构变量对应关系之外，本文在 kernel 的 plat 文件夹[6]添加了 3A5000 文件夹，声明了 3A5000 对应 dts 文件的路径以及中断控制器[7]。该文件夹还涉及其他架构同系列机器不同的设备树设置需要的文件，因目前只针对 3A5000 平台的移植，所以没有 dts 覆盖文件。dts 覆盖文件通过 cmake 的 file 命令读写完成，最后通过 dtc 程序编译为可用的 dtb，这部分功能已经在 cmake 框架中实现，具体方法可参考 arm 架构的某些型号机器。

4.2.2 Kernel 设置

本节是 seL4 微内核相关的主要配置。下表 4-6 中的前 6 个变量代表的路径指向 6 个脚本(库)，seL4-test 构建编译过程中调用这些脚本来生成项目所需的头文件，以_gen 标识。为适配 LoongArch 架构，团队对这些脚本进行修改，此处不展开叙述。

表 4-6 kernel 设置的主要变量

变量名	值
CPP_GEN_PATH	cpp_gen.sh 脚本路径
CIRCULAR_INCLUDES	circular_includes.py 脚本路径
BF_GEN_PATH	bitfield_gen.py 脚本路径
HARDWARE_GEN_PATH	hardware_gen.py 脚本路径
INVOCATION_ID_GEN_PATH	invocation_header_gen.py 脚本路径
SYSCALL_ID_GEN_PATH	syscall_header_gen.py 脚本路径
XMLLINT_PATH	xmllint.sh 脚本路径

[6] 里面存储平台相关 cmake 的配置文件。
[7] 目前中断尚未完全实现，这里用 dummy 填充。

MAX_NUM_IRQ	10
BITS	4
CONFIGURE_IRQ_SLOT_BITS	4
CONFIGURE_TIMER_PRECISION	0
KernelHaveFPU	OFF
KernelSetTLSBaseSelf	OFF
KernelPTLevels	3
KernelLoongarchExtF	OFF
KernelLoongarchExtD	OFF
KernelClz64	64
KernelCtz64	64
KernelPaddrUserTop	1 << 47
KernelHardwareDebugAPIUnsupported	ON
KernelDTBSize	最终生成 DTB 的字节数
KernelRootCNodeSizeBits	13
KernelTimerTickMS	2
KernelTimeSlice	5
KernelRetypeFanOutLimit	256
KernelMaxNumWorkUnitsPerPreemption	100
KernelResetChunkBits	8
KernelMaxNumBootinfoUntypedCaps	230
KernelFastpath	ON
KernelNumDomains	1
KernelNumPriorities	256
KernelMaxNumNodes	1
KernelEnableSMPSupport	OFF
KernelStackBits	12
KernelVerificationBuild[8]	OFF
KernelDebugBuild	ON(在内核中启用调试工具)

[8] 启用后，此配置选项可防止使用任何其他会危及内核验证的选项。但是启用此选项并不意味着内核已被验证。

HardwareDebugAPI[9]	OFF
KernelPrinting	ON(允许内核在启动和执行期间将消息输出到串行控制台)
KernelInvocationReportErrorIPC	OFF(启用后允许内核将用户错误写入 IPC 缓冲区)
KernelBenchmarks	none
KernelEnableBenchmarks	OFF
KernelLogBuffer	OFF
KernelMaxNumTracePoints	0
KernelIRQReporting	ON
KernelColourPrinting	ON
KernelUserStackTraceLength	16
KernelOptimisation	-O2
KernelFWholeProgram[10]	OFF
KernelDangerousCodeInjection[11]	OFF
KernelDebugDisablePrefetchers[12]	OFF
KernelSetTLSBaseSelf[13]	OFF
KernelClzNoBuiltin	OFF
KernelCtzNoBuiltin	OFF
CMAKE_C_FLAGS[14]	-D__KERNEL_64__ -march=loongarch64 -mabi=lp64d -mtune=loongarch64 -Wno-error=array-bounds
CMAKE_CXX_FLAGS	同上

[9] 启用后会构建支持用户空间调试 API 的内核，该 API 可以允许用户空间进程设置断点，观察点和单步线程执行。

[10] 此选项打开后，链接内核时启用-fwhole-program。

[11] 启用后会添加一个系统调用，允许用户指定要在内核模式下运行的代码。

[12]如果开启可能需要改写与平台相关的禁止预取的逻辑。

[13] 启用后会构建没有 capability 的内核。

[14] CMAKE_C_FLAGS、CMAKE_CXX_FLAGS 和 CMAKE_ASM_FLAGS 是相同的值，但是真正编译时不仅有这里设置的编译参数，seL4 还通过 add_compile_options 语句添加了其他编译参数，详见 kernel 的 CMakeLists。

CMAKE_ASM_FLAGS	同上
CMAKE_EXE_LINKER_FLAGS	-D__KERNEL_64__ -march=loongarch64 -mabi=lp64d -mtune=loongarch64 -Wno-error=array-bounds -static -Wl,--build-id=none -Wl,-n -O2 -nostdlib -fno-pic -fno-pie -DDEBUG -g -ggdb
CMAKE_BUILD_TYPE	RelWithDebInfo
Linker_source	lds 文件路径

出于简洁性，项目未开启 FPU、单双精度浮点扩展标识。根据页表，设置内核页表级数 KernelPTLevels=3。设置虚拟地址空间为 48 位。其他设置参考 riscv 的 spike 平台。编译和链接参数在 CMAKE_C_FLAGS 等 4 个 FLAGS 中设置。链接用到的 lds 文件为 common_loongarch.lds。lds 具体设计请查看项目文件。

4.2.3 组件设置

组件设置分为 elfloader 设置和 sel4test-driver 设置。sel4test-driver 会设置除 elfloader 外构建 seL4-test 使用到的其他组件。

原 seL4-test 的设计中除 x86 架构外，其他架构均会使用 elfloader，我们选择在 LoongArch 架构上也采用 elfloader。在 elfloader 设置中，除将某些变量生成到配置头文件外，还设置了 elfloader 编译的选项。更多技术细节请查阅 readme 中移植笔记仓库。

在 sel4test-driver 设置中，主要内容是编译生成 rootserver 根服务器，并将其与 kernel.elf 和任何所需的加载程序捆绑到一起。具体的修改这里不再叙述，请查看项目文件。

4.3 自动化测试程序（github workflow）移植

在 seL4 的官方微内核 git 仓库里，有生成手册、语法检查、编译内核、测试程序等 14 个 github workflow（github 提供的自动化工作测试程序，用户编写的脚本运行在 github 提供的服务器上），当编程者的行为触发测试程序的特定条件时，github 服务器会执行相应动作。

截至决赛结束，除了与架构无关的自动化程序（生成手册等架构无关的自动化程序可以正常运行），本项目成功移植了 compile workflow、C parser workflow 和 CI workflow 自动化测试程序，以及运行自动化程序依赖的 docker 镜像。

本团队在 seL4 官方镜像的基础上，加入了龙芯的交叉编译工具、测试脚本等内容，构建出用于执行自动化程序的龙芯版本 docker 镜像，各镜像的功能如下表 4-7 所示。

表 4-7 自动化程序使用的龙芯版 docker 镜像

镜像名	镜像功能
la-seL4:latest	该镜像包含单独编译内核的所有依赖（包括龙芯交叉编译工具），支持编译龙芯版 seL4 内核。
la-l4v:latest	该镜像包含构建 l4v 的所有工具和依赖（包括龙芯版本脚本），也是构建 la-cparser-builder 镜像、形式化验证等工作的基础镜像。
la-cparser-builder:latest	该镜像包含 cparser 源码编译分析工具（包括龙芯版本），也是 la-cparser-run 等镜像的基础镜像。
la-cparser-run:latest	该镜像包含上述基础镜像和其他 seL4 官方镜像、脚本，能对内核源码执行更严格的语法检查，为形式化验证工作做准备。

表 4-7 中的镜像资源已经上传到 docker hub，可以公开使用。镜像具体链接见项目 readme 以及本文第 5 章资源分享章节。

4.3.1 Compile workflow

本地库向远程库推送代码时或合并分支时触发 Compile workflow，该 workflow 检查内核源码能否编译出 Loongarch64、x86_64、arm 和 riscv 版本的 seL4 微内核镜像。

4.3.1.1 项目仓库、docker 构建及配置文件修改

关于 seL4-CAmkES-L4v-dockerfiles 仓库：修改 build.sh 和 sel4.Dockerfile：增加 build_sel4_la()函数，增加 sel4la.Dockerfile 文件。sel4la.Dockerfile 以原 trustworthysystems/seL4 镜像为基础层，加入龙芯交叉编译工具链层，build_sel4_la()最终根据 sel4la.Dockerfile 文件生成 la-sel4:latest 镜像，将 la-sel4:latest push 到 docker hub 上；

关于 seL4-ci-actions 仓库：修改/standalone-kernel/Dockerfile，引用 docker hub 的 la-sel4:latest 镜像；修改/standalone-kernel/Dockerfile /compile_kernel.sh，作为 Dockerfile 入口，增加对 LoongArch64 的支持；

关于 seL4 仓库：修改/.github/workflows/compilation-checks.yml 文件，添加待编译架构 LoongArch64，以及对应编译器 gcc。

4.3.1.2 Compile workflow 运行效果

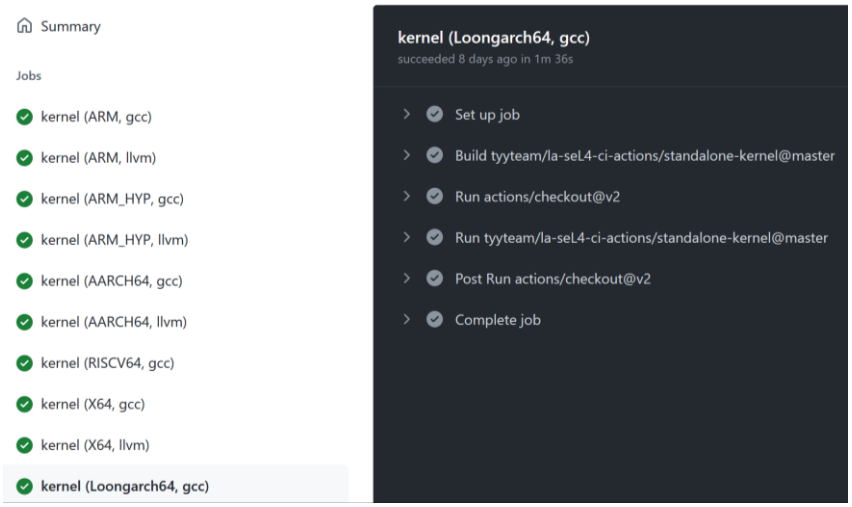

图 4-6 seL4-Loongarch64 compile worflow 运行效果

如图 4-6 所示，当本地库推送代码到远程库时，该 workflow 自动编译 Loongarch64、x86_64、arm 和 riscv 版本的 seL4 微内核。该 workflow 的运行结果表明：Loongarch64 等版本的 seL4 微内核均通过编译，为运行用户测试程序做好准备。

4.3.2 C Parser workflow

本地库向远程库推送代码时触发 C Parser workflow，该 workflow 检查 seL4 的 C 语言子集能否支持现有内核代码，并执行比编译器更严格的语法规范检查，

为形式化验证作准备。

4.3.2.1 项目仓库、docker 构建及配置文件修改

关于 sel4-ci-actions 仓库：修改 cparser-run/下的 Makefile 和 Dockerfile，生成龙芯版 la-cparser-run 镜像；修改 docker/下的 Makefile 和 cparser-builder.dockerfile，生成 la-cparser-builder 镜像，该镜像是 la-cparser-run 镜像的基础镜像。

关于 la-seL4-CAmkES-L4v-dockerfiles 仓库：增加 dockerfiles/ l4vla.Dockerfile 文件；增加 scripts/下的 l4vla1.sh，l4vla2.sh，l4vla3.sh，拆成 3 个文件是需要利用 docker build 过程中的缓存加速构建过程，降低构建失败的成本；增加 build.sh 下的 build_l4v_la()函数，编译出龙芯版 la-l4v 镜像，该镜像是 la-cparser-builder 的基础镜像。

关于 l4v 仓库：修改 tools/c-parser/standalone-parser/Makefile；增加 tools/c-parser/umm_heap/Loongarch64 文件夹；构建 la-cparser-builder 镜像过程中拉取该仓库并执行 make standalone-cparser 命令，生成 Loongarch64 等架构的 cparser 解析器。

4.3.2.2 C Parser workflow 运行效果

该 workflow 的运行效果如下图 4-7 所示。该 workflow 使用 cmake 工具构建出 kernel_all_pp.c，该文件是编译器用于生成 kernel.elf 内核镜像的综合文件。然后，该 workflow 用 Loongarch64-cparser 分析 seL4 的 C 语言子集能否支持 kernel_all_pp.c 的代码，并执行严格的语法规范检查。

本项目**尚有少量代码未通过严格规范检查，但不影响代码的编译运行，这部分工作需要熟悉 seL4 相关的** Isabelle/HOL **语言。团队将继续推进 seL4-Loongarch64 代码规范，为形式化验证做好准备。**

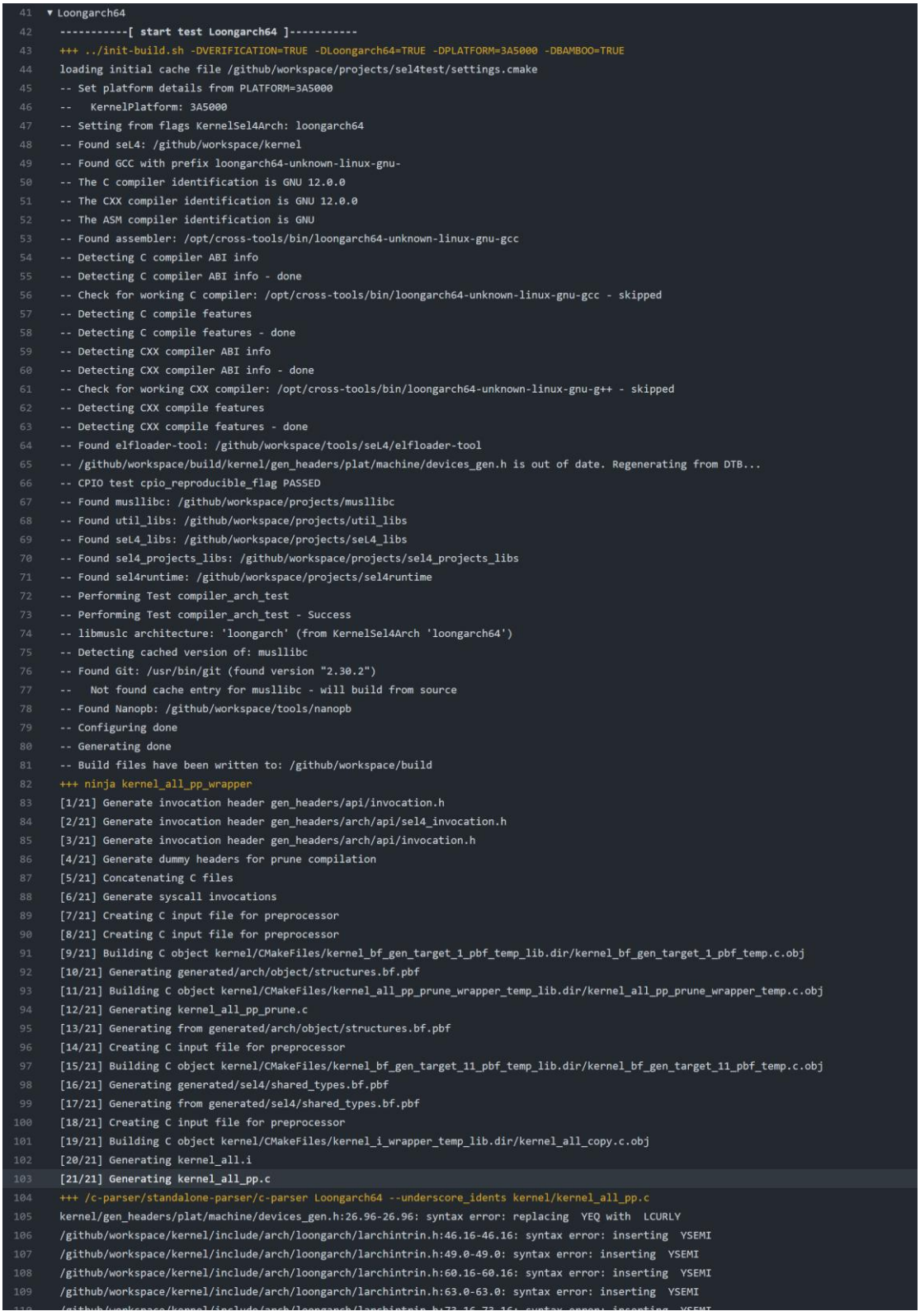

图 4-7 seL4-Loongarch64 C Parser worflow 运行效果

4.3.3 CI workflow

本地库向远程库推送代码时或合并分支时触发 CI workflow，该 workflow 包含 License Check 和 Links 两个 action。License Check 采用 fsfe/reuse-tool 检查 seL4 源码文件的开源许可证和版权是否有效，Links 检查超链接是否有效。

4.3.3.1 版权和许可证信息

本项目的团队版权信息如下：

Copyright 2022, tyyteam(Qingtao Liu, Yang Lei, Yang Chen)

qtliu@mail.ustc.edu.cn, le24@mail.ustc.edu.cn, chenyangcs@mail.ustc.edu.cn

seL4-Loongarch64 文件或比赛文档使用许可证如下：

SPDX-License-Identifier: GPL-2.0-only

4.3.3.2 CI workflow 运行效果

该 workflow 的运行效果如图 4-8 所示。

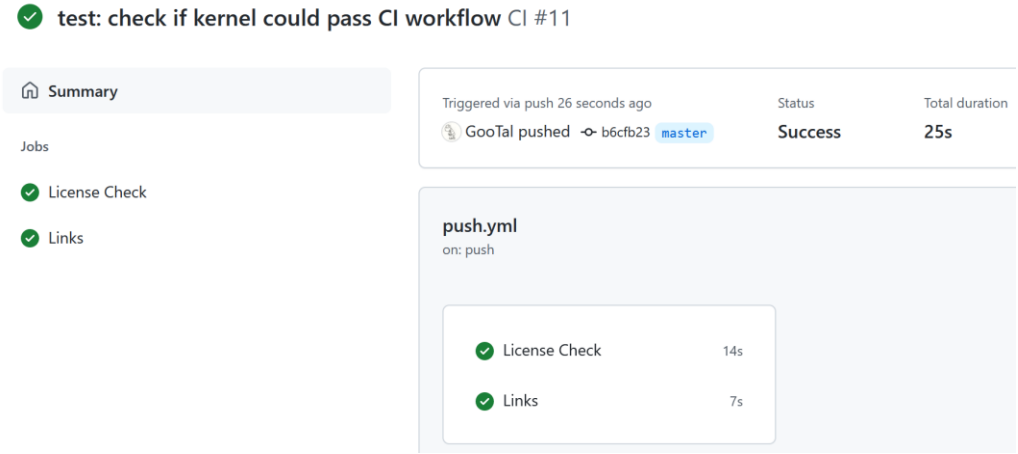

图 4-8 seL4-Loongarch64 CI workflow 运行效果

图 4-8 表明 seL4-Loongarch64 项目的版权信息和开源许可证有效，文件的超链接和 http 连接有效。图 4-9 和图 4-10 显示了 License Check 和 Links 的详细信息。

图 4-9 CI workflow 中 License Check 详细信息

图 4-9输出语句说明该workflow检查微内核使用的开源许可证和版权信息，最后的语句说明 seL4-Loongarch64 的工作能够兼容 REUSE3.0 规范。

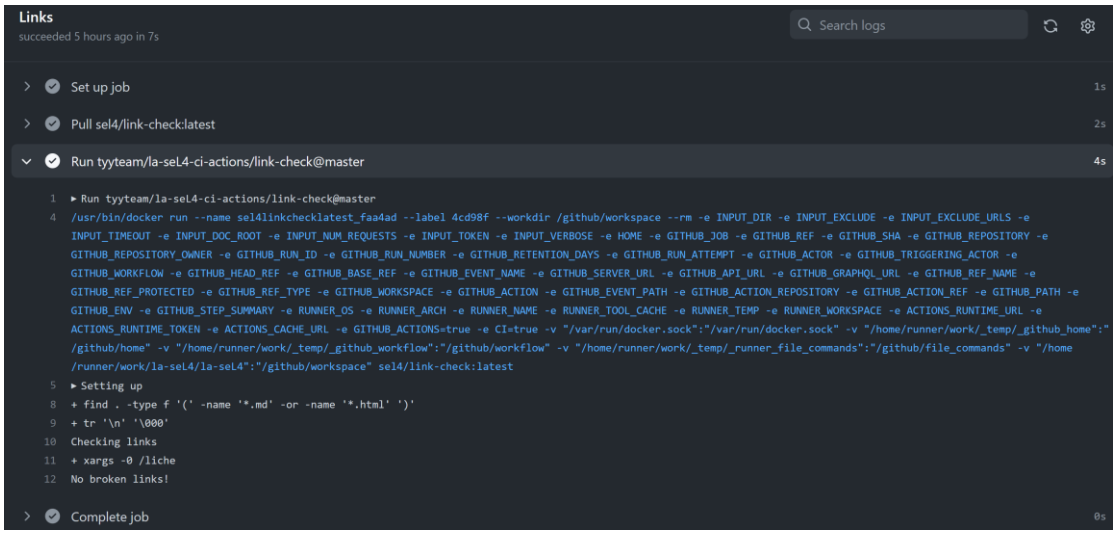

图 4-10 CI workflow 中 Links 详细信息

图 4-10 输出语句说明该 workflow 检查微内核的.md 或.html 文件中的超链接，经检查，所有超链接均有效。

4.4 其他移植工作

4.4.1 设备树移植

设备树（device tree， dts）最初用于 Linux 内核中，seL4 也使用这类文件为内核提供硬件设备信息，如内存起始地址，uart 串口设备信息等。seL4 内核中架

构相关的 dts 文件存放在 **kernel/tools/dts** 文件夹下。针对龙芯 3A5000 平台，本团队添加了 3A5000.dts 文件。

从 qemu 导出 dtb，用 dtc 命令反编译出 dts，针对 seL4 的主要修改如下：

1) 物理内存，起始地址设为 0x90000000，结束地址设为 0xefffffff。
2) uart 串口设备，地址设为 0x1fe001e0，compatible 字符串设置为 "3A5000,loongson3A5000-uart"。

seL4 的 kernel/config.cmake 文件调用 kernel/tools/hardware/outputs 下的 python 文件解析 dts，然后在 build 过程中生成内核代码。

4.4.2 uart 串口设备驱动程序

Loongson 3 号 I/O 控制器支持 UART 寄存器，Loongson3A5000 芯片内部集成两个 UART 接口，其中 UART0 寄存器物理地址为 0x1FE001E0。

内核由 elfloader 启动，由于内核与 elfloader 在不同的项目中，无法使用同一套驱动程序，所以本团队为 elfloader 和内核均添加了驱动支持。

- kernel 项目的 uart 支持和编译过程分析：

1) 本文在 4.4.1 描述了向设备树添加 uart 节点的工作，uart 节点的兼容性属性定义为"3A5000,loongson3A5000-uart"。
2) 在 kernel/src/drivers/serial 文件夹下添加 loongson3A5000-uart.c，该文件定义了 Loongson3A5000 uart0 串口的字符读写函数。
3) 除上述驱动程序以外，还在 kernel/src/drivers/serial 下的 config.cmake 文件中注册该 uart 的驱动程序文件 loongson3A5000-uart.c，并设置兼容性字符串为"3A5000,loongson3A5000-uart"。在 kernel/tools/hardware.yml 文件中也将"3A5000,loongson3A5000-uart"加入 compatible 集合。

seL4 内核编译和运行时，cmake 文件根据当前架构解析龙芯的 dts 文件，读入 uart 节点的兼容属性字符串，然后在 kernel/tools/hardware.yml 文件中检查是否有匹配字符串，并在 kernel/src/drivers/config.cmake 文件中根据该字符串编译注册的 uart 驱动程序文件 loongson3A5000-uart.c。seL4 内核的 printf()最终调用架构相关的 uart 驱动程序接口，实现字符打印。

注：在编译过程中，seL4 内核会根据内核中的 dts 文件生成头文件 build_3A5000/kernel/gen_headers/plat/machine/devices_gen.h, elfloader 将读取其中关于 uart 的物理地址。elfloader 中没有使用 dts 或其他文件来说明 uart 物理地址。

- elfloader 项目的 uart 支持和调用过程分析：

1) 在 tools/seL4/elfloader-tool/src/drivers/uart 文件夹下添加文件 loongson3A5000-uart.c。该文件包含打印字符函数，设备初始化函数，以及 dtb_match_table，elfloader_uart_ops 和 elfloader_driver 结构体。在 dtb_match_table 结构体中设置兼容字符串；在 elfloader_uart_ops 中设置了打印字符函数指针;在 elfloader 中设置了设备类型,初始化函数指针，elfloader_uart_ops 指针。

2) 在 tools/seL4/elfloader-tool/src/arch-loongarch/boot.c 文件的 main()函数中调用 initialise_devices 函数初始化 uart 设备。

elfloader 运行时，进入 main()函数，首先调用 initialise_devices 初始化 LoongArch 架构的设备。initialise_devices() 在 tools/seL4/elfloader-tool/src/drivers/driver.c 文件中，这是 seL4 初始化驱动设备的上层函数，该函数根据匹配的兼容字符串，调用相应设备的 init 函数指针，初始化驱动设备，即初始化 uart 设备。elfloader 的 printf()根据 uart 设置的打印字符函数指针，调用 uart 驱动函数，实现字符打印。

除 uart 以外，其他部分也可以见到回调函数的设计：底层注册 uart 的驱动函数，上层用回调函数来调用底层驱动实现字符打印。这种分层的软件体系结构模式，降低了高层代码对底层驱动程序的依赖程度，有利于系统的完善和维护。

4.5 运行展示与分析

Qemu 模拟的硬件平台为龙芯 3A5000+7A1000 桥片

1. Qemu 系统模式运行 seL4-elfloader

```
SetUefiImageMemoryAttributes - 0x00000000FE9C0000 - 0x0000000000040000 (0x000000
0000000000)
SetUefiImageMemoryAttributes - 0x00000000FE920000 - 0x0000000000040000 (0x000000
0000000000)
kernel argc: 2, argv: 0xFE6F0000, bpi: 0xFFFDA418.
entry kernel ...
ELF-loader started on (HART 0) (NODES 1)
  paddr=[2000000..2461097]
Looking for DTB in CPIO archive...found at 20ad400.
Loaded DTB from 20ad400.
   paddr=[92030000..92030fff]
ELF-loading image 'kernel' to 90000000
  paddr=[90000000..9202ffff]
  vaddr=[ffff810010000000..ffff81001202ffff]
  virt_entry=ffff810010000000
ELF-loading image 'sel4test-driver' to 92031000
  paddr=[92031000..92476fff]
  vaddr=[120000000..120445fff]
  virt_entry=12000eff0
l1pt:2458000
l2pt:245c000
l2pte0:900011d3
Enabling MMU and paging
Jumping to kernel-image entry point...
```

图 4-11 qemu 模拟-seL4 elfloader

Qemu 启动以后，首先运行龙芯的 UEFI BIOS，UEFI BIOS 装载内核时，跳转到内核 elf 文件的入口地址抹去高 32 位的地址。这里我们 elfloader 的地址是 0x0200_0000，装载地址也是 0x0200_0000。如上文所述我们跑的是 sel4test-driver-image-loongarch-3A5000，其实质上是 elfloader，而把 kernel image、dtb 和 user image 用 cpio 打包作为 elfloader 的数据部分。

elfloader 开始运行时，使用物理地址访问，然后配置了一个 0x9000 开头的直接映射窗口，自跳转到直接映射窗口中的地址继续运行。

elfloader 做了如下工作：

1) 配置直接映射窗口，配置 CRMD 和 PRMD；
2) 从 CPIO 中提取 kernel image、dtb 和 user image，解析各自的 elf 头文件信息，得到 kernel image 和 user image 的入口地址、大小等相关信息，并将其装入对应的装载地址，其中 dtb 跟在 kernel image 之后；
3) 用 32MB 大页对 kernel image 区域做了个临时页表映射；
4) 配置和页表映射相关的 CSR：PWCL 和 PWCH，具体为三级页表，页表索引位 11 位，页内偏移量 14 位；将顶级页表物理地址设置到 CSR.PGDH；
5) 初始化 TLB，配置相关 CSR：TLBIDX、STLBPS、TLBREHI，配置 tlb 重填例外函数入口地址到 TLBRENTRY；

6) 跳转到 kernel image 的虚拟入口地址，开始运行 kernel。

2. Qemu 系统模式运行 seL4-kernel

```
kernel_phys_region_start: 90000000
kernel_phys_region_end: 92030000
kernel_phys_virt_offset: 7f0080000000
kernel_virt_entry: ffff810010000000
ui_phys_region_start: 92031000
ui_phys_region_end: 92477000
ui_phys_virt_offset: ffffffff72031000
ui_virt_entry: 12000eff0
dtb physical address: 92030000
dtb size: 1320
Init local IRQ
no extio present, skip hart specific initialisation
Bootstrapping kernel
Initializing extend io interrupt...
no extio interrupt supported yet. Will be supported later
available phys memory regions: 1
  [90000000..17fffffff]
reserved virt address space regions: 3
  [ffff800090000000..ffff800092030000]
  [ffff800092030000..ffff800092030528]
  [ffff800092031000..ffff800092477000]
Booting all finished, dropped to user space
```

图 4-12 qemu 模拟-seL4 kernel

elfloader 通过 a0～a5 传递给 kernel 初始化所需的参数（user image 物理起始地址、user image 物理终止地址、物理/虚拟地址偏移量、user image 虚拟入口地址、DTB 的物理地址和 DTB 的大小，这些参数都在 elfloader 解析 kernel image 和 user image 的 elf 头文件时获取）

kernel 启动后主要做了如下工作：

1) 映射 PSpace、Kernel ELF、Kernel Devices 页表，替换掉 elfloader 使用的临时页表；
2) 重新初始化 TLB；
3) 使能软中断、定时器中断、性能监测计数溢出中断、硬件中断；
4) 配置例外入口，TLB 重填例外入口在 elfloaer 中以已经配置，这里只配置普通例外入口和机器错误例外入口；
5) 根据 kernel image、dtb、user image 所用的内存区域及 dts 中设置的可用物理内存区域计算出剩余可用的空闲物理内存区域，如图 4-12 所示启动过程中有打印输出；
6) 计算 initial thread 大小，并为其分配空间，创建 initial thread，分配相关的 capability；
7) 创建 idle thread；

8) 根据剩余空闲内存区域创建 untyped objects；

9) 开始线程调度，切换用户线程。

3. Qemu 系统模拟内核加载完后进入用户态

```
Booting all finished, dropped to user space
Node 0 of 1
IOPT levels:      0
IPC buffer:       0x12045c000
Empty slots:      [386 --> 8192)
sharedFrames:     [0 --> 0)
userImageFrames: [17 --> 296)
userImagePaging: [14 --> 16)
untypeds:         [296 --> 386)
Initial thread domain: 0
Initial thread cnode size: 13
List of untypeds
------------------
Paddr     | Size   | Device
0 | 31 | 1
0x80000000 | 28 | 1
0x180000000 | 31 | 1
0x200000000 | 33 | 1
0x400000000 | 34 | 1
0x800000000 | 35 | 1
0x1000000000 | 36 | 1
0x2000000000 | 37 | 1
```

图 4-13 qemu 模拟-进入用户态

如图 4-13 所示，终端打印了一些内核加载过程中给用户主线程分配的资源信息（IPC buffer、frames、cnode 等）。

```
Starting test suite sel4test
Starting test 0: Test that there are tests
Starting test 1: SYSCALL0000
Starting test 2: SYSCALL0001
<<seL4(CPU 0) [decodeCNodeInvocation/54 T0xffff80017ffdc200 "sel4test-driver" @
120005178]: CNodeCap: Illegal Operation attempted.>>
<<seL4(CPU 0) [decodeCNodeInvocation/54 T0xffff80017ffdc200 "sel4test-driver" @
120005178]: CNodeCap: Illegal Operation attempted.>>
<<seL4(CPU 0) [decodeCNodeInvocation/54 T0xffff80017ffdc200 "sel4test-driver" @
120005178]: CNodeCap: Illegal Operation attempted.>>
<<seL4(CPU 0) [decodeCNodeInvocation/54 T0xffff80017ffdc200 "sel4test-driver" @
120005178]: CNodeCap: Illegal Operation attempted.>>
<<seL4(CPU 0) [decodeCNodeInvocation/54 T0xffff80017ffdc200 "sel4test-driver" @
120005178]: CNodeCap: Illegal Operation attempted.>>
<<seL4(CPU 0) [decodeCNodeInvocation/54 T0xffff80017ffdc200 "sel4test-driver" @
120005178]: CNodeCap: Illegal Operation attempted.>>
<<seL4(CPU 0) [decodeCNodeInvocation/54 T0xffff80017ffdc200 "sel4test-driver" @
120005178]: CNodeCap: Illegal Operation attempted.>>
<<seL4(CPU 0) [decodeCNodeInvocation/54 T0xffff80017ffdc200 "sel4test-driver" @
120005178]: CNodeCap: Illegal Operation attempted.>>
<<seL4(CPU 0) [decodeCNodeInvocation/54 T0xffff80017ffdc200 "sel4test-driver" @
```

图 4-14 qemu-用户态运行 seL4test 测试程序

如图 4-14 所示，qemu 在用户态运行 seL4test 程序，seL4test 包括 seL4test-driver 和 seL4test-test 两个模块，seL4test-driver 负责加载 seL4test-test，然后进行了系统调用、IPC、进程管理、调度等内核功能测试。

第5章 开源资源分享

本章介绍项目的学习资源和软件资源，以促进操作系统教学，推进龙芯生态建设，扩大 seL4 开源社区影响力。**具体链接在项目 readme 中**。

5.1 项目资源

团队在 2022 年 8 月 7 日技术分享会上的报告。

团队的 github 组织（tyyteam org）现有 15 个仓库，包含移植笔记、代码注释、CI 仓库、docker 库、微内核库等资源。

Docker hub 镜像，包含:la-seL4、la-l4v、la-cparser-builder、la-cparser-run。

5.2 龙芯资源

张老师提供了 qemu-loongarch-runenv 仓库,包含指令集手册、3A5000 手册、7A1000 手册,以及 qemu-system-loongarch64、mini_kernel demo、linux-loongarch64、交叉编译工具链等资源。

5.3 seL4 资源

seL4 的 github 组织（seL4 org）有 55 个仓库。seL4 官网也有文档资料，包含参考手册、CamkES 构建工具、动态库教程、API 等。

第6章 项目回顾与展望

回顾初赛到决赛的 5 个月，团队熟悉了 seL4 设计思想，API 教程，代码框架，LoongArch、riscv 架构文档，cmake、ninja 等项目工具。

团队开发思路：首先完成 seL4 的入门教程，结合论文学习内核架构无关代码的设计思想；然后阅读 seL4 源码，并结合 riscv 教程、xv6-mips 源码、Linux-LoongArch 源码编写底层代码；阅读并修改 cmake 文件，成功编译内核；最后利用 qemu 模拟器调试内核，不断优化代码结构，完善移植工作。

初赛期间，团队移植了 seL4 的 elfloader 到 LoongArch 平台；结合 LoongArch 指令、虚拟内存管理、中断例外处理方式和 seL4 内核特点设计并移植实现了 seL4 内核的 LoongArch 版本，调试到激活线程的位置；在原 Cmake 文件添加了龙芯架构支持，成功编译出 LoongArch 版本的 seL4 项目相关可执行 elf 文件。

决赛期间，团队完成了内核移植，在初赛基础上完善了内存管理、中断与例外等模块，使得内核能够引导用户空间程序；进入 sel4test 测试程序，通过了 15 个测试样例；移植自动化测试程序（github workflow）：通过 Compile workflow、C Parser workflow、CI workflow 和 RefMan workflow。

项目展望：在内核等仓库的移植方面，拟持续优化虚拟内存管理、例外与中断等模块，移植 CI 库、docker 库、用户级库函数、CamkES 构建工具等仓库；在分析与优化方面，拟深入研究软硬件协同的性能优化，虚拟化支持，形式化验证和安全系统等方向；在资源搜集与分享方面，拟在第 5 章基础上继续搜集内核设计思想、移植和优化等资料，分享团队的移植经验和方法，完善项目文档和代码注释，为国产 LoongArch 指令集和 seL4 社区贡献力量。

致谢

2022 年 3 月，团队成员的操作系统及内核开发经验可谓“一穷二白”，但经过队员不懈努力，成功挺进决赛。截至 8 月 15 日，团队已经熟悉 seL4 微内核框架，cmake 项目构建工具，LoongArch、RISC-V 架构手册，qemu 模拟器，CI 库和 docker 库，用户测试程序等工作。

团队的进步离不开队员（刘庆涛，雷洋和陈洋）之间的支持和鼓励。团队每周开 2 次组会，讨论技术，积累文档，探索方向，队员分工明确，配合默契，可谓精诚合作，金石为开。

团队的进步更离不开指导老师（张福新老师和高燕萍老师）的辛勤付出。张福新老师不仅为团队指导了工作方向，还给出 LoongArch 代码示例和 qemu 调试细节。高燕萍老师为项目组织和远程机器提供了有力支持。

团队还要感谢徐淮，胡起，袁宇翀，谢本壹，梁思远的帮助和建议，感谢 seL4 技术团队（Kent McLeod，Axel Heider，Jashank Jeremy，Gernot Heiser，Gerwin Klein 等人）在 github issue 上的指导和支持。

谨以此篇，向本项目的指导老师，团队成员和其他参与者表达感谢。

参考文献

[1] Hansen P B. The nucleus of a multiprogramming system[J]. Communications of the ACM, 1970, 13(4): 238-241.

[2] Condict M, Bolinger D, Mitchell D, et al. Microkernel modularity with integrated kernel performance[R]. Technical report, OSF Research Institute, Cambridge, MA, 1994.

[3] Chen J B, Bershad B N. The impact of operating system structure on memory system performance[C]//Proceedings of the fourteenth ACM symposium on Operating systems principles. 1993: 120-133.

[4] Liedtke J. Improving IPC by kernel design[C]//Proceedings of the fourteenth ACM symposium on Operating systems principles. 1993: 175-188.

[5] Härtig H, Hohmuth M, Liedtke J, et al. The performance of μ-kernel-based systems[J]. ACM SIGOPS Operating Systems Review, 1997, 31(5): 66-77.

[6] Klein G, Andronick J, Elphinstone K, et al. Comprehensive formal verification of an OS microkernel[J]. ACM Transactions on Computer Systems (TOCS), 2014, 32(1): 1-70.

[7] Shapiro J S, Smith J M, Farber D J. EROS: a fast capability system[C] //Proceedings of the seventeenth ACM symposium on Operating systems principles. 1999: 170-185.

[8] Hardy N. KeyKOS architecture[J]. ACM SIGOPS Operating Systems Review, 1985, 19(4): 8-25.

[9] Dennis J B, Van Horn E C. Programming semantics for multiprogrammed computations[J].Communications of the ACM, 1966, 9(3):143-155.

[10] seL4.About seL4[EB/OL].[2022-5-18].https://sel4.systems/About/.

[11] Heiser G, Elphinstone K. L4 microkernels: The lessons from 20 years of research and deployment[J]. ACM Transactions on Computer Systems (TOCS), 2016, 34(1): 1-29.

[12] 纤夫张.十年 seL4，依然是最好，依然在进步 [EB/OL].(2020-5-31)[2022-5-18]. https://zhuanlan.zhihu.com/p/144802629

[13] Gernot Heiser.seL4 Design Principles [EB/OL]. (2020-3-11) [2022-5-18].https://microkerneldude.org/2020/03/11/sel4-design-principles/.

[14] seL4 Docs [EB/OL]. [2022-5-23]. https://docs.sel4.systems.

[15] Heiser G. The seL4 Microkernel–An Introduction[J]. 2020.

[16] Biggs S, Lee D, Heiser G. The jury is in: Monolithic os design is flawed:

Microkernel-based designs improve security[C]//Proceedings of the 9th Asia-Pacific Workshop on Systems. 2018: 1-7.

[17] Trustworthy Systems Team. seL4 Reference Manual: version 12.1.0[M]. 2021.