



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

内核实现赛道

2022 年 5 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，项目名称：开源操作系统的 LoongArch 移植-seL4 微内核。本项目由龙芯中科技术股份有限公司和中国科学院计算技术研究所提供技术支持，由中国科学技术大学的涛羊羊团队设计并实现。

seL4 是一个高安全性、高性能的操作系统微内核，作为安全关键系统的可靠基础部件，其独特之处一是完全形式化验证而不影响性能，二是使用了基于 capability 的访问控制。目前，seL4 官方已经实现 x86，Arm 和 RISC-V 版本的 seL4 微内核，为推进国产指令集 LoongArch 的生态建设和 seL4 社区工作，本团队移植 seL4 到 LoongArch 架构。

项目进展如下：

1. 移植 elfloader，参考 seL4 其它架构实现的启动流程，将 elfloader 移植到 LoongArch 平台。
2. 移植 seL4-test，包括：结合 LoongArch 指令、虚拟内存管理、中断例外处理方式和 seL4 内核特点设计并实现了 seL4 微内核的 LoongArch 版本。目前调试到激活线程的位置。
3. 解析原 Cmake 文件并添加 LoongArch 架构支持，成功编译出 LoongArch 版本的 seL4-test 可执行 elf 文件，用龙芯交叉编译工具对其调试。
4. 充分调研，共享大量文档资料和代码注释。

本文共分为四个章节：

第一章，项目背景。本章介绍 seL4-la 项目背景，包括 seL4 的发展历史和简介，以及 seL4 微内核移植到 LoongArch 的意义。

第二章，seL4 简介。本章详尽调研 seL4 微内核设计理念，阐述了 seL4 的设计模式，以及一些现有的基于 seL4 的项目。

第三章，seL4-test 的 LoongArch 移植。本章介绍移植 seL4 微内核到

LoongArch 架构具体工作, 包括内存管理相关设计, 用户上下文和通信消息设计, 中断与例外处理方式, Cmake 的设计以及 dts 和 uart 驱动的编写。

第四章, 回顾与展望。本章是从 3 月至今, 团队对 la-seL4 移植项目的回顾与展望。

目录

| | |
|--|-----|
| 摘要..... | I |
| 目录..... | III |
| 图目录..... | V |
| 表目录..... | VI |
| 第一章 项目背景..... | 1 |
| 1.1 L4 微内核家族..... | 1 |
| 1.2 seL4 微内核的发展..... | 2 |
| 1.3 项目意义..... | 4 |
| 第二章 seL4 简介..... | 5 |
| 2.1 seL4 是什么..... | 5 |
| 2.1.1 seL4 与主流操作系统的区别..... | 5 |
| 2.1.2 seL4 也是管理程序..... | 6 |
| 2.2 seL4 设计模式..... | 7 |
| 2.2.1 基于 capabilities 的访问控制..... | 7 |
| 2.2.2 seL4 内核服务和对象..... | 9 |
| 2.2.2.1 内核对象..... | 10 |
| 2.2.2.2 内核服务——系统调用..... | 11 |
| 2.2.2.3 内核内存分配..... | 13 |
| 2.2.3 Capability 空间..... | 15 |
| 2.2.4 IPC..... | 16 |
| 2.2.4.1 消息寄存器..... | 16 |
| 2.2.4.2 端点 Endpoint 上的 IPC..... | 17 |
| 2.2.5 Notifications..... | 18 |
| 2.2.5.1 Notification 对象..... | 18 |
| 2.2.5.2 发信号 (Signal)、轮询 (Poll) 和等待 (Wait)..... | 18 |
| 2.2.5.3 绑定 notification 信号量..... | 19 |

| | | |
|---------|-----------------------------|----|
| 2.3 | 基于 seL4 的项目 | 19 |
| 第三章 | seL4-test的LoongArch移植 | 20 |
| 3.1 | 内核设计..... | 20 |
| 3.1.1 | 内存管理相关设计..... | 20 |
| 3.1.1.1 | 虚拟地址结构..... | 20 |
| 3.1.1.2 | 虚拟地址空间..... | 21 |
| 3.1.1.3 | 内核装载流程..... | 22 |
| 3.1.1.4 | TLB 的使用 | 24 |
| 3.1.2 | 用户上下文和通信消息设计..... | 24 |
| 3.1.2.1 | 用户上下文设计..... | 24 |
| 3.1.2.2 | 通信消息设计..... | 26 |
| 3.1.3 | 中断与例外处理..... | 27 |
| 3.1.3.1 | LoongArch 中断与例外简介 | 27 |
| 3.1.3.2 | TLB 重填例外 | 28 |
| 3.1.3.3 | 普通例外..... | 29 |
| 3.2 | Cmake 设计 | 30 |
| 3.2.1 | 预设置..... | 32 |
| 3.2.2 | Kernel 设置..... | 34 |
| 3.2.3 | 组件设置..... | 37 |
| 3.3 | 其他移植工作..... | 37 |
| 3.3.1 | 设备树移植..... | 37 |
| 3.3.2 | uart 串口设备驱动程序 | 37 |
| 3.4 | 运行展示..... | 39 |
| 3.4.1 | Qemu 虚拟机运行..... | 39 |
| 第四章 | 回顾与展望..... | 42 |
| | 致谢..... | 43 |
| | 参考文献..... | 44 |

图目录

| | |
|------------------------------------|----|
| 图 1 L4 家族..... | 1 |
| 图 2 微内核和内核[15]..... | 5 |
| 图 3 虚拟化技术将 Linux 提供的服务集成到本机..... | 6 |
| 图 4 capability 是传递权限的密钥..... | 7 |
| 图 5 虚拟地址结构..... | 20 |
| 图 6 虚拟地址空间..... | 21 |
| 图 7 seL4 在 LoongArch 上的内核启动流程..... | 22 |
| 图 8 龙芯 3 号处理器+龙芯 7A 桥片地址空间划分..... | 23 |
| 图 9 seL4-test 项目结构..... | 31 |
| 图 10 qemu 模拟-seL4 elfloader..... | 39 |
| 图 11 qemu 模拟-seL4 kernel..... | 40 |

表目录

| | |
|-----------------------------------|----|
| 表 1 各种对象的权限..... | 9 |
| 表 2 对象大小..... | 15 |
| 表 3 seL4_IPCBuffer 结构体字段..... | 17 |
| 表 4 seL4-LoongArch 用户上下文信息描述..... | 24 |
| 表 5 TLB 重填例外涉及的重要寄存器..... | 28 |
| 表 6 普通例外涉及的重要寄存器..... | 28 |
| 表 7 seL4-test 结构描述..... | 31 |
| 表 8 预设置的主要变量..... | 32 |
| 表 9 kernel 设置的主要变量..... | 34 |

第一章 项目背景

本章主要介绍 seL4 的历史背景和发展情况，阐明将 seL4 移植到 LoongArch 架构的意义。

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是由在微内核基础上构建的用户模式服务器提供的[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，低 IPC 成本是十分必要的。

到 20 世纪 90 年代初，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 $100\mu s$ ，对于构建高性能系统代价过于沉重。而这也导致了微内核设计模式逐渐式微[2]，甚至有人开始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，还有人实现了对 L4 全面的形式验证，包括微内核实现的功能正确性证明和低至可执行二进制文件的高级安全属性的完整证明链[6]。L4 的出现对其他的研究工作也产生了较大的影响，如 EROS [7]。

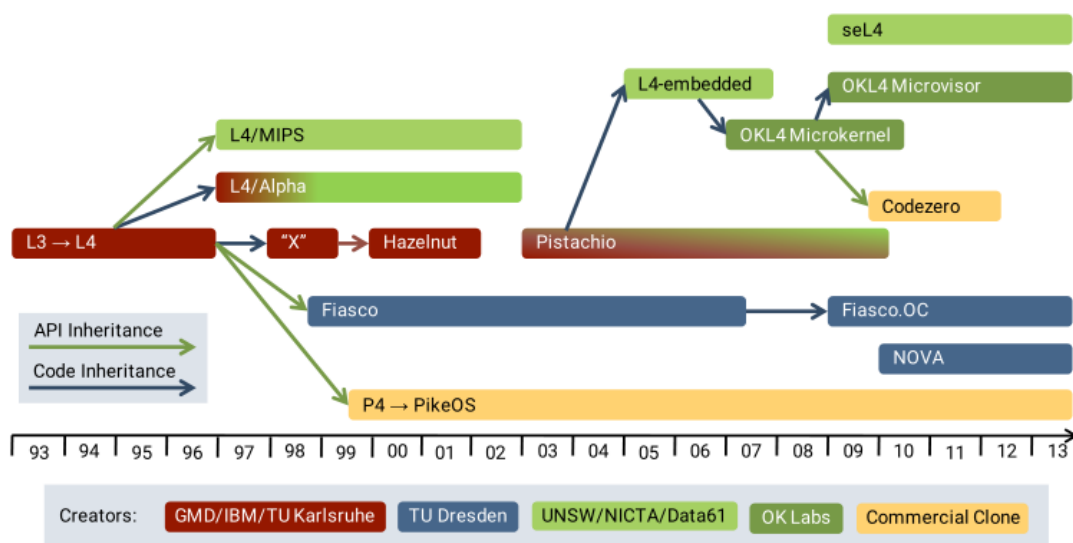


图 1 L4 家族

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的实现（图 1-1[11]），已经拥有了活跃的用户和开发人员社区，并且有了在安全关键商业系统大规模部署的应用。

1.2 seL4 微内核的发展

后来的 EROS 和 KeyKOS[8] 系统的设计模式以及人们对操作系统安全性的日益关注使得 L4 为了实现访问控制采用了 capabilities[9] 的概念。在此基础上，为了实现对支持访问控制的 L4 的形式验证，在 2004 年，Gernot Heiser 等人着手开始实现基于 capabilities 的 L4 微内核——seL4。

2009 年，seL4 完成了除引导代码外的形式化验证；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），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 年证明了在 separation-kernel 配置下的 seL4 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 的 C 代码的正确翻译，从而不必信任 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 是一个高保证、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。他的独特之处在于其全面的形式验证，而不影响性能。seL4 旨在作为可信的基础，以此来构建安全和安保关键型系统[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，以特权模式运行。作为微内核，seL4 被简化为了一个不受策略限制的最小核心，可以为构建服务于不同使用场景的任意系统提供可靠的基础。seL4 在系统中运行的应用程序之间提供了最高的隔离保证，这个实现在数学上已经被证明是正确的，而且 seL4 内核的操作也已经被证明了是在最坏情况下执行时间的安全上限。

seL4 是 L4 家族发展 20 年研究的成果，而且还在进一步发展中。seL4 设计

之初就受到许多原则的驱动，概况为以下几点[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

这同样也是 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 十分重要。

3. 通用性

同样也是 L4 乃至所有微内核的最初设计目标：成为非常大类系统设计的基础。也正是这个原因，极简性并没有极端化。

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 的核心驱动¹。

6. 安全性

这也是在 seL4 之前的 OKL4 微内核同样采用 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该确保危险的操作只会被值得信任的人来使用。
- 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 设计者的工作。

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。

-
- 3) seL4 不会将硬件层进行抽象，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

除安全性和可验证性外，其他原则并不是绝对的。seL4 的各个原则存在冲突的情况下，需要进行取舍来进行均衡，比如，可验证性和高效性之间的均衡。

1.3 项目意义

世界已步入智能设备时代，硬件+软件的生态体系对于基于芯片的智能设备至关重要。目前的 PC 产业以 windows+intel x86 架构生态体系为代表，基本在市场内形成了垄断。龙芯作为后起之秀想要在时代洪流中站稳脚跟，建立和扩大属于龙芯自己的生态体系是重中之重。

作为国产芯片的象征，龙芯自创立以来坚持自主研发，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 等架构的主要特点，推出了龙芯指令系统 LoongArch。自此，龙芯生态的硬件部分基本形成，而对软件生态的需求也日益增长，其中将各类软件移植到 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极具重要性。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 架构上成功实现。除此之外，还有更多操作系统尚且等待着移植到 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。

第二章 seL4 简介

本章主要基于对 seL4 官方资料的整理和理解, 结合在 seL4 移植过程中已经使用和在未来可能应用的内容进行介绍阐述。

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。

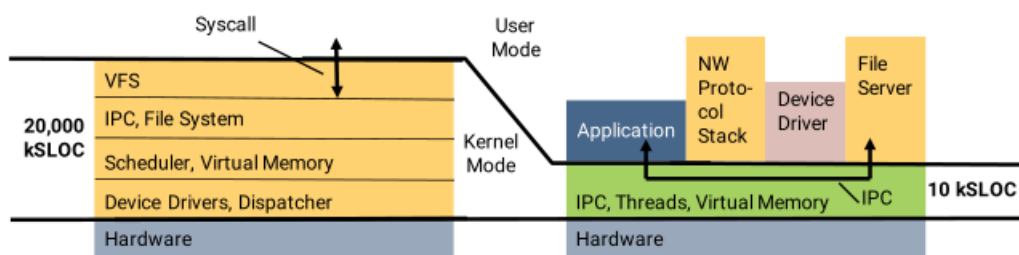


图 2 微内核和内核[15]

图 2 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, 每一层都向上提供了对下层实现的抽象。这样的模式下, 如果特权模式代码有一个漏洞被发现, 攻击者可以利用该漏洞在特权模式下运行攻击者的代码。

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, 必须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。

正因为以上原因, 微内核被设计来减少 TCB 的规模, 以此减少受攻击的概率。图 2 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

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

中，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，但是已经足以安全地复用硬件资源。微内核主要的功能是隔离，即作为程序在不受其他程序干扰的情况下正确安全地执行的沙箱。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%将被微内核设计完全消除，另外 55%将被减轻到足以不再被视为关键问题[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。

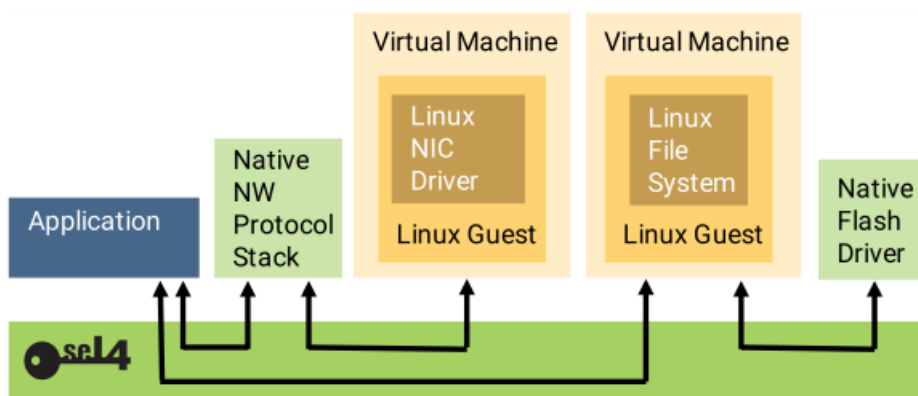


图 3 虚拟化技术将 Linux 提供的服务集成到本机

2.2 seL4 设计模式

由于项目关注内容有限，这里不介绍 seL4 的混合临界系统 MCS (mixed-criticality systems)，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密钥，作为访问令牌被 seL4 所采用。所有的内核服务都会受到访问控制；为了执行操作，应用程序必须调用其拥有的对所请求的服务具有足够访问权限的 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。



图 4 capability 是传递权限的密钥

一个 Capability³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。

seL4 实现了 10 种引用不同对象的 capability:

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)。

-
1. **Endpoints:** 用于执行受保护的函数调用;
 2. **Reply Objects:** 表示来自受保护的调用过程返回的路径;
 3. **Address Spaces:** 提供各种组件的沙箱(抽象硬件页表的薄包装);
 4. **Cnodes:** 用来存储代表组件访问权限的 `capability`;
 5. **Thread Control Blocks (TCBs):** 用来表示执行的线程;
 6. **Scheduling Contexts:** 代表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. **Untyped:** 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` 作访问控制的原因有:

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 在这时候就无法安全地实现。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，从他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 1 所示。

表 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 内核服务和对象

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:** 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。

2.2.2.1 内核对象

这里介绍内核实现的对象类型。应用程序可以调用这些对象的实例，内核提供的接口由这些对象提供。内核服务的创建和调用就是通过创建、操作和组合这些内核对象来实现的。seL4 内核对象如下：

CNodes: 存储 capability，赋予线程调用特定对象方法的权限。每个 CNode 都有固定数量的插槽 (Slot)，总是 2 的指数幂，这取决于 CNode 是在什么时候创建的。插槽可以是空的，也可以包含一个 capability。

Thread Control Blocks: 线程控制块 (TCBs)，在 seL4 中表示一个执行的线程。线程是调度、阻塞、解除阻塞等执行的单元，这取决于应用程序与其他线程的交互。

Endpoints: 端点对象促进线程之间的消息传递通信。IPC 是同步的：试图在端点上发送或接收消息的线程会阻塞，直到消息能够被传递。这意味着只有当发送方和接收方在端点汇合时才会发生消息传递，并且内核可以使用单个副本传递消息 (或者仅使用寄存器，不复制短的消息)。

Notification Object: 通知对象提供了一个简单的信令机制。Notification 是一个字大小的标志数组，每个标志的行为都类似于二进制信号量。其上操作有：seL4_Signal () 在单个操作中发送全体标志集的子集；seL4_Poll () 轮询检查

任何标志；`seL4_Wait()` 在收到信号前阻塞。通知 `capability` 可以是仅信号 (`signal-only`) 或仅等待 (`wait-only`)。

Virtual Address Space Objects: 虚拟地址空间对象，用于为一个或多个线程构造一个虚拟地址空间(又称 `VSpace`)。这些对象在很大程度上直接对应于硬件，因此是依赖于体系结构的。内核还包括 `ASID Pool` 和 `ASID Control` 对象，用于跟踪地址空间的状态。

Interrupt Objects: 中断对象使应用程序能够接收并确认来自硬件设备的中断。最初，`IRQControl` 有一个 `capability`，它允许创建 `IRQHandler` 的 `capability`。一个 `IRQHandler capability` 允许管理与特定设备关联的特定中断源。它被委派给设备驱动程序以访问中断源。`IRQHandler` 允许线程等待并确认单个中断。

Untyped Memory: 无类型内存是 `seL4` 内核中内存分配的基础。无类型内存 `capability` 有一个单一的方法，它允许创建新的内核对象。如果方法成功，调用线程将获得对新创建对象的 `capability` 的访问权。此外，无类型内存对象可以被分成一组较小的无类型内存对象，允许委托部分(或全部)系统内存。

2.2.2.2 内核服务——系统调用

`seL4` 内核为线程之间的通信提供了消息传递服务。这种机制也用于与内核提供的服务进行通信。有一种标准的消息格式，每条消息包含许多数据，可能还包含一些 `capability`。

线程通过在其 `capability` 空间内调用 (`invoke`) `capability` 来发送消息。当以这种方式调用 `endpoints`、`notifications` 或 `reply capability` 时，消息将通过内核传输到另一个线程。当调用内核对象的其他 `capability` 时，消息将以特定于内核对象类型的方式解释为方法调用。例如，使用正确格式的消息调用线程控制块(TCB) `capability` 将暂停目标线程。

从根本上说，我们可以认为内核提供了三个系统调用：`Send`、`Receive` 和 `Yield`。但是，还有基本的 `Send` 和 `Receive` 调用的组合和变体。一个重要的变体是 `Call` 操作，它由一个标准的 `Send` 操作和一个等待应答的 `Receive` 操作组成。

应答消息总是通过一个特殊的资源来传递，而不是使用标准的 IPC 机制;详情请参见下面的 `seL4_Call()`。

在内核对象上调用除 `endpoint capability` 和 `notification capability` 之外的方法是通过 `Send` 或 `Call` 来完成的，这取决于调用者是 `(Call)` 否 (`Send`) 需要内核的响应。通过使用 `libseL4` API 提供的函数，可以保证始终使用更合适的函数。`Yield` 系统调用不与任何内核对象关联，并且是唯一不调用 `capability` 的操作。在非 MCS 配置中，`Wait` 是 `Receive` 的同义词，因为两个调用都没有提供 `reply` 对象。

基本的系统调用如下：

`seL4_Yield()` 是唯一不需要使用 `capability` 的系统调用。它没收了调用它的线程的时间片的剩余部分，并导致内核调度器的调用。如果没有与调用者具有相同优先级的其他可运行线程，调用线程将立即用一个新的时间片调度。

`seL4_Send()` 通过指定的 `capability` 传递消息。如果调用的 `capability` 是 `endpoint capability`，并且没有接收方准备立即接收消息，则发送线程将阻塞，直到可以传递消息为止。接收对象不会返回错误代码或响应，这样能避免反向通道 `back channel` 问题。

`seL4_Recv()` ("`receive`") 用于线程通过 `endpoint` 或 `notification` 接收消息。如果没有发送方或 `notification` 处于挂起状态，则调用方将阻塞，直到可以传递消息或 `notification` 为止。此系统调用仅在 `Endpoint capability` 或 `Notification capability` 上才起作用，当尝试使用其他 `capability` 类型时将引发错误。

其余的系统调用是 `seL4_Send()` 和 `seL4_Recv()` 的变体和组合，可以有效地适应系统编程中的常见用例。

`seL4_NBSend()` 在 `endpoint` 上执行轮询发送。如果消息不能立即传递，也就是说，目标端点上没有等待接收者，则消息将被静默地丢弃。发送线程继续执行。与 `seL4_Send()` 一样，不会返回错误代码或响应。

`seL4_NBRecv()` 被线程用来检查 `notification` 对象上挂起的信号或 `endpoint` 上挂起的消息，而不会阻塞。此系统调用仅对 `endpoint capability` 和

Notification capability 起作用(类似普通 Recv), 当尝试使用其他 capability 类型时将引发错误。

seL4_Call() 结合了 **seL4_Send()** 和 **seL4_Recv()**, 并有一些重要的区别。调用将阻塞发送线程, 直到它的消息被传递并收到应答消息。当调用内核服务而不是 endpoint 的 capability 时, 使用 **seL4_Call()** 允许内核通过应答消息返回错误代码或其他响应。当发送的消息通过 endpoint 传递到另一个线程时, 内核执行与 **seL4_Send()** 相同的操作。接下来发生什么取决于内核配置。在非 MCS 配置中, 内核会在接收端 TCB 的一个专用插槽 (Slot) 中存储一个特殊的 reply capability。此 capability 是发送应答消息并唤醒调用者的单一使用 capability, 这意味着一旦调用该应答消息, 内核就会使其失效。对于这两个变体, 在调用 reply 对象的 capability 之前, 调用线程都会被阻塞。

seL4_Reply() 用于响应 **seL4_Call()**, 方法是调用 reply capability, 这个 capability 是由 **seL4_Call()** 系统调用生成并存储在应答线程 TCB 的专用槽中的。它具有与使用 **seL4_Send()** 调用 reply capability 完全相同的行为。

seL4_ReplyRecv() 由 **seL4_Reply()** 和 **seL4_Recv()** 组合而成。它的存在主要是为了提高效率。通常情况下, 响应一个请求并等待下一个请求可以在单个内核系统调用中执行, 而不是两个。从应答到接收阶段的转换也是原子性的。

seL4_Wait() 类似于 **seL4_Recv()**; 在非 MCS 配置上, 它们实际上是同义的。

2.2.2.3 内核内存分配

seL4 微内核不会动态地为内核对象分配内存。相反, 对象必须通过 Untyped memory capability 从应用程序控制的内存区域显式创建。为了创建新对象, 应用程序必须具有对内存的显式权限(通过 Untyped memory capability 获得), 而所有对象在创建后都将消耗固定数量的内存。这些机制可用于精确控制应用程序可用物理内存的特定数量, 包括能够在应用程序或设备之间强制物理内存访问的隔离。除了那些由硬件(对虚拟 ASID 的处理强加了固定数量的地址空间。这个限制将在 seL4 的未来版本中消除)决定的限制外, 内核中没有任何的资源限制, 因此可以避免许多通过资源耗尽而导致的 Dos 攻击。

在引导时, `seL4` 预先分配内核本身所需的内存, 包括代码、数据和堆栈部分(`seL4` 是一个单一的内核堆栈操作系统)。然后创建一个初始用户线程(带有适当的地址和 `capability` 空间)。然后内核将所有剩余的内存以 `Untyped memory capability` 的形式交给初始线程, 并将一些额外的 `capability` 交给内核对象(引导初始线程需要这些 `capability`)。然后可以使用 `seL4_Untyped_Retype()` 方法将这些 `Untyped Memory` 区域分割成更小的区域或其他内核对象; 创建的对象被称为原始 `Untyped memory` 对象的子对象。使用 `seL4_Untyped_Retype()` 创建对象的用户级应用程序接受对结果对象的所有权限。然后, 它可以将其对该对象拥有的全部或部分权限委托给它的一个或多个客户端。

`seL4` 内核还允许重用(reuse)无类型内存(`Untyped Memory`)区域。只有当没有悬空引用(即 `capability`)留给该内存中的对象时, 重用一个区域的内存才是被允许的。内核会跟踪派生 `capability` (`capability derivations`), 即由方法 `seL4_Untyped_Retype()`、`seL4_CNode_Mint()`、`seL4_CNode_Copy()` 和 `seL4_CNode_Mutate()` 生成的子进程。这样生成的树的结构称为 `capability` 派生树(`capability derivation tree`, `CDT`。尽管 `CDT` 在概念上是一个独立的数据结构, 但它是作为 `CNode` 对象的一部分实现的, 因此不需要额外的内核元数据)。例如, 当用户通过 `retype untyped memory` 创建新的内核对象时, 新创建的 `capability` 将作为 `untyped memory capability` 的子 `capability` 插入到 `CDT` 中。

对于每个 `Untyped Memory` 区域, 内核保留一个水印, 记录该区域之前被分配了多少空间。当用户请求内核在非类型内存区域中创建新对象时, 内核将执行两种操作之一: 如果该区域中已经分配了对象, 内核将在当前水印级别上分配新对象, 并修改水印。如果先前在区域中分配的所有对象都已删除, 内核将重置水印, 并再次从区域的开始分配新对象。

最后, `CNode` 对象提供的 `seL4_CNode_Revoke()` 方法销毁了从作为参数的 `capability` 派生的所有 `capability`。撤销内核对象的最后一个 `capability` 会触发对现在未引用的对象的销毁操作。这只会清除它、其他对象和内核之间的所有的内核依赖关系。

通过对 `untyped memory` 对象的原始 `capability` 调用 `seL4_CNode_Revoke()`，用户删除所有 `untyped memory` 对象的子对象——也就是说，所有指向 `untyped memory` 区域对象的 `capability`。因此，在此调用之后，对 `untyped memory` 区域内的任何对象都没有了有效的引用，该区域可以安全地 `retyped` 和 `reused`。

在 `retype untyped memory` 时，了解对象将需要多少内存是必要的。对象大小在 `libseL4`⁴ 中定义。`Cnodes` 和 `Untyped Objects` 都有变量大小。当将无类型内存 `retype` 到 `CNodes`，或者将一个 `untyped Object` 分解为更小的 `untyped Object` 时，`seL4_Untyped_Retype()` 的 `size_bits` 参数被用来指定生成对象的大小。对于所有其他对象类型，大小是固定的，`seL4_Untyped_Retype()` 的 `size_bits` 参数被忽略。

表 2 对象大小

| 类型 | Size_bits 参数含义 | 对象字节大小 |
|----------------------|--------------------------------|--|
| <code>CNode5</code> | $\text{Log2}(\text{Slot 槽位数})$ | $2^{\text{size_bits}} \times 2^{\text{seL4_SlotBits}}$ |
| <code>Untyped</code> | $\text{Log2}(\text{字节数})$ | $2^{\text{size_bits}}$ |

对 `seL4_Untyped_Retype()` 的单个调用可以将单个 `Untyped Object` `retype` 为多个对象。要创建的对象的数量由它的 `num_objects` 参数指定。所有创建的对象必须是由 `type` 参数指定的相同类型。对于大小可变的对象，每个对象也必须具有相同的大小。如果所需内存区域的大小(通过对象大小乘以 `num_objects` 计算)大于 `Untyped object` 的剩余未分配内存，则会导致错误。

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`

⁴ `libseL4` 是 `seL4` 官方提供的组件。

⁵ `seL4_SlotBits` 在 32 位机器上是 4，在 64 位机器上是 5。

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。

2.2.4 IPC

seL4 微内核为线程之间的通信提供了消息传递 IPC 机制。这个机制也用于与内核提供的服务进行通信。消息是通过调用内核对象的 capability 来发送的。发送到端点 Endpoints 的消息的目的地是其他线程，而发送到其他对象的消息则由内核处理。

2.2.4.1 消息寄存器

每条消息包含一些消息字和一些可选的 capability。通过将消息字放入线程的消息寄存器，可向线程发送消息字或从线程接收消息字。消息寄存器中，前几个消息寄存器使用物理 CPU 寄存器实现，而其余的消息寄存器由一个称为 IPC 缓冲区的固定内存区域支持。这样设计是因为非常短的消息如果经过内存效率将会很低。IPC 缓冲区将被分配给调用线程。

每个 IPC 消息也有一个 tag 标记(`seL4_MessageInfo_t` 结构体)。标记由四个字段组成: `label` 标签、消息长度、`capability` 数量(`extraCaps` 字段)和 `capsUnwrapped` 字段。消息长度和 `capability` 数量决定了发送线程希望传输的消息寄存器和 `capability` 的数量, 或者决定了实际传输过来的消息寄存器和 `capability` 的数量。`label` 标签不经内核解释, 并作为消息的第一个数据传递。例如, `label` 标签可以用来指定所请求的操作。`capsUnwrapped` 字段仅在接收端使用, 用于指示接收 `capability` 的方式。

表 3 `seL4_IPCBuffer` 结构体字段⁶

| 类型 | 名字 | 描述 |
|---------------------------------|---------------------------|---|
| <code>seL4_MessageInfo_t</code> | <code>tag</code> | 信息 tag 标签 |
| <code>seL4_Word[]</code> | <code>msg</code> | 消息内容 |
| <code>seL4_Word</code> | <code>userData</code> | 结构体的基址 |
| <code>seL4_CPtr</code> | <code>caps</code> | 待传输的 <code>capability</code> |
| <code>seL4_CapData_t</code> | <code>badges</code> | endpoint <code>capability</code> 接收到的 badge |
| <code>seL4_CPtr</code> | <code>receiveCNode</code> | 接收插槽对应的 <code>CNode</code> 指针 |
| <code>seL4_CPtr</code> | <code>receiveIndex</code> | 接收插槽的指针 |
| <code>seL4_Word</code> | <code>receiveDepth</code> | 要使用的 <code>receiveIndex</code> 的 bit 数 |

2.2.4.2 端点 Endpoint 上的 IPC

端点允许在两个线程之间传输少量的数据和 `capability` (即 IPC 缓冲区)。端点对象通过 `seL4` 系统调用直接调用。

`IPC Endpoint` 是同步和阻塞的。端点对象可以对发送或接收的线程进行排队。如果没有接收方就绪, 执行 `seL4_Send()` 或 `seL4_Call()` 系统调用的线程将在队列中等待第一个可用的接收方。如果没有发送方准备好, 执行 `seL4_Recv()` 系统调用或 `seL4_ReplyRecv()` 的后半部分的线程将等待第一个可用的发送方。在没有写入 `capability` 的情况下尝试 `send` 或 `call` 将失败并返回错误。在 `send` 的情况下, 错误被忽略(内核不允许应答, 没有办法知道发送是否已经失败)。另

⁶ `badges` 和 `caps` 是结构体的同一个字段, 一个用于发送, 一个用于接收。

一方面，使用没有 Read 权限的 endpoint capability 调用 `seL4_Recv()` 将引发错误，这是为了将错误消息与通过端点从另一个线程接收到的正常消息区分。

2.2.5 Notifications

Notifications 是一种非阻塞的信号量，它表示逻辑上一组二进制信号量。

2.2.5.1 Notification 对象

Notification 对象仅包含一个 notification 数据字。该对象支持两种操作：`seL4_Signal()` 和 `seL4_Wait()`。

类似于 Endpoint Capability, Notification capability 可以也可以被 `seL4_CNode_Mutate()` 或 `seL4_CNode_Mint()` 函数添加标记，但已被标记的 notification capability 无法取消标记，重新标记或创建具有不同标记的子 capability。

2.2.5.2 发信号 (Signal)、轮询 (Poll) 和等待 (Wait)

`seL4_Signal()` 把被调用 notification capability 的 badge 和 notification 数据字按位求或，并把结果更新到 notification 数据字。它还会阻塞等待 notification 数据字的首个线程。因此，`seL4_Signal()` 的工作方式类似于同时发送多个信号量（由 badge 的位指示）。如果信号发送方的 capability 没有标记或 0 标记，该函数只会唤醒等待 notification 数据字的首个线程。

`seL4_Wait()` 函数工作方式类似于在信号量集合上 select 方式等待。在调用 `seL4_Wait()` 时，如果 notification 字是 0，调用者将被阻塞。如果 notification 字不是 0，`seL4_Wait()` 立即将之前的 notification 数据字返回，并将 notification 数据字置 0。

`seL4_Poll()` 函数和 `seL4_Wait()` 函数工作方式相似，它们之间的区别是，`seL4_Poll()` 函数如果发现没有挂起的信号量（notification 数据字为 0），将立即返回，不会阻塞。

如果调用 `seL4_Signal()` 时，有线程等待着 `notification` 对象，队列中第一个线程会接收到 `notification` 数据字。其他线程继续等待，直到下一次 `notification` 数据字到来。

2.2.5.3 绑定 `notification` 信号量

调用 `seL4_TCB_BindNotification()`，可以将 `notification` 对象和 TCB 对象一对一绑定。如果 `notification` 和 TCB 绑定，即使线程正在 IPC 端点接收信号，`notification` 信号也会被发送。软件应该检查 `badge` 值区分收到的消息是 `notification` 信号还是 IPC 消息，如果为 `notification` 的 `capability badge` 保留某些特殊的值，就可以确定消息来源是 `notification` 还是 IPC 消息了。

一旦 `notification` 数据字和 TCB 绑定，只有被绑定线程可以执行对相应 `notification` 的 `seL4_Wait()` 操作。

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]，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。

就 `seL4` 自身的测试和简单应用而言，`seL4` 官方提供了两个项目：`seL4-tutorials` 和 `seL4-test`。前者用来学习 `seL4` 设计思想和技术细节，后者用来测试 `seL4` 是否可以在机器上成功运行。经过综合考虑，我们小组决定将 `seL4-test` 移植到 `LoongArch`，这样能比较好地评估移植是否成功。除此之外，因为 `seL4-test` 是由 `seL4kernel` 和其他 `seL4` 提供的系统组件共同组成的，在我们移植成功后，可以在此基础上进行一些扩充，比如建立一些简单的示例，或者继续完成 `seL4-tutorials` 的移植。

第三章 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 设计和其他设计工作。除此之外，本章最后一节还有移植工作到目前为止开发情况的运行展示。

3.1 内核设计

3.1.1 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，物理地址默认直接等于虚拟地址的 $[PALEN-1:0]$ 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 微内核加载时只有头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，具体会在下面介绍，此模式下翻译地址时优先看能否按照直接映射模式进行翻译，无法进行后再按照页表映射模式进行翻译。在 seL4 中基本用到的都是页表映射模式，只有在启动阶段的 elfloader 中我们用了直接映射窗口。

3.1.1.1 虚拟地址结构

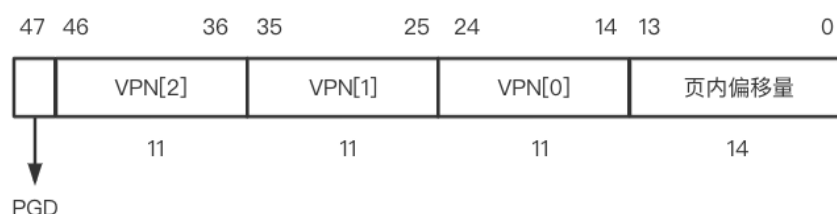


图 5 虚拟地址结构

LoongArch 物理地址空间范围是： $0 \sim 2^{PALEN} - 1$ 。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。

因此在 3A5000 上虚拟地址共 48 位，最高位为 PGD。我们设计了三级页表，

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。

3.1.1.2 虚拟地址空间

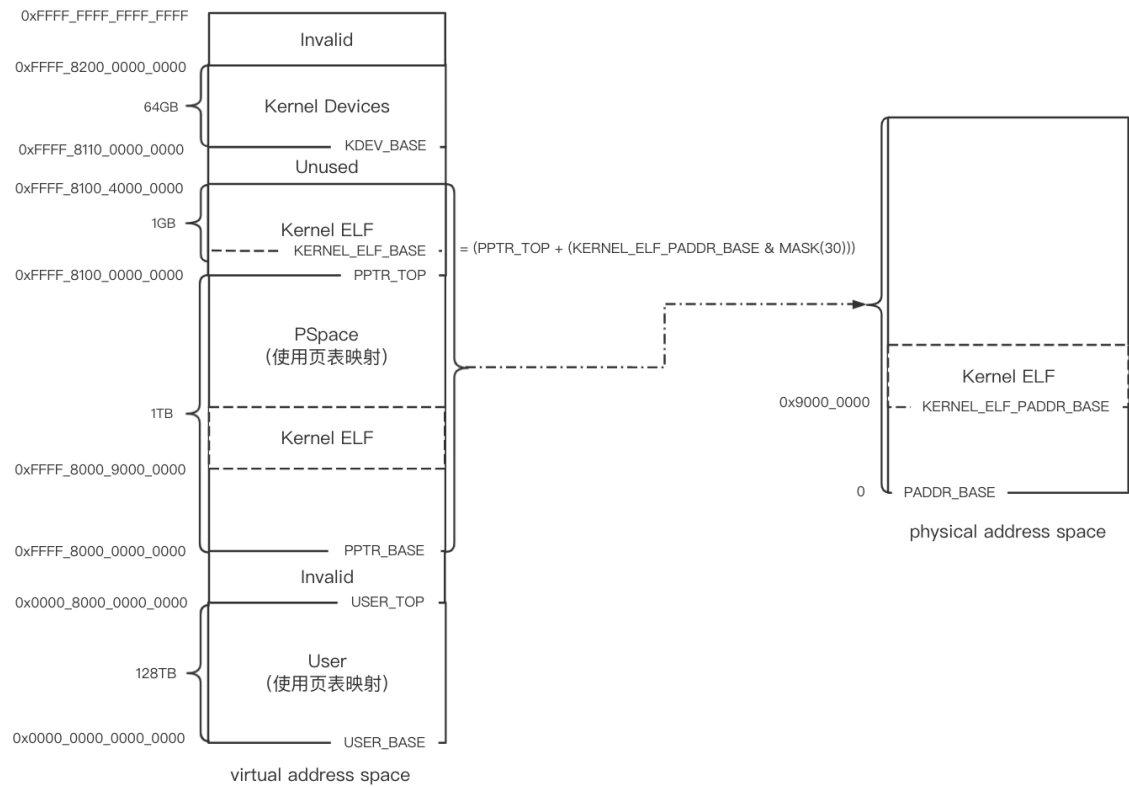


图 6 虚拟地址空间

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，因为我们使用的机器 3A5000+7A1000 桥片目前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，实际我们只使用了其中的 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。

3.1.1.3 内核装载流程



图 7 seL4 在 LoongArch 上的内核启动流程

目前 UEFI BIOS 装载内核时，会把从内核 elf 文件获取的入口点地址抹去高 32 位使用。比如 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 这么

做是因为它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 8），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，我们将 kernel 的装载地址定位 0x9000_0000（参考图 8），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 8 龙芯 3 号处理器+龙芯 7A 桥片地址空间划分

3.1.1.4 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, 我们使用到了 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)。我们将 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.TLBRENTY 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 3.1.3.2

3.1.2 用户上下文和通信消息设计

3.1.2.1 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。用户上下文信息与指令集架构的寄存器有关。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 所示。

表 4 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_crmd | — | 当前模式信息 |
| 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 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。

3.1.2.2 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。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, 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 d 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 }

3.1.3 中断与例外处理

3.1.3.1 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.TLBRETRY)，机器错误例外入口 (CSR.MERRETRY) 和普通例外入口

(CSR. EENTRY)。目前，seL4 仅实现了 TLB 重填例外和普通例外。

下面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 5 所示。

表 5 TLB 重填例外涉及的重要寄存器

| 寄存器名 | 寄存器功能 |
|-----------------|---------------------------|
| CSR. TLBREENTRY | 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 6 所示。

表 6 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 |

3.1.3.2 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. TLBREENTRY。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 3.1.1.4。

- 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。

3.1.3.3 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;

将普通例外入口 trap_entry 写入 CSR.EENTRY。在 k_entry 段中定义 trap_entry 陷入函数, 在 lds 中将 k_entry 段 16K 对齐;

配置 CSR.ECFG.LIE, 使能 13 个中断源。中断还处于调试阶段, 后续仅使能需要相应的中断;

扩展中断暂未开启;

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_KS0`, 用于保存当前上下文的指针（与 `riscv` 的 `sscrartch` 寄存器功能相同）；

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 `seL4-LoongArch` 用户上下文信息描述；

设置 `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；（各处理函数待完善，目前框架已经完成）

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 `seL4-LoongArch` 用户上下文信息描述；

`ertn` 返回。

3.2 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 9 所示。

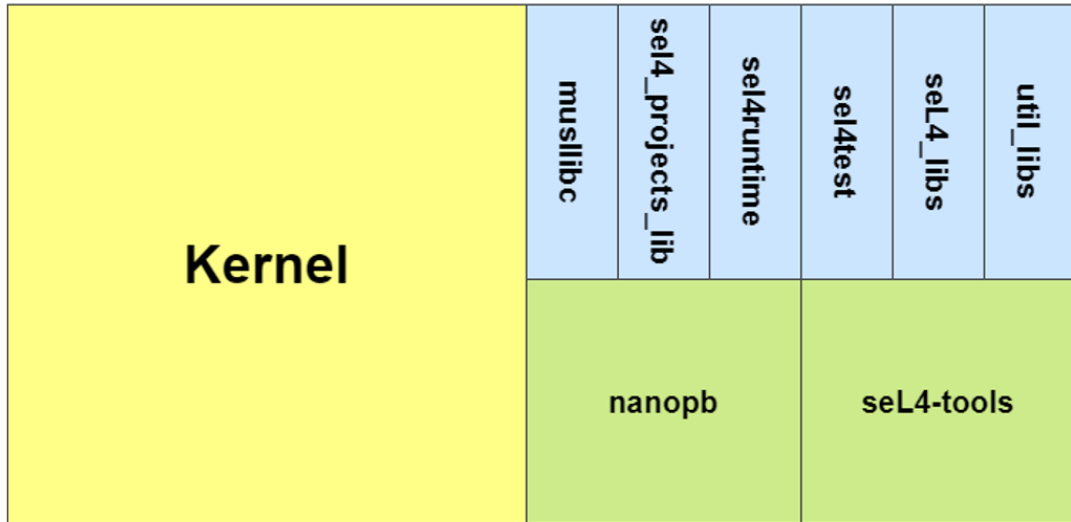


图 9 seL4-test 项目结构

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 nanopb⁷。各部分解释如下：

表 7 seL4-test 结构描述

| 组件 | 描述 |
|-------------------|--|
| Kernel | seL4 微内核 |
| 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

⁷ 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。

CMakeLists。Cmake 过程分为预设置、kernel 设置和组件设置三部分。

3.2.1 预设置

预设置主要是为 kernel 设置准备全局变量和文件路径，以及一些其他的准备工作。整个 cmake 构建过程中有很多全部变量最后将通过 cmake 语法及脚本为具体构建的项目生成配置头文件⁸。seL4 现有的 cmake 结构扩展性较强，比较容易添加 LoongArch 架构分支的设置，实现细节不再详述。

预设置主要设置的一些 cmake 变量见表 8。

表 8 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 |
| 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 |

⁸ 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。

| | |
|--------------------------------|-------------------------------|
| 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 |

需要注意的是，项目移植目前没有考虑 seL4 的 MCS 和 SMP 多核配置。

除配置通用的架构变量，添加龙芯交叉编译器与架构变量的对应关系之外，我们在 kernel 的 plat 文件夹⁹添加了 3A5000 文件夹，在里面声明 3A5000 对应 dts 文件的路径以及中断控制器¹⁰。其他架构同系列机器不同的设备树设置需要的覆盖文件也是放置于此，因目前只针对 3A5000 平台的移植，所以没有 dts 覆盖文件。dts 覆盖文件的使用通过 cmake 的 file 命令读写完成，最后通过系统

⁹ 里面存储平台相关 cmake 的配置文件。

¹⁰ 目前中断尚未完全实现，这里用 dummy 填充。

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 架构的一些机器。

3.2.2 Kernel 设置

这里是 seL4 微内核相关的主要配置。下表中的前 6 个变量代表的路径指向 6 个脚本(库)，seL4-test 构建编译过程中会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，我们也对脚本做了部分修改，但是这并非我们的主要工作内容，这里不再叙述。

表 9 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 |
| 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 ¹¹ | OFF |
| KernelDebugBuild | ON(在内核中启用调试工具) |
| HardwareDebugAPI ¹² | 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 |

¹¹ 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。

¹² 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。

| | |
|---|--|
| KernelUserStackTraceLength | 16 |
| KernelOptimisation | -O2 |
| KernelFWholeProgram ¹³ | OFF |
| KernelDangerousCodeInjection ¹⁴ | OFF |
| KernelDebugDisablePrefetchers ¹⁵ | OFF |
| KernelSetTLSBaseSelf ¹⁶ | OFF |
| KernelClzNoBuiltin | OFF |
| KernelCtzNoBuiltin | OFF |
| <hr/> | |
| | -D__KERNEL_64__ - |
| CMAKE_C_FLAGS ¹⁷ | march=loongarch64 -mabi=lp64d - mtune=loongarch64 -Wno- error=array-bounds |
| CMAKE_CXX_FLAGS | 同上 |
| CMAKE_ASM_FLAGS | 同上 |
| <hr/> | |
| | -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_EXE_LINKER_FLAGS | |
| CMAKE_BUILD_TYPE | RelWithDebInfo |
| Linker_source | lds 文件路径 |

出于简洁性考虑，我们不开启 FPU，也不开启单双精度浮点扩展标识。按页表设计，在这里设置内核页表级数 KernelPTLevels=3。按设计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

¹³ 此选项打开后，链接内核时启用-fwhole-program。

¹⁴ 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。

¹⁵ 如果开启可能需要改写与平台相关的禁止预取的逻辑。

¹⁶ 启用后会构建没有 capability 的内核。

¹⁷ 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。

等 4 个 FLAGS 中设置。而链接用到的 lds 文件为 `common_loongarch.lds`。lds 的具体设计这里不再叙述，请查看项目文件。

3.2.3 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项目文件。

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。

3.3 其他移植工作

3.3.1 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。针对龙芯架构，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 `0xffffffff`。
- 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。

3.3.2 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 3.3.1 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 文件中加入也将 "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 初始化驱动设备的中层函数，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。

3.4 运行展示

3.4.1 Qemu 虚拟机运行

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 (0x0000000000000000)
SetUefiImageMemoryAttributes - 0x00000000FE920000 - 0x0000000000040000 (0x0000000000000000)
kernel argc: 2, argv: 0xFE6F0000, bpi: 0xFFFFDA418.
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 10 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 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 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 TLBREENTRY；
- 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..17ffffff]
reserved virt address space regions: 3
[ffff800090000000..ffff800092030000]
[ffff800092030000..ffff800092030528]
[ffff800092031000..ffff800092477000]
Booting all finished, dropped to user space
```

图 11 qemu 模拟-sel4 kernel

elfloader 通过 a0~a5 传递进来 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 elfloader 中已经配置, 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 11 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。

第四章 回顾与展望

回顾过去的 3 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。截至初赛结束，本团队工作进展：移植了 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 内核特点设计并实现了 seL4-test 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-test 可执行 elf 文件。

团队开发思路：首先完成 seL4 的入门教程，结合论文了解上层设计思想。然后阅读底层源码，并结合 riscv 教程、xv6-mips 源码、Linux-LoongArch 源码编写底层代码。为了让项目编译起来，队员阅读并修改了 cmake 文件。然后利用 qemu 模拟器调试内核。

项目推进难点：seL4 项目除了内核源码，还有更多工具需要学习，例如用 cmake 和 ninja 编译项目，用 Linux 的 dts 传递设备信息，用自带的 elfloader 引导内核等。相较于 xv6 等教学操作系统内核，seL4 微内核的架构相关文档或技术博客非常少，尽管我们努力地学习 seL4 项目结构，但也经常因为理解不足而重复修改源码。相对于 x86、riscv 而言，LoongArch 架构的技术博客较少，可供参考的示例程序不多。

决赛阶段计划：完善线程相关的模块，完成微内核的移植。移植 seL4 官网的微内核测试程序，确保移植工作的正确性。了解 seL4 形式化验证的工作。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 linux 内核开发经验可谓“一穷二白”，截至 5 月末，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 模拟器等知识，并将 seL4 微内核调试到线程调度的位置。

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，所谓精诚合作，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 seL4 的调试细节。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.