seL4 操作系统编程

elastos.org &

Tongji University High Dependable Operating System Group

2015/7

目录

1 seL4 微内核介绍

seL4 是一个操作系统微内核，也就是说，seL4 本身不是一个完整的操作系统。因为它是微内核，所以它提供很有限的 API，没有提供象传统的操作系统 Linux 那样的内存管理、页内外交换、驱动程序等等。

seL4 是一组基于微内核架构的操作系统内核，澳大利亚研究组织 NICTA http://nicta.com.au/ 创造了一个新的 L4 版本,称为 Secure Embedded L4(简写为 seL4)，宣布在世界上率先开发出第一个正规机器检测证明（formal machine-checked proof）通用操作系统。

seL4 微内核设计针对实时应用，可潜在应用于强调安全和关键性任务的领域内，如军用和医疗行业。形式证明在较小的内核中已经实现，但这次是首次在执行复杂任务的操作系统内核中实现。研究发现常用的攻击方法对 seL4 无效，如恶意程序经常采用的缓存溢出漏洞。

seL4 是第三代微内核操作系统，它基本是可以说是基于 L4 的，它提供了虚拟地址空间（virtual address spaces）、线程（threads）、IPC（inter-process communication）

受 EROS、KeyKOS、CAP 等操作系统设计的影响，seL4 的控制机制是基于 capabilities 的，capabilities 机制提供了访问内核对象（kernel objects）的方法，这种机制使得 seL4 与其它 L4 比起来，显示出一定的高效率。

1.1 基础概念

1.1 1 内核对象（Kernel Objects）

每个内核动态数据结构都表述为内核对象，内核并不为这些内核对象分配内存，seL4 一旦起动，内核就不再需要新的内存，所有内存都是在用户态申请的。用户态程序为了申请内存，需要提供原始内存（untyped memory），内存对象最终存储在这个原始内存中。

原始内存无法直接被应用程序访问（access）。

1.1.2 句柄（Capabilities）

capability 概念上类似于句柄（Handle），在操作系统这样的系统软件设计中，有一个可靠性的设计原则，就是下层软件不信任上层软件，而上层软件要无条件信任下层软件。那么，下层软件如何把自己的实现细节隐藏，不让上层软件通过“简单”方法就可以直接操作下层软件的资源呢，句柄是一个常用的设计。

一个句柄指向一个内核对象，内核对象也只能通过与其绑定的句柄来操作它，无法直接访问。大多数的内核对象都有方法，用来操作这个对象的数据结构，就象是 c++中的封装中的 protect 数据结构，方法自身可以访问数据结构中的数据，但是，使用这个对象的其它程序，只能通过 API 与这个对象打交道。

举例来说，用户来以通过调用一个原始内存 untyped-memory 句柄的 invoke 方法，用来在这个原始内存上创建内核对象。

句柄存储在 CNodes 中，CNodes 本身也是一个内核对象。

有些内核对象只有方法，但自身没有什么状态（state）信息，这类对象的目的是为了让应用程序与内核打交道，这类对象没有自身的存储实例。

下面泛义地讨论一下什么是句柄？为什么会有句柄？

句柄是一个概念，Handler 是一个句柄，seL4 中的 Capability 也是一个句柄，打开文件，fopen 得到的那个 FILE *，open 得到的那个 int 都是句柄。

从广义上，能够从一个数值（或数据结构）拎起一大堆数据的东西都可以叫做句柄。句柄的英文是“Handle”，本义就是“柄”，只是在计算机科学中，被特别地翻译成“句柄”，其实还是个“柄”。从一个小东西拎起一大堆东西，这难道不像是个“柄”吗？

然后，指针其实也是一种“句柄”，只是由于指针同时拥有更特殊的含义——实实在在地对应内存里的一个地址——所以，通常不把指针说成是“句柄”。但指针也有着能从一个 32 位的值（在 32 位硬件平台上）引用到一大堆数据的作用，这不是句柄又是什么？

一个操作系统中，可能有许多内核对象（这里的对象不完全等价于“面向对象程序设计”一词中的“对象”，虽然实质上还真差不多），比如打开的文件，创建的线程，程序的窗口，等等。这些重要的对象肯定不是 4 个字节或者 8 个字节足以完全描述的，他们拥有大量的属性。为了保存这样一个“对象”的状态，往往需要上百甚至上千字节的内存空间，那么怎么在程序间或程序内部的子过程（函数）之间传递这些数据呢？拖着这成百上千的字节拷贝来拷贝去吗？显然会浪费效率。那么怎么办？当然传递这些对象的首地址是一个办法，但这至少有两个缺点：

- 暴露了内核对象本身，使得程序（而不是操作系统内核）也可以任意地修改对象的内部状态（首地址都知道了，还有什么不能改的？），这显然是操作系统内核所不允许的。上面不是已经介绍过操作系统的可靠性设计原则了吗？
- 操作系统有定期整理内存的责任，如果一些内存整理过一次后，对象被搬走了怎么办？

1.1.3 虚拟地址空间（Virtual Address Spaces）

传统上，象 Linux 这样的操作系统，虚拟地址空间是针对进程（Process）来说的，每个进程工作地独立的地址空间中，进程为其中的计算提供隔离的计算容器。Java 虚拟机中的 class-loader 也提供了一个计算的隔离能力，Linux 的 cgroup 也提供了一种计算隔离能力。因为不同的计算隔离能力，产生了很多的内存管理措施。

seL4 内核没有虚拟地址空间管理的能力，它也没有进程这样的概念，换句话说，你是自己想玩 Windows CE slot 那样的进程，还是 Linux 那样的完整的概念的进程，那是在 seL4 上搭建的计算模型支持层的事。seL4+Elastos 中的 Elastos 就是样的一个计算模型支持层。

Windows CE 只能管理 512MB 的物理内存和 4GB 大小的虚拟地址空间。在 Windows CE 4GB 的虚拟地址空间中，从 0x00000000 到 0x41FFFFFF 由所有应用程序使用。这块地址空间分成 33 个槽(Slot)，每个槽占有 32MB 的地址空间。槽 0 由当前占有 CPU 的进程使用。槽 1 由 XIP DLL 使用。其它槽用于进程使用，每个进程占用一个槽，进程彼此不能够随意访问。因为槽 0 只有 32MB，因此每个进程运行时只有 32MB 的虚拟地址空间。

seL4 没有对虚拟地址空间的定义，虚拟地址与物理地址的影射（map）是被用户态的程序自己管理的。例如，一个 page fault 中断触发的异常是被送到定义在用户态空间的处理线程用户态页机制（user-level pager），这个页处理机制决定这个缺页中断如何处理，你是映射到一块物理内存，还是做什么处理，seL4 是不管的。

用户态的桢（User-level frame）是唯一可以在用户态被直接访问的内存，它们通过原始内存（untyped memory）创建而来。在这些帧被访问之前，它们需要被映射到虚拟地址空间，地址空间通过创建、链接页表对象得到，地个地址空间是一个指向最上层页表的句柄可能（a capability to its top-level page table）标识的。

1.1.4 线程（Threads）

线程（Thread）是被内核调度的执行单位。线程被创建后，需要为其指定内存地址空间，它在运行的过程中，可以改变这个指定的内存地址空间。一段内存地址空间，可以被0个、一个或多个线程指定，这些线程没有主线程、原始线程之类的概念。

seL4中，一个象Linux这样的传统的操作系统进程的概念被分解为：地址空间、虚实映射机制、程序及原始数据（装到内存里的那些用户自己写的逻辑）、执行线索等等。

seL4的每个线程对应内存中的一个线程控制块（TCB，thread control block）对象，没有内核线程这样的说法，也就是说，没有一个线程是只工作在内核态中的，除非它是一个哑（idle）线程，所有系统调用都是工作在调用它的那个线程的用户态空间中的。

1.1.5 进程间通信（Inter-Process Communication，IPC）

IPC发生在端点（endpoint）间。一个端点是一个独立的内核对象，拥有端点的线程通过调用端点的句柄（endpoint capability）发送或接收IPC消息。当一个端点被调用（发送消息Send或接收消息Receive），IPC消息传输立即发生在这两个线程（发送者与接收者）之间。IPC只能发生在处理同一个端点的两个线程之间。

IPC在语义上并不要求发生请求与应答的双方在一个地址空间或不在一个地址空间。

IPC的端点有同步与异步的说法，当访问同步端点，当发出请求或接收请求，线程将被阻塞，直到消息发出（此时接收者需要要接收），或有消息传过来（当然是请求者把消息传过来，它发了请求）。

消息（Message）最大480字节（120个消息字，每个消息字32比特）。

发送一个异步消息从来不被阻塞,异步消息只能是一个消息字,即32比特,4个字节。这个消息被存储在这个端点，端点上不能存储多个消息，如果多个消息被发送，这些消息将被通过位或（bitwase OR operation）运算为新的异步消息。

1.1.6 设备驱动与中断（Device Drivers and IRQs）

微内核操作系统的驱动通常不在内核中实现，为了让用户态中的程序可以访问设备寄存器，内核向用户态程序提供了设备桢（device frame），这些设备桢覆盖了设备的寄存器端口地址空间（memory-mapped registers）。

中断被异步递送到侦听这个中断的驱动程序端点。seL4内核允许设置哪些端点侦听哪些中断，这些端点也被用来设置中断的使能与告知（enable、disable、acknowledge）。

1.1.7 抢占（Preemption）

在处理一个事件时，seL4不能直接被打断，在内核执行时，不允许发生中断。为了防止高优先级的中心被延迟,seL4在需要长时间操作的事务(long-running operation)中，设置是抢占点（preemption point）。在内核执行到抢占点时，将检查被挂起的中断，如果检查到有挂起的中断，触发一个抢点异常（preemption exception），这个异常被传给内核入口函数（the main kernel entry function），在那里，中断被触发，然后我们再次进入内核处理这个中断。处理完这个中断后，内核返回到用户态，原来的系统调用被重新执行，然后这个长事务被继续执行。

1.2 零碎信息

1.2.1 L4 操作系统综述

http://hi.baidu.com/l4oss/item/a0c987973e7c1ecfb725310e

关于 L4 系统，一般来讲，目前公认的 L4 系统有 2 个特点，Fast IPC 和 Sigma0 协议。Sigma0 是一种基于 IPC 的内存管理协议，使用 Sigma0，内存管理呈现出一种层次状。举例：有 A 和 B 两个程序，如果程序 B 想使用程序 A 的内存。如果使用 Sigma0，那么很容易实现，只要把 A 设置为 B 的 pager,并提供 B 的 page fault handler 程序就可以。在这种情况下，程序 A 和程序 B 依然具有不同的 Address Space，A 和 B 之间互相隔离（关于这种层次式的内存管理，请参考 The sawmill framework for virtual memory diversity http://www.cse.unsw.edu.au/~kevine/pubs/acsac01.pdf ）。 但是如果使用 Linux 来实现这种模式，除非使用 share memory，我想不出其它更好的办法，但是 share memory 使得 A 和 B 之间有的 Address Space 有了交集，从 security 和 safety 两个方面来讲，都不是很好的解决方法。近年来，越来越多的 L4 系统开始支持一种新的特性-Capability，Capability 是为了提高操作系统安全性而设计的，Capability 和要访问的 Resource 之间的关系类似于文件描述符号和文件之间的关系一样，要访问一个 Resource，必须通过 Capability 来进行，Capability 里面规定了那么资源可以被访问等安全特性，Capability 允许被 grant（从一个用户转移到另外一个用户），总之，Capability 是比 Access Control List 更好的一种增强系统安全性的方法。

本书中把“文件描述符”这样的一个代表一个实际资源，但自身只是个描述，的数据结构，称为句柄，它的英文是 Handle。我们把 seL4 中的 capability 也称为句柄。

处于活动状态的项目：

1. PikeOS/ELinOS 德国 SYSGO AG 公司的商用非开源系统,它提供了很好的 Resource Isolation 机制，使用 ParaVirtualization，让每一个 OS Personality 运行在一个 VM 里面，可以支持 Java 和 Ada 的应用程序。PikeOS 不但具有 Spatial Isolation，还具有很好的 Temporal Isolation，因此也支持 Real-Time application。ELinOS 是移植到 PikeOS 的嵌入式 Linux 系统（2.4 和 2.6），支持众多硬件平台和开发板。PikeOS 通过 ARINIC 653，DO-178B 认证，因此被用于军工航天等 Safety-Critical 和 Secure Application。PikeOS 是从 98 年开始开发的，近几年 Sysgo 已经成为欧洲增长最快嵌入式厂商，ELinOS 也成为比较流行的嵌入式 Linux 开发环境。因为是商用系统，可参考的资料很少。(www.sysgo.com，http://en.wikipedia.org/wiki/PikeOS）
2. Fiasco/L4Env/L4Linux Fiasco 是 TUD Operating System Group (os.inf.tu-dresden.de)开发的 Real－Time 微内核，支持 L4 V2.0 和 L4 X.0 标准(L4 的接口标准)，Fiasco 是由 C＋＋实现的典型 L4 系统，Fiasco 提供众多 L4 系统调用以及 Fiasco 的实时扩展，请点击 Fiasco Syscalls。L4Env 是一套基于 Fiasco 的服务程序,包括 roottask, sigma0, log, names, dm_phys, l4vfs, l4io, dope, con 等各种 server，L4Env 是一种典型 SawMill Multi－Server OS （参考 paper **The SawMill multiserver approach）,关于 L4Env 的一些基本情况，请点击 L4Env Manual。L4Linux** 是基于 L4Env 移植的 Linux 系统，Linux－2.0, Linux-2.2, Linux-2.4, Linux-2.6 前后分别被移植到 L4Env 上面，目前 L4Linux 版本更新到 2.6.26，L4Linux 相当于一种基于“L4 CPU”的 Linux 系统，对 Linux 系统的修改都存放在 arch/l4 目录下面，较好地维持了 Linux 系统 semantic integrity。关于 Fiasco/L4Env/L4Linux 的设计，请参考 Paper **The Performance of µ-Kernel-Based Systems** ，这个 Paper 也是微内核领域最著名的 Paper 之一。

值得注意的是，基于 Fiasco 和 L4Linux，有 2 个很重要的研究成果，DROPS 实时系统是一种面向服务质量需求的实时系统，可以提供某种程度的保证(guarantee)。L4/Nizza，一种面向 Trusted Computing 的基于微内核的系统架构,这也是最早利用 L4 微内核进行 security system 研究的工作,可以参考 Paper：Security Architecture Revisited。此外，他们维护一个 IDL for L4，称为 Dice。

3. Pistachio/AfterBurner Pistachio 是目前最好的 L4 微内核之一，它由卡尔斯鲁尔大学系统体系结构研究组和新南威尔士州立大学操作系统研究小组共同开发的微内核。跟所有的研究机构一样，一开始大家都是单干，卡尔斯鲁尔作 Hazelnut，新南威尔士州立做 L4/MIPS,L4/Alpha。后来，大家联合起来，做成了 Pistachio。不过微内核之上的部分大家一直单干，各有各的系统。卡尔斯鲁尔的 Pistachio 小组在很长的时间内，一直使用来自 TUD 的 L4Linux 作为基于 Pistachio 的虚拟化技术，直到 Pistachio 的 Afterburner 技术出现为止才有了改观。AfterBurning 是该小组研发的一种 Pre-Virtulization 技术(Pre-Virtualization 是一种兼顾 Para-Virtulization 的高性能和 Modularity 的可维护性而出现的一种尝试，具体来说，是把一种 source code 可以根据需求编译出不同的系统，同样的 Linux，可以编译出适用于 Xen 的 Guest OS，也可以编译出使用 L4:Pistachio 的 Virtual Machine。由于这项工作是在编译阶段完成的，因此诸多优化也可以同时生效，而避免了 Para-Virtualization 的单一性。比如 L4Linux 只能应用用 Fiasco，XenLinux 只适合于 Xen 等等),因为是通过编译来完成的，所以性能会更好一些。比较有趣的是，他们有一个 BurnNT 技术，可以支持 Multi-Windows，网站上面提供源代码下载。他们关于 Device Driver Virtualization 有一篇 Paper 是 OSDI 的，Unmodified Device Driver Reuse ，是近几年 L4 领域一篇少有的佳作。其主要思想是把每一个 Virtual Linux 当作一个 Device Driver Server，从而提供 Dependable System.
4. OKL4/Iguana OKL4 是 L4:Pistachio-Embedded 的延续,它目前有 Open Kernel Labs 公司维护，但是研究工作基本上都是在 ERTOS 完成的。目前 OKL4 的市场化推广作的不错，已经有很多产品使用了 OKL4，包括基于 OKL4 的 OpenMoko，也已经面世。相当于 TU-D 和 Uni Karlsruhe 的 L4 小组，ERTOS 规模显得很庞大，他们网站上面的项目也很多，各种项目都有。主要有：1）基本系统维护， OKL4+ Iguana+ Magpie＋ Wombat, Iguana 类似于 L4Env, Magpie 类似于 Dice,Wombat 类似于 L4Linux，一一对应。2）Security seL4+L4.Verified，我分不清楚 2 个项目的目标有什么不同，seL4 是 security Embedded L4 的意思，总之，其核心内容即使使用 formal method 来验证 OKL4 is secure kernel，似乎他们现在已经达到验证机器码的程度，大概步骤就是使用 Haskell 重新实现 OKL4 的 API，然后使用 Isabela 进行证明，在这个方面，Kernel Verification，他们作的很成功，这也是他们可以赢得众多工业厂商青睐的一个原因 3）Real-Time 既然是面向 Embedded 的，Real-Time 自然是要用；Component-based Microkernel；Power Management，这些也都是目前 OS 研究的一些 Hotopic.
5. Coyotos 首先，Coyotos 不是 L4，但是 Coyotos 和 L4 之间的关系之密切远远胜过了其他微内核和 L4 之间的关系。比如 Fast IPC， Capability-Based OS, IDL。Coyotos 是 KeyKOS 和 EROS(Extremely Reliable OS)的改进版本，从 EROS 的名称或许可以看出，这个系统和以上的系统有些不同，它强调 reliable，所以 EROS 刚开始的时候，被应用于一些军用系统，但是后来发现 Synchronous IPC 会导致一个 Denial of Service 的 Bug，这个 Bug 存在于所有基于 Synchronous IPC 的系统中，当然也包括所有的 L4，在 Vulnerability In Synchronous IPC design 中有详细的描述。当然，现在这个 Bug 也已经被修正。Coyotos 的目标应该是提供具有军用级别的（EAL7 =Evaluation Assurance Level）的 microkernel，它使用一种新的称为 BitC(类似于 Haskell 的 Safety Language)来实现这个系统，而且整个系统采用一种类似于 OOP 的形式（所有的 L4 系统都是 OOP 的，Fiasco 和 Pistachio 都是 C++的）。Coyotos is still persistant and transactional microkernel OS。从概念上来讲，Coyotos 更为先进，Capability 也是在该系统上面首次被应用，所以有志于研发 3rd Microkernel 的不妨多多关注这个

Microkernel.

6. Mungi Mungi 是 ERTOS 小组开发的 persistant SASOS (single address space operating system)，这个项目已经停止。但是作为第一个利用 L4/Microkernel 来开发不同类型的 OS，仍然可以给我们提供很多新的 idea,尤其是 persistant，即使到目前为止，应该有值得研究的价值。http://www.coyotos.org/官网上面有很多该 Microkernel 的设计文档，值得一看。
7. M. D. Bennett 《A Kernel For IMA Systems》 这是 University of York 的以为 PH.D 基于 L4 尝试构建 IMA 系统,IMA 全程是 Integrated_Modular_Avionics，也就是面向航空航天控制的系统，这也是我能找到的基于 L4 所作的 safety critical system 的唯一一次尝试，因为我自己是倾向于作这个方向的，所以列举这个 Kernel 在这里。因为 University Of York 是 safety critical system 的大本营，所以这篇 paper 应该不错。
8. Genode OS Framework. 一个 recursive hierarchical constructive os framework. 目前基于 L4/Fiasco 开发，主页：genode.org

1.2.2 微内核操作系统分代

微内核操作系统及 L4 概述
http://wenku.baidu.com/view/90929762caaedd3383c4d311.html

微内核(microkernel)并非是一个新的概念，这个名词至少在七十年代初就有了。一般认为，他的发明权属于 Hansen 和 Wulf 。但是在这一名词出现之前已经有人使用类似的想法设计计算机操作系统了。

早期的操作系统绝大多数是 Monolithic Kernel，意思是整个操作系统 - 包括 Scheduling （调度），File system （文件系统），Networking （网络），Device driver （设备驱动程序），Memory management （存储管理），Paging（存储页面管理） - 都在内核中完成。一直到现在广泛应用的操作系统，如 UNIX、Linux 和 Windows 还大都是 monolithic kernel 操作系统。但随着操作系统变得越来越复杂（现代操作系统的内核有一两百万行 C 程序是很常见的事情），把所有这些功能都放在内核中使设计难度迅速增加。

微内核是一个与 Monolithic Kernel 相反的设计理念。它的目的是使内核缩到最小，把所有可能的功能模块移出内核。理想情况下，内核中仅留下 Address Space Support（地址空间支持）、IPC (Inter-Process Communication，进程间通讯)和 Scheduling（调度），其他功能模块做为用户进程运行。对于内核来说，他们和一般用户进程并无区别。它们与其他用户进程之间的通讯通过 IPC 进行。

在八十年代中期，微内核的概念开始变得非常热门。第一代微内核操作系统的代表作品是 Mach。Mach 是由位于匹兹堡的卡内基梅隆大学（CMU）设计。 CMU 是美国计算机科学研究重镇，其计算机排名长期位于美国大学前五位。美国只有少数几所大学的计算机是学院不是系，CMU 就是其中之一。除 Mach 外，

CMU 的另一重要成果是衡量计算机软件设计能力的 CMM (Capability Maturity Model)模型，广泛用于评估业界软件公司的计算机软件开发能力。好像印度的软 件公司们非常热衷于此，通过 CMM-5 最高规格评价的软件公司们有一半是印度的。

在微内核刚兴起时，学术界普遍认为其优点是显而易见的：

- 支持更加模块化的设计；
- 小的内核更易于更新与维护，bug 会更少。大家知道 Windows 的死机很多是因为 device drivers 造成的。如果把他们移出内核，他们中的 bugs 很可能就不会造成死机；
- 许多模块的 bugs 可被封闭在其模块内，更加易于 debug。软件工程师都知道 kernel debug 是件非常头疼的事情。如果 file system、memory management 和 device drivers 成为一个个独立的进程，不用说这肯定会使 kernel engineers

的日子好过许多。

由于上述原因，很多学术研究人员和软件厂家开始尝试使用微内核的概念设计操作系统。甚至 Microsoft 也有所动作，在设计 Windows NT 时，他们把 UI (User Interface) 从 Windows kernel 中整个拿了出来。由此可看出 microkernel 的流行程度，因为 Microsoft 总是最后一个尝试新想法的。但是这一热潮很快就冷了下来，原因只有一个字：Performance （Well ，汉语是两个字：性能）。Microsoft 在 Windows NT 后续版本中，又把部分底层 UI 放回了 Windows kernel。这种现象在计算机界并不少见。在 Java 刚开始流行时，由于其许多 C/C++没有的优点，很多人认为它会很快取代 C/C++成为第一编程语言。但直至今天也没有发生，其中一个重 要原因就是 Java 程序的 performance 落后于 C 程序，至今还限制着它在许多场合的 应用。

第一代的微内核操作系统的性能，包括 Mach 在内，远不及 monolithic kernel 操作 系统。所以大多数人又回到传统技术中去了。Microkernel 也像过时的流行歌曲或 减肥方法，很快被遗忘了。

但在九十年代后期，微内核迎来了其生命中的**第二春**。一些研究人员认真分析了微内核系统性能差的原因，指出其性能差并非根本内在的因素造成，而是设计实现的失误。为证明其论点，他们设计并实现了几个性能远超第一代的微内核操作系统， 我们把它们称为第二代微内核系统。其中的一个代表作品就是 L4。

L4 由德国的 GMD 设计。GMD 是德国国家信息技术研究院，相当于中科院计算所加上软件所（但在计算机研究能力方面，GMD 远非中科院可比）。L4 的创始人和总设计师是 Jochen Liedtke。此公在 GMD 之后，还供职于 IBM 的 T.J. Watson Research Center，后成为德国 Univ. of Karlsruhe 操作系统方向的正教授（full professor）。

了解德国大学系统的人都知道，当德国的正教授可比当美国的正教授要难许多倍， 地位也要高许多，因为一个系往往只有一两个正教授。Prof. Liedtke 已于 2001 年过 世，但他创建的 L4 正在发展壮大中。近年来，多个运行于不同硬件平台的 L4 系 统已被几个不同研究机构设计出来。

上面的说法实际上是给半懂操作系统的人看的，因为操作系统内核完成的功能，远不只是调度、内存管理那么简单。在宏内核中，内核是一个朱元璋那样的一个独裁者，他独裁，你的一切行为他都可能干预，同时他也真的是有能力、有精力（据说朱元璋曾经一天批阅 400 道奏折），你程序退出时的屁股是否擦得干净之类，他都替你做。可微内核操作系统，谁给你干这些？于是为了应用程序的模型，你不得在在用户态又搞一个象宏内核中的内核一样的一个服务机构，这个机构如果挂了，你的系统也挂了。当然，这时因为内核还活着，你还有起死回生的机会。

再举个例子，你的程序正在等待别的程序的一个锁，这时，你 exit()了，你的程序留下的那些信息谁替你清理？内核啊。当你的程序 exit()后，要把整个计算环境清理成就象这个世界上从来没有出现过你的程序，这都是很费力气的工作。在传统的 Linux 这样的计算模型下，这些工作又是必须的。

2 内核服务与内核对象

内核服务与对象，Kernel Services and Objects，微内核操作系统，内核只提供有限的原始系统服务，这些服务，对于一个习惯于 Linux 这样的成熟 PC 操作系统的程序员员来说，显得很不完整，但是微内核的设计者，不希望内核完成更多的工作，内核是跑在保护模式（privileged mode）的，这里面的程序尽量的少，会使得系统处于活着（哪怕只有内核是活着，因为只要内核活着，就象人有一口气在，你总会有办法让系统恢复过来）容易些。

seL4提供的基础服务（basic services）如下：

- 线程（Threads）是CPU执行的最小单位，是个抽象的概念；
- 地址空间（Address spaces）是虚拟内存空间，每个虚拟空间中，可以容纳一个程序，这个程序只能访问这个虚拟地址空间限定了的内存；
- 进程间通信进程间通信（Inter-process communication，IPC)允许线程通过端点（endpoint）互相之间发送消息；
- 设备基础单元（Device primitives）允许在用户态实现硬件设备驱动，内核把实现驱动所需要的中断通过IPC形式提供；
- 句柄空间（Capability spaces）存储了与内核对象对应的句柄（capability）。

本章将概要描述seL4内核提供的这些服务，描述用户程序（工作在用户态）如何访问这些内核对象及这些对象如何被创建。

2.1 基于句柄的访问控制

seL4 提供了基于句柄的访问控制（Capability-based Access Control）机制，访问控制掌控着所有对内核的访问。用户态程序，必须有足够的权限，通过访问它自己拥有的句柄来发出请求（invoke)。通过这个机制，系统使得软件构件间得以隔离，并且允许审计、控制构件间的通信，手段就是有选择地授权特定的通信句柄。这个机制使得软件构件在更高层面上得以检查，因为只有被明确授权过的操作才能以被调用。

一个句柄是一个不可伪造的令牌,它唯一地标识了一个内核对象(如线程控制块 TCB)，句柄的管理机构中，有这个句柄的访问权限信息，这些信息中包含哪些方法可以被访问。概念上，句柄存储于应用程序的特定存储区（capability space)，这个存储区的存储单元你为 slot，一个 slot 可以包含一个句柄，也可以啥也不包含。应用程序通过句柄（即存有这个句柄的那个 slot 的地址）来请求内核中的服务。这意味着 seL4 的 capability 模型是一个被内核管理的隔离的（segregated）或分区的（partitioned) capability 模型，也就是说这些句柄的内存虽然是存储在用户态中的，但是被内核管理。

句柄空间（Capability space）被实现为内核管理的capability nodes的有向图（CNodes)。一个CNode是一个槽（Slot）表，每个Slot可以包含一个CNode句柄。句柄空间（Capability space）中的一个地址串联起句柄、指向目标槽（Slot）的路径（Path）等。

句柄在Capability space中可以被复制与移动，或者通过IPC发送出去。这将允许创建具有特定访问权限的应用、与其它应用打交道的代理，或把与某个程序打交道的权限传给刚创建的（或选定的）内核服务。此外，句柄可以通过铸造（Mint）操作，获得包含原来的句柄的一个权限子集（没有更多的权利）的新句柄。

句柄上的权限也可以被撤销，递归地撤销所有从原句柄中继承来的权限。句柄的繁殖控制模型被称为take-grant-based模型。

2.2 系统调用

seL4内核提供了线程间的消息(Message)传送服务。消息由一些消息字(data words)及可能的一些句柄组成。

线程是seL4程序的基本执行单位，所以这里做事情的主体都用线程来指代。

线程通过调用它们句柄空间（capability space）中的句柄，向该句柄所对应的计算发送消息。当一个端点（endpoint）句柄以这种方式被调用，即向这个句柄发送消息，内核将把消息的内容发送到另外一个线程中。当指向内核对象的句柄被调用，这个消息将被解释为内核对象的方法调用。例如，用合适的消息调用线程控制块TCB（Thread Control Block）的句柄，将引起目标线程的挂起。

内核提供有下列系统调用：

- seL4_Send() 阻塞式消息发送
  向某个有名句柄发送一个消息，然后让程序继续执行。如果被调用的句柄是 个端点，而且没有消息接收者正在等待，发送消息的线程将阻塞，直到消息可以投递，即有消息接收者接收了这个消息。接收消引息的线程或内核对象，无法把错误代码或其它应答返回给调用者。

- seL4_NBSend() 非阻塞式消息发送
  与seL4_Send()类似，除了如果消息不能有接收者立即被接收，这个消息将被丢弃。象seL4_Send一样，接收消引息的线程或内核对象，无法把错误代码或其它应答返回给调用者。

- seL4_Call() 阻塞式调用
  它实质上是一个seL4_Send()调用，发出请求的线程将被阻塞直到消息被接收。当消息被发送给接收者（通过端点Endpoint），一个附加的应答句柄被同时发送，这使得消息接收者可以有权发应答信息。应答句柄存储于接收线程的TCB的特殊槽（slot）中。当通过seL4_Call()调用内核对象句柄时，内核对象在返回消息中返回错误码或其它应答数据。

- seL4_Wait() 阻塞式接收
  用于线程阻塞式通过端点（endpoint）接收消息。如果当前没有消息可接收，没有线程Send消息，本线程将阻塞，直到有消息可以接收。这个系统调用的对象只能是端点（Endpoint），如果在其它类型的句柄上调用seL_Wait()，将引起一个异常。

- seL4_Reply() 应答阻塞式调用
  用来应答seL4_Call()，用seL4_Call()生成并存储于TCB中的句柄，向调用者发送消息，唤醒它的线程。
  TCB中只有一个用作应答的句柄，所以seL4_Reply只能用来应答最近的消息。
  seL4_CNode_SaveCaller()可以用来在句柄空间中保存应答句柄，然后通过seL4_Send向这个句柄发送消息向调用者传送消息。

- seL4_ReplyWait() 应答并等待
  它是一个seL4_Reply()与seL4_Wait()的复合体，是为了效率的原因而设置的。应答然后马上进入等待状态，从而在一个系统调用中完成多件事情。

- seL4_Yield() 放弃CPU权利
  是唯一不需要句柄指引的系统调用，它的功能是让当前线程把它的时间片放弃，让同优先级的其它线程先执行。如果没有同优先级的线程可以执行，当前线程将继续，就当什么也没有发生过。

2.3 内核对象

操作系统内核提供的功能，很多并不是内核本身提供的，而是跑在内核态的程序提供的，象Linux这样的操作系统的内核，这类内核程序可能是驱动，或以伪驱动形式存在的内核程序。

seL4中的内核对象是内核向用户程序提供的一组功能集，应用程序眼中的这些内核对象，与内核眼中的这些对象，在接口上是一样的，就是说，服务态程序中看到的这些对象，并没被单独包装过，只是需要通过句柄才能标识这些对象。

- CNodes
  存储句柄（capabilities），以使线程可以访问特定对象中的方法。每个CNode有固定数量的槽（Slot），槽的数量于CNode创建时确定，槽中可以有一个句柄，也可以为空。

- 线程控制块 Thread Control Blocks （TCBs）
  线程是seL4中执行与调度的基本单位，提供有阻塞、非阻塞等等功能。

- IPC端点 IPC Endpoints
  实现线程间的通信，seL4内核提供有下列两种端点：
  - 同步端点 Synchronous endpoints (Endpoint), which cause the sending thread to block until its message is received; and
  - 异步端点 Asynchronous endpoints (AsyncEP), which only allow short messages to be sent, but do not cause the sender to block.

  一个指向端点的句柄可以被限定为只发送（send-only）或只接收（receive-only），并可以被设置为可以在线程间传递。

- 虚拟地址空间对象 Virtual Address Space Objects
  用来创建虚拟地址空间（或VSpace），这些虚拟地址空间可以给一个线程或多个线程使用。这些对象管理着物理的存储设备，例如，页字典（page directory）管理着页表（page directory），页表就是让你的虚拟地址与物理地址对应起来的那个物理器件MMU，内核还包括ASID Pool和ASID Control对象，用来跟踪地址空间状态。

- 中断对象 Interrupt Objects
  give applications the ability to receive and ac-knowledge interrupts from hardware devices. Initially, there is a capability to IRQControl, which allows for the creation of IRQHandler capabilities. An IRQHandler capability permits the management of a speci c interrupt source associated with a speci c device. It is delegated to a device driver to access an interrupt source. The IRQHandler object allows threads to wait for and acknowledge individual interrupts.

- 原始内存 Untyped Memory
  is the foundation of memory allocation in the seL4 kernel. Untyped memory capabilities have a single method which allows the creation of new kernel objects. If the method succeeds, the calling thread gains access to capabilities to the newly-created objects. Additionally, untyped memory objects can be divided into a group of smaller untyped memory objects allowing delegation of part (or all) of the system's memory. We discuss memory management in general in the following sections.

```
typedef enum api_object {
    seL4_UntypedObject,
    seL4_TCBObject,
    seL4_EndpointObject,
    seL4_AsyncEndpointObject,
    seL4_CapTableObject,
    seL4_NonArchObjectTypeCount,
} seL4_ObjectType;
```

2.4 内核内存申请

seL4不为内核对象动态分配内存，内核对象必须在用户程序控制的内存区中通过Untyped Memory句柄创建。这个机制可以显式地控制应用程序可以使用的内存数量，提供应用程序间的内存隔离。一个应用程序可以使用多少物理内存，seL4没有绝对限制。

在系统起动（boot）的时候，seL4先预申请一块内存给内存用，包括代码（code）、数据（data）、栈等段（section）。seL4是一个单内核栈操作系统。余下的内存留给初始化线程，说是留，也就是把指向原始内存（Untyped Memory）的句柄传给初始线程，初始线程再把原始内存分解成一些小的部分，或者通过seL4_Untyped_Retype()创建内核对象。

初始线程起动时，还有一些其它的内核句柄传给它。

新创建的原始内存对象都是最初的这个原始内存对象的子对象。

用户态程序利用seL4_Untyped_Retype()创建对象，得到的是句柄，通过这个句柄进行后续操作。

The user-level application that creates an object using seL4 Untyped Retype() re-ceives full authority over the resulting object. It can then delegate all or part of the authority it possesses over this object to one or more of its clients.

2.4.1 内存重用

seL4这种把虚拟地址管理、虚实映射管理、物理内存（原始内存）管理等分开，每一部分都允许应用程序自己决定如何使用的内存管理策略，对于应用程序申请内核对象，在客户程序间分享授权（authority）信息，获得这些对象代表的内核提供的各种服务则够用了的。对单一的静态（static）小型应用是够用了。

se4内核也提供原始内存（Untyped Memory）重用机制。重用一段内存区域（region）是允许的，只有当那段原始内存上没有摇摆的引用（references，例如句柄capability），也就是没有别的句柄指向这段内存里面的空间，seL4内核跟踪对象的引用、继承关系（capability derivations），例如，调用下列方法得到的子对象：seL4_Untyped_Retype()、seL4_CNode_Mint()、seL4_CNode_Copy()、seL4_CNode_Mutate()等。

seL4内核对象被记录在句柄继承树（capability derivation tree，CDT）。举例来说，当一个用户在原始内存上创建了一个内核对象，这个对象句柄将要被插入CDT树，成为原始内存句柄的一个子对象。这个原始内存句柄就是系统系统时，内核传给应用程序的那个原始内存句柄。

虽然CDT理论上是内核数据结构，但是它被实现为CNode的一部分，不需要客外的内核数据存储。

每个原始内存区（Untyped Memory region），内核都保留一个水印（watermark），用来记录在这个内存区之前，已经被申请了多少内存区。任何时候，当用记申请内核在原始内存区创建一个对象，内核都要做两件事：

- 如果在这个内存区中已经有对象，内核在同一水印级别（level）申请新对象，并提高水印值。
- 如果在这个内存区中申请的对象都被删除了，内核将重置水印，并在内存区的开始重新申请内存。

最后，seL4_CNode_Revoke()删除从指定句柄继承出来的所有句柄。当删除指向一个内核对象的最后一个句柄时，内核将触发针对无引用对象的删除机制，删除机制将清除它的继承关系。这其实就是一个引用记数方式管理的对象，当引用记数变成0时，引起对象的析构机制，把对象清除干净。

当在指向原始内存对象（original capability to an untyped memory object）身上调用seL4_CNode_Revoke()，用户将删除所有原始内存对象的子对象，即所有指向这个原始内存区的句柄。

句柄只是一个象c++ io库打开文件函数返回的那个整数。

int open(const char * pathname, int flags);

关闭文件，是要释放这个整数代表的那些资源，反观之，文件都关了，你拿一个代表这个“文件”的整数就没有意义了。

当一段内存区上的对象都释放干净了，它就回归为原始内存区了，交回给物理内存管理机制了，被重新使用。

3 句柄空间

句柄空间（Capability Spaces，CSpace），就是存放句柄的内存空间。

seL4实现了一种基于句柄（capability-based）访问控制模型。每个用户空间线程都与一个句柄空间关联，其中存储了这个线程可以访问的句柄，因此控制着这个这个线程可以访问的资源。

访问句柄代表的内核对象，这类对象被称为CNode。一个CNode是一张由记录槽（Slot）组成的表，可能记录着指向其它CNode的句柄，形式是有向图。概念上，一个线程的CSpace是一个通过CSpace根可以遍历得到的有向图的一部分。

一个CSpace地址指向一个独立的槽slot（CSpace中的CNode），一个槽slot可以包含一个句柄，当然也可能是空的，即不包含一个句柄。

句柄在CSpace中可以移动。也可以通过消息发送，而且句柄可以被锻造（Mint）为权限只是它的子集的新的句柄。seL4管理了一个句柄继承树（capability derivation tree，CDT），CDT中跟踪记录了句柄间的继承关系，是从别的句柄复制过来的，还是原始的。revoke方法移除派生自某个句柄的（在CSpace中的）所有句柄。这个机制可以用来复原传给客户端的句柄，因为传给客户端的句柄往往是从服务器端的某个句柄加工而得到的。原始内存管理机构也可能通过这种方式删除某个内存区的所有对象，从而让其回归原始内存（untyped memory）的本质。

seL4要求应用程序管理所有在内核中的数据结构，因为seL4内核的这些数据结构，实际上是在用户态被用户程序创建的，这类数据结构包括CSpace。这就意味着用户程序主动通过传给它的地址句柄构造CSpace。

本章后面将要描述句柄在程序中的表示及在程序中如何引用CSpace中的独立句柄。

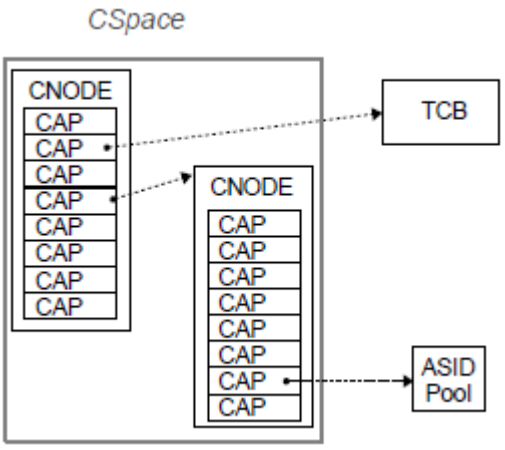

图3.1 CSpace与CNode关系示意图

句柄空间是内核数据结构的一部分，只是因为微内核操作系统的设计理念是系统一经稳定运行，内核就不再申请内存，所以这块内存被放在一块内核可以访问得到的地方，如初始进程（内核创建的第一个进程，象Linux的init0进程）内的一段内存。

```
/* allocator static pool */
#define ALLOCATOR_STATIC_POOL_SIZE ((1 << seL4_PageBits) * 10)
static char allocator_mem_pool[ALLOCATOR_STATIC_POOL_SIZE];

/* initialise allocator */
allocman_t *allocator = bootstrap_use_current_1level(init_data->root_cnode,
```

```
        init_data->cspace_size_bits, init_data->free_slots.start,
        init_data->free_slots.end, ALLOCATOR_STATIC_POOL_SIZE,
        allocator_mem_pool);
```

所以，句柄空间CSpace，并不象虚拟地址空间VSpace之类的资源一样，每个进程Process自己管理一份，而只是在系统初始化时，由初始进程创建一份，然后把这个CSpace信息传递给新创建的进程，从道理上来讲，所有进程都是初始进程的子进程。

```
init_data = receive_init_data(endpoint);
env.cspace_root = init_data->root_cnode;
```

3.1 句柄与 CSpace 管理

句柄与 CSpace 管理，Capability and CSpace Management 。

3.1.1 CSpace 创建

CSpaces通过创建与操作CNode对象而创建。当创建CSpace时，用户必须先指定其中将要存储的Slot个数，这也决定了这个CSpace将要占用的内存数量。每个槽Slot占用16字节，每个槽刚好存储一个句柄。刚其它内核对象一样，CNode也必须通过调用seL4_Untyped_Retype()创建。创建CNode将占有一定的原始内存，调用者自己有多少原始内存可用，是否容得下要创建的对象，显然要清楚，否则出错那是肯定的。§9.3.27里面对每种对象的尺寸有描述。

3.1.2 CNode 方法（CNode Methods）

句柄（Capability）主要通过下列CNode方法管理。

CNodes support the following methods:

- seL4_CNode_Mint()
  基于一个已经存在的句柄在一个槽中创建一个新的句柄，这个新创建的句柄的权限是原句柄的子集（§3.3.1）。seL4_CNode_Mint()函数也能从一个未标记句柄（unbadge capability）创建一个标记句柄（badged capability）。
  当IPC时，同步调用时的endpoint就可能是一个标记句柄，你要把句柄传给别人，没个标记，你让人家咋认出它？见§4.2.1。

- seL4_CNode_Copy()
  功能类似于seL4_CNode_Mint()，拷贝创建一个新句柄，新句柄中的权限与原句柄完全相同。

- seL4_CNode_Move()

moves a capability between two speci ed capability slots. You cannot move a capability to the slot in which it is currently.

- seL4_CNode_Mutate()
can move a capability similarly to seL4 CNode Move() and also reduce its rights similarly to seL4 CNode Mint(), although without an original copy remaining.

- seL4 CNode Rotate()
moves two capabilities between three speci ed capability slots. It is essentially two seL4 CNode Move() invocations: one from the second spec-i ed slot to the rst, and one from the third to the second. The rst and third speci ed slots may be the same, in which case the capability in it is swapped with the capability in the second slot. The method is atomic; either both or neither capabilities are moved.

- seL4 CNode Delete()
removes a capability from the speci ed slot.

- seL4 CNode Revoke()
类似于seL4_CNode_Delete()，删除从一个句柄派生出去的所有句柄，对句柄本身没有影响除非§3.2中描述的情形。

- seL4 CNode SaveCaller()
把一个内核创建的当前线程的应答句柄到指定槽中。
IPC时，被调用方可以发一个应答消息，这个应答消息的句柄是保存在当前线程的控制块（TCB）中的，而一个线程只能保存一个这样的句柄，如果想把一个请求应答记录下来怎么办，通过这个函数把句柄记下来，以备以后向其发送消息。

- seL4 CNode Recycle()
类似于seL4_CNode_Revoke()，除了把对象一些状态复位为其初始状态。

3.1.3 句柄权限（Capability Rights）

句柄权限（Capability Rights）这个词很容易让人迷糊，它实际上包含了两层意思，一层是你对句柄本身能干点啥，seL4_CNode_Mint()、seL4_CNode_Mutate()之类的操作；再一层是你能对这个句柄代表的对象做点啥。

- **写权限** 允许你向句柄发送数据，例如一个端点endpoint句柄一定要有这个权限，用来发消息。
- **读权限** 允许你从对角上接收或读数据
- **锻造（Mint）权限** 允许你在同一对象上创建不同的句柄，两个句柄指向同一对象，但是具有不同的功能，例如，用不同的标记（badge）创建端点（endpoint）句柄。也允许通过Identify操作从其它句柄上读取标记到同一对象上。

有些句柄有关联在它上面的操作权限。一般情况下，权限包括几种类型：端点Endpoints、AsyncEPs（见§4）和页Pages（见§6）。一个句柄，其上面关联有什么权限，意味着可以对它做哪些调用。seL4支持三种权限的正交组合：读Read、写Write、授予Grant。与各种对象相关的各种权限的意思见表3.1。

当一个对象被创建后，它的初始权限是最大的，而基于它通过seL4_CNode_Mint()、seL4_CNode_Mutate()之类函数创建的权限可能会越来越少。如果一个更大的权限被设置到句柄上，seL4将把它降低到原句柄的权限。

表3.1 seL4句柄访问权限

Type	Read	Write	Grant
Endpoint	请求等待 Required to wait.	请求发送 Required to send.	发送句柄（包括应答句柄） Required to send capabilities (including reply capabilities).
AsyncEP	Required to wait.	Required to send.	N/A
Page	Required to map the page readable.	Required to map the page writable.	N/A

3.1.4 句柄继承树（Capability Derivation Tree）

很多方法，如：seL4_CNode_Copy()或seL4_CNode_Mint()，都涉及到句柄派生（derive），不是所有的句柄都支持派生操作。概括来说，只有原始的（original）句柄支持派生调用。但也有特例。表3.2概括了句柄的可派生性，及不可派生出错信息。未列出的句柄类型可以被派生一次。

表 3.2 句柄派生

句柄类型 Cap Type	派生条件 Conditions for Derivation	错误码 Error Code on Derivation
ReplyCap	Cannot be derived	Dependent on syscall
IRQControl	Cannot be derived	Dependent on syscall
Untyped	Must not have children (Section 3.2	seL4_RevokeFirst
Page Table	Must be mapped	seL4_IllegalOperation
Page Directory	Must be mapped	seL4_IllegalOperation
IO Page Table（只适用于IA-32）	Must be mapped	seL4_IllegalOperation

图3.1展示了一个示例句柄派生树，最上层是一个大的untyped句柄，第二层分裂成两个，第三层左边是两个untyped句柄，Untyped句柄当复制时，永远会创建子句柄，而不是产生兄弟句柄。本例中，Untyped句柄被赋形（type）成两个独立的对象，第四层创建了两个句柄，都是原句柄的能力，都是派生句柄的子句柄。

赋形（type）操作，有些数据是没有类型的，如原始内存，你把它当作某种类型处理，相当于是赋予它某种数据类型了，这种操作就叫赋形。类似于c++里的强制数据类型转换：

```
void *b = malloc( sizeof(TypeA) );
TypeA *a = (TypeA *)b;
```

指针a和b指向的是同一个位置，但b是没有类型的，void的，而a是有类型的。

原始句柄可以有一个句柄，基于这个派生句柄的派生句柄将创建独立的兄弟句柄。

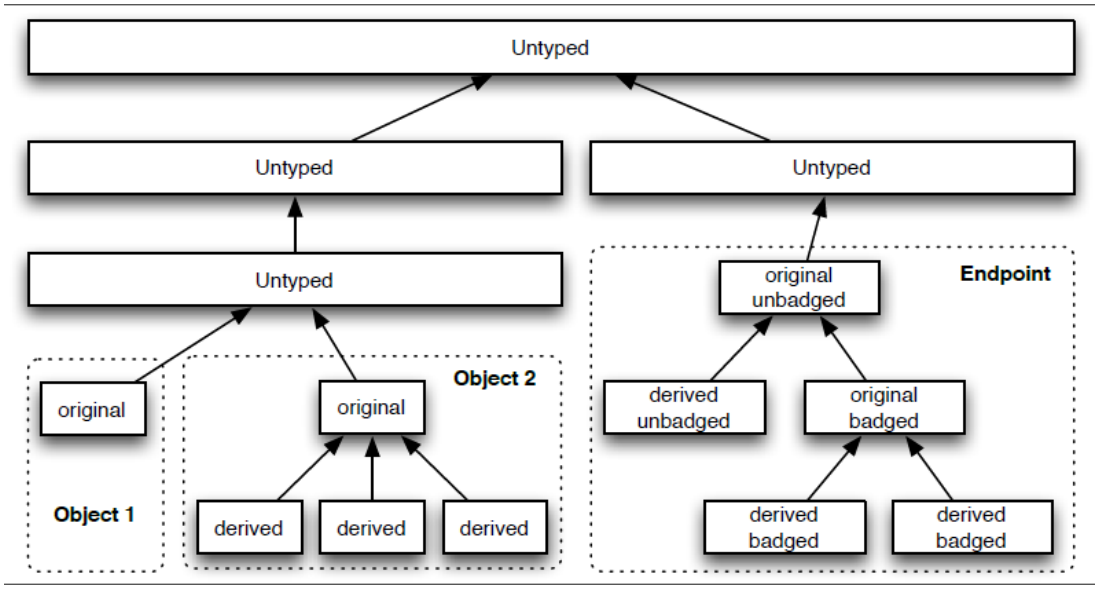

Figure 3.1: 句柄继承树示例.

3.2 删除、撤销、回收（Deletion, Revocation, and Recycling）

Capabilities in seL4 can be deleted and revoked. Both methods primarily a ect capa-bilities, but they can have side e ects on objects in the system where the deletion or revocation results in the destruction of the last capability to an object.

As described above, seL4 CNode Delete() will remove a capability from the speci ed CNode slot. Usually, this is all that happens. If, however, it was the last typed capability to an object, this object will now be destroyed by the kernel, cleaning up all remaining in-kernel references and preparing the memory for re-use.

If the object to be destroyed was a capability container, i.e. a TCB or CNode, the destruction process will delete each capability held in the container, prior to destroying the container. This may result in the destruction of further objects if the contained capabilities are the last capabilities. 1

The seL4 CNode Revoke() method will seL4 CNode Delete() all CDT children of the speci ed capability, but will leave the capability itself intact. If any of the revoked child capabilities were the last capabilities to an object, the appropriate destroy operation is triggered.

Note: seL4 CNode Revoke() may only partially complete in two speci c circumstances. The rst being where a CNode containing the last capability to the TCB of the thread performing the revoke (or the last capability to the TCB itself) is deleted as a result of the revoke. In this case the thread performing the revoke is destroyed during the revoke and the revoke does not complete. The second circumstance is where the storage containing the capability that is the target of the revoke is deleted as a result of the revoke. In this case, the authority to perform the revoke is removed during the operation and the operation stops part way through. Both these scenarios can be and should be avoided at user-level by construction.

The _seL4 CNode Recycle() method can be used to partially reset an object without fully removing all capabilities to it. Invoking it will rst revoke all child capabilities, but it will not remove siblings or parents. Only if, after revocation, the capability is the last typed capability to the object, the same destroy operation as in seL4 - CNode Delete() will be executed. Otherwise, not all aspects of the object will be reset: for badged endpoint capabilities, only IPC with this badge will be cancelled in the endpoint, for TCBs the capabilities will be reset, for CNodes, the guard on the capability will be reset.

Note that for page tables and page directories, neither _ seL4 CNode Revoke() nor _seL4 CNode Recycle() will revoke frame capabilities mapped into the address space. They will only be unmapped from the space.

3.3 CSpace 编址（CSpace Addressing）

当进行一个系统调用时，线程传给内核一个句柄，这个句柄在线程自己的句柄空间CSpace中，句柄存储在调用者句柄空间的特定的槽slot里。

CSpace设计上就禁止乱七八糟的访问，所以它被实现为受监控的页表（guarded page tables）中。CSpace需要特别高的访问效率，这里的访问是指：查找、读、写、编址等操作。

CSpace是一个CNode的树状数据结构，每个CNode是一个由槽slot组成的表，每个槽中存一个句柄或指向其它CNode的句柄。内核在每个线程的线程控制块TCB中保存了代表CSpace的根CNode。从概念上来说，一个CNode句柄不只存了对CNode的引用，还包括两个值：守卫guard和基数radix。

cspacepath_t数据结构

seL4编程中，表示一个句柄的数据结构是cspacepath_t，这个数据结构中包含了CSpace中槽的所有信息。

```
/* cspacepath_t
 * ------------
 *
 * This struct contains all the data to describe a slot in the CSpace.
 * It can be also used to describe a range of slots beginning at 'capPtr'
 * and count 'window'.
```

```
*
 capPtr: The address/pointer to a capability.
* capDepth: The depth of the address. (Most used value 32-Bit.)
* root:   Capability pointer to the root CNode in our CSpace.
* window: Count of caps in the range, beginning with 'capPtr'.
*
* !! Now this is where things are getting complicated. !!
* dest:   Capability pointer to the destination CNode. Resolved relative
*         to the root parameter.
* depth:  Number of bits of dest to translate when addressing the
*         destination CNode.
* offset: Number of slots into the node at which capabilities start being
*         placed
* ------------------------------
* ??? But what does this mean ???
* ------------------------------
* Example: Let's assume a three level cspace layout
*          ROOT [  |A|              ] 20 = bitlength
*                   |
*                   v
*          A        [  |B|          ]  8 = bitlength
*                      |
*                      v
*          B            [  |x|      ]  4 = bitlength
*
*          (where bitlength = guardsize + radix)
*
* Let's say we want to address the capability x which lies in the CNode B.
* That means our destination CNode is B.
*
* Imagine capPtr x as a concatenated value of offsets. In other words
* take the offset value of every level and put them together starting
* at the root.
*
* capPtr x  [rrrr rrrr rrrr rrrr rrrr|aaaa aaaa|bbbb]
*           |<-------------------->|<------->|<-->|
* bitlength            20                8       4
*
*
* In this case to address the CNode B we would get:
*
* dest:     [rrrr rrrr rrrr rrrr rrrr|aaaa aaaa]
*           |<------------------------------->|
* depth:               20       +        8       = 28
*
* To address the slot where x is, we specify the offset as:
*
* offset:   [bbbb]
*
*/
```

```
typedef struct _cspacepath_t {
    seL4_CPtr   capPtr;
    seL4_Word   capDepth;
    seL4_CNode  root;
    seL4_Word   dest;
    seL4_Word   destDepth;
    seL4_Word   offset;
    seL4_Word   window;
} cspacepath_t;
```

3.3.1 句柄地址查找 Capability Address Lookup）

象虚拟地址一样，一个句柄地址只是一个简单的整数，而不是指向哪个物理内存的地址(指针),想想虚拟地址的情形,也是一样。一个句柄指向一个句柄槽(capability slot)。当用户态程序试图查找一个句柄时，内核首先从线程控制块（TCB）中拿到CSpace的根，线程控制块TCB中存有对应CSpace的句柄CNode。内核然后检查你要查找的句柄的声明比特（significant bits），看它与CNode的守卫（guard）值是否匹配。

在c++中，假如我们在堆中申请内存，malloc()，得到的内存是8字节对齐的，那么返回的地址空间的后三位一定为0，如果不为0，就一定不是个合法的地址指针。这里的句柄空间CSpace也是这样，它是一棵树，象Linux的i结点一样存储，在其中找数据（以槽Slot为单位）时，每层结点都有一个守卫值，如果你的句柄存在了当前结点中，那么你的句柄整数的某些位应该与这个守卫值相同。这样做是在一定程度上防止应用程序伪造句柄。

如果你的句柄与本层的守卫匹配，拿出你的索引值（radix），从你的句柄里拿，那里的一些位就是这个偏移植，从那个偏移里拿出来的如果是个Slot，并且你的句柄的其余部分为0，那就算是找到了；如果从那个偏移里拿出来的是个CNode句柄，那就到那个CNode里去找，你的句柄剩余的部分，就是在那个CNode中的索引值。

Like a virtual memory address, a capability address is simply an integer. Rather than referring to a location of physical memory (as does a virtual memory address), a capability address refers to a capability slot. When looking up a capability address presented by a userspace thread, the kernel rst consults the CNode capability in the thread's TCB that de nes the root of the thread's CSpace. It then compares that CNode's guard value against the most signi cant bits of the capability address. If the two values are di erent, lookup fails. Otherwise, the kernel then uses the next most-signi cant radix bits of the capability address as an index into the CNode to which the CNode capability refers. The slot s identi ed by these next radix bits might contain another CNode capability or contain something else (including nothing). If s contains a CNode capability c and there are remaining bits (following the radix bits) in the capability address that have yet to be translated, the lookup process repeats, starting from the CNode capability c and using these remaining bits of the capability address. Otherwise, the lookup process terminates successfully; the capability address in question refers to the capability slot s.

图3.2展示了具有下列属性的一个CSpace：

- a top level CNode object with a 12-bit guard set to 0x000 and 256 slots;

a top level CNode with direct object references;

- a top level CNode with two second-level CNode references; second level CNodes with di erent guards and slot counts;
- a second level CNode that contains a reference to a top level CNode;
- a second level CNode that contains a reference to another CNode where there are some bits remaining to be translated;
- a second level CNode that contains a reference to another CNode where there are no bits remaining to be translated; and
- object references in the second level CNodes.

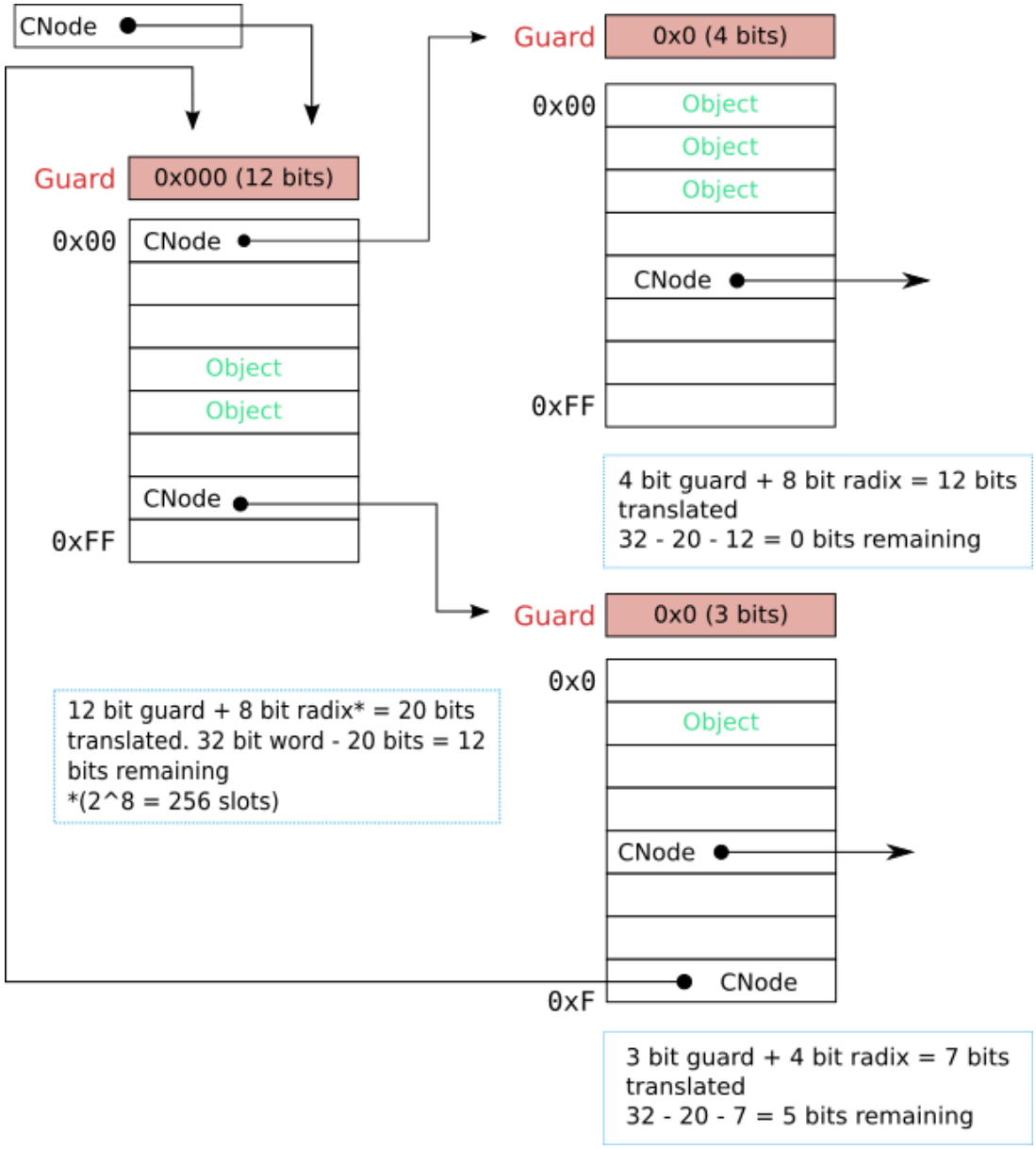

图3.2 CSpace示例图，各层次（level）的对象引用，各种守卫（guard）、索引尺寸（radix sizes）和CNode内部的相互引用

需要注意的是，图3.2展示的只是一种可能的场景，虽然其中的CSpace是合法的，但是它是不易工作的，原因就是每个结点的槽的数量太少，而且还有循环引用。

3.3.2 句柄编址（Addressing Capabilities）

一个句柄地址被存在于一个CPointer（缩写为）CPTR）中，它是一个无符号的整数。存储句柄的这个整数的定义上面有一些描述。两个特殊的情形是CNode句柄自身的存储和一段句柄槽（a range of capability slots）的存信储。

上面提到的CNode句柄转换算法，有地址位数的问题，所以在具体使用CNode句柄时，用户不只要提供一个句柄地址，还要提供句柄地址最大位数，这个位数被称为“深度限制”（depth limit）。

某些方法，如seL4_Untyped_Retype()，要求用户提供一组句柄空间（a range of capability slots），提供的方法是提供一个句柄地址的基地址（base capability address），基地址是句柄空间第一个句柄的地址，然后是窗口尺寸，就是紧跟基地址槽的连接槽的数量。

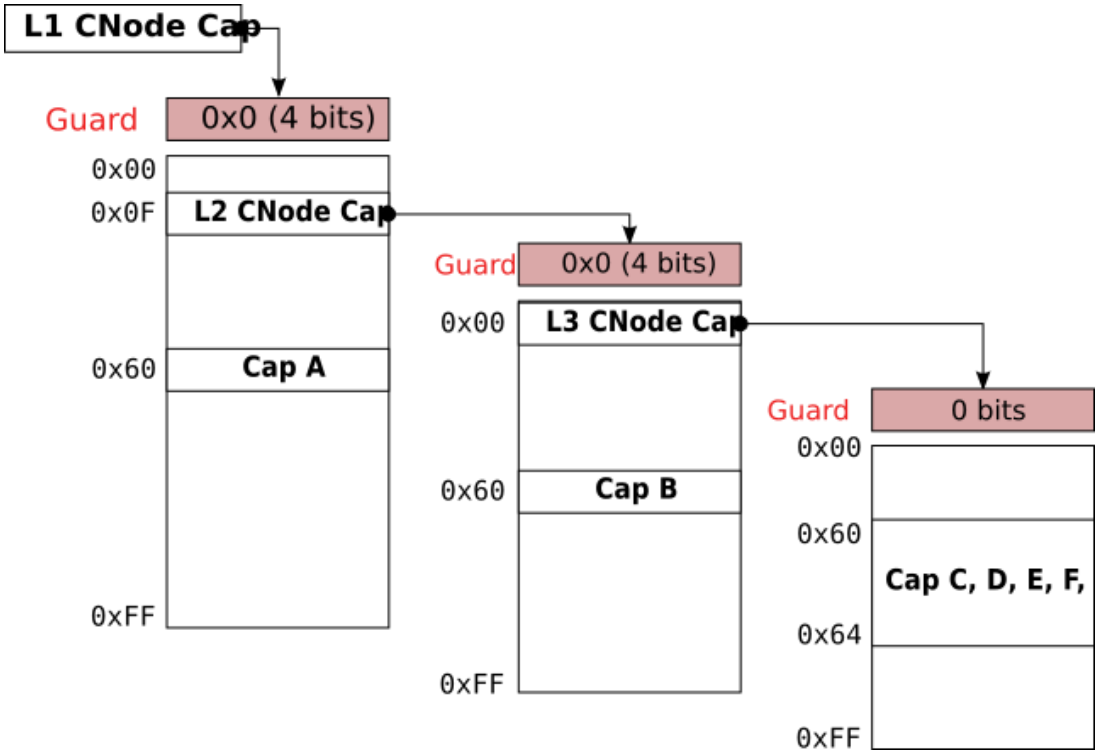

图3.3 任意CSpace拓扑图

Cap A. 第一个CNode有4比特的守卫（4-bit guard），被设为0，有8比特的索引（8-bit index）。A句柄存在于槽0x60（偏移0x60的那个地方的Slot）处，所以指向这里的句柄可以是类似这样的无符号整数：0x060xxxxx，这里xxxxx可以是任意数字，因为句柄查找算法只处理前12比特。简单来说，我们认可这样的无符号整数作为句柄：0x06000000。

Cap B. 再来，第一个CNode有4比特的守卫被设为0，还有8比特的索引（radix）。第二个CNode通过L2 CNode句柄而来，它也有4比特被设为0x0的守卫，Cap B存在于偏移0x60的槽中。所以Cap B的地址是0x00F06000，转换工作终止于前24比特。0x00F06000（分成三个部分：0x0 0F 060 00）中的0x0是4比特的守卫，0x0F是L2 CNode句柄的偏移，0x060是向下级CNode找的地址。

Cap C. 这个句柄经过两个CNode才找到，第三个CNode经过L3 CNode句柄找到，L3 CNode存在于索引0x00处，第三个CNode没有守卫，Cap C在索引0x60处，于是，它的地址是：0x00F00060（分成三个部分：0x0 0F 00 060）。转换这个无符号整，没有剩余比特的位了。

Caps C-G. 这样的句柄区间是通过提供一个基地址来表示的，这个基地址就是它的第一个槽Cap C的地址，它是0x00F00060，窗口是5。

L2 CNode Cap. 回想一个CNode句柄，用户必须不只提供一个句柄地址，还要提供一个限制深度（depth limit），深度限制就是有多少比特位参与地址转换。L2 CNode句柄存在于偏移值为0x0F的地方，也就是距离第一个CNode0xF0个槽，它有4个比特的守卫，所以它的地址是0x00F00000，限制深度是12比特。

L3 CNode Cap. 这个句柄存在于索引值为0x00的地方，在第二个CNode中，第二个CNode有4比特被设为0的守卫，所以这个句柄的地址是0x00F00000，限制深度是24比特。注意L2和L3 CNode句柄的地址相同，但是它们的限制深度不同。

总结一下，引用句柄空间CSpace中的句柄或槽，用户必须提供它的地址，当这个句柄可能是CNode时，用户还必须提供它的限制深度。指定一个句柄区间，用户需要提供首个句柄的地址及窗口大小。

3.4 查找失败描述（Lookup Failure Description）

当查找一个句柄出错，seL4将给出一个描述信息，这个信息存放在调用线程，或线程的异常处理者（exception handler），这个描述信息的格式是固定的，但会因为错误的不同而存放在不同偏异的地方。

下面会描述这个信息的格式。第一个字节放的是错误码，其它字节的定义与错误码相关。

3.4.1 Invalid Root

你传给查找一个句柄的CSpace CPTR的根是错的（invalid的，不合法的），例如，这个句柄不是一个CNode句柄。

Data	Meaning
Offset + 0	seL4 InvalidRoot

3.4.2 Missing Capability

找不到句柄，或没有足够的权限调用这个句柄。

Data	Meaning
Offset + 0	seL4 MissingCapability
Offset + 1	*Bits left*

3.4.3 Depth Mismatch

深度不匹配，当解析一个句柄时，When resolving a capability, a CNode was traversed that resolved more bits than was left to decode in the CPTR or a non-CNode capability was encountered while there were still bits remaining to be looked up.

Data	Meaning
Offset + 0	seL4 DepthMismatch
Offset + 1	Bits of CPTR remaining to decode
Offset + 2	Bits that the current CNode being traversed resolved

3.4.4 Guard Mismatch

守卫不匹配，当解析一个句柄时，剩下的比特数与守卫（guard）尺寸不匹配，也就是剩下的比特位数，连与守卫比较都不够，或者比特位数够，但两者不相同。

Data	Meaning
Offset + 0	seL4 GuardMismatch
Offset + 1	剩下还没解析的比特位 Bits of CPTR remaining to decode
Offset + 2	The CNode's guard
Offset + 3	The CNode's guard size

3.5 句柄调用 Capability Invocation

用 seL4 内核提供的方法 Send 或 Wait 可以调用句柄，调用时传的参数是 CSpace 地址。

所有消息传递都是单向的，通常应用场景下，用户线程永远不会即是消息发送者，也是消息接收者，实际上，大多数句柄，只是实现两个操作中的一个，试图做其它操作将导致失败。

消息的接收者是谁，工作在哪个线程，内核知道，内核知道哪个句柄对应哪个计算资源。对于大多数对象类型，内核本身就是一个消息的接收者，当然，消息也可能发送到其它用户线程中。

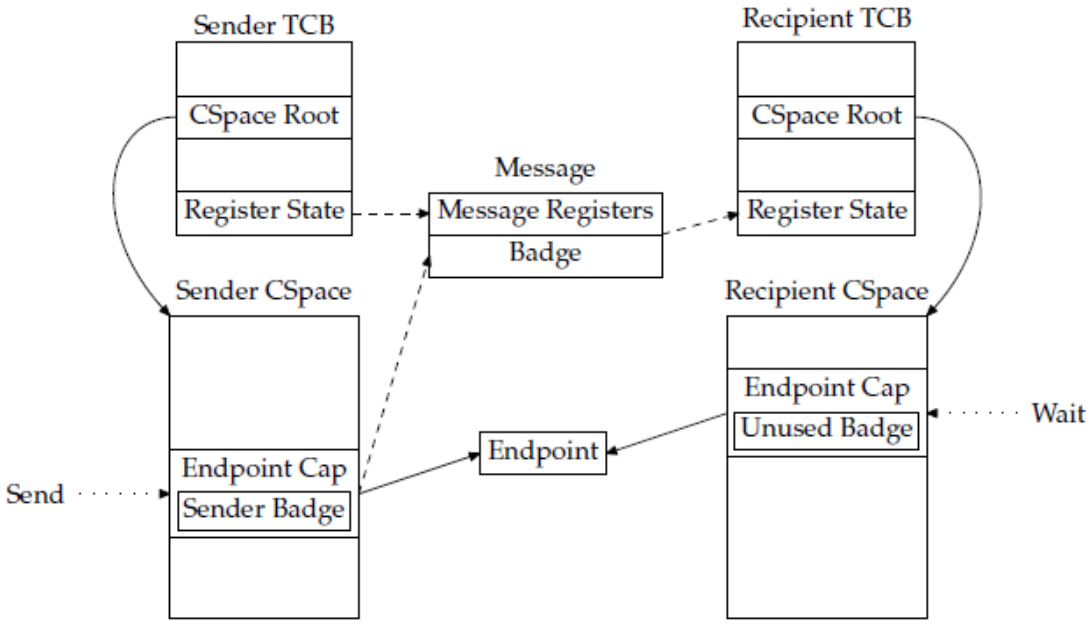

图 3.4 IPC 操作

没有消息缓冲区，虚线表示引用内核对象，线段表示线程间的消息传输

4 进程间通信 IPC

进程间通信Inter-process Communication

进程间通信就是在不同进程之间传播或交换信息，那么不同进程之间存在着什么双方都可以访问的介质呢？进程的用户空间是互相独立的，一般而言是不能互相访问的，唯一的例外是共享内存区。除此以外，那就是双方都可以访问的外设了。在这个意义上，两个进程当然也可以通过外存上的普通文件交换信息，或者通过“注册表”或其它数据库中的某些表项和记录交换信息。广义上这也是进程间通信的手段，但是一般都不把这算作“进程间通信”。

象Linux这样的操作系统，内核自己管理内存空间，这个内核持有的内存，即系统空间是个“公共场所”，也就是说，内核提供这样的数据交换的条件。但seL4内核不直接持有内存，一经起动完成，就不再动态申请内存。这条路是走不通的。

seL4微内核提供了线程间传递消息的机制，这个机制也用来传递用户程序与内核提供的服务之间的通信。消息被传送到同步（Endpoint）或异步（AsyncEP）IPC端点，这些端点由其它线程约定。其它消息类型由内核约定。本章将讨论通用的消息格式、两种类型（同步、异步）端点，及它们在不同应用间如何通信。

4.1 消息寄存器 Message Register

每个消息包含一些消息字（word，32平台上，seL4把一个32位无符号整数称为一个字），还可以包含一些句柄。消息的传递是尽量通过通用寄存器（register）来完成，但是哪个硬件平台的通用寄存器的数量都是十分有限的，于是寄存器放不下的就要放到被称为IPC缓存（IPC buffer）的内存中。线程的IPC缓存区的设置见“seL4_TCB_SetIPCBuffer()的使用（§5.1、§9.3.21）”。

每当说起寄存器，都想让人明白一件事：寄存器是CPU的一部分，而内存不是。CPU访问寄存器，不需要任何设置，它在CPU实现时，就是统一在一起的数字电路。CPU访问内存则不然，要经过北桥这样的器件。明白这些道理，就容易明白寄存器的实现代价（功耗、晶体管数量等等）及两者访问的时延特性。

表4.1 IPC时物理寄存器的使用（IA-32架构）

作用 Role	CPU寄存器 CPU Register
句柄寄存器 Capability register (in)	ebx
标记寄存器 Badge register (out)	ebx

Message tag (in/out)	esi
Message register 1 (in/out)	edi
Message register 2 (in/out)	ebp

通过下面的源码，更容易理解消息寄存器里都放了些啥（IA-32体系结构）：

```
static inline void
seL4_Send(seL4_CPtr dest, seL4_MessageInfo_t msgInfo)
{
    asm volatile (
        "pushl %%ebp       \n"
        "movl %%ecx, %%ebp \n"
        "movl %%esp, %%ecx \n"
        "leal 1f, %%edx    \n"
        "1:                \n"
        "sysenter          \n"
        "popl %%ebp        \n"
        :
        : "a" (seL4_SysSend),
        "b" (dest),
        "S" (msgInfo.words[0]),
        "D" (seL4_GetMR(0)),
        "c" (seL4_GetMR(1))
        : "%edx"
    );
}
```

每个IPC消息包含一个标记（tag）。这个标记的数据结构为：structure seL4_MessageInfo_t，包含4个字段：

- 标签，label
- 消息长度，length字段
- 句柄数量，extraCaps字段
- capsUnwrapped

表4.2 IPC时物理寄存器的使用（ARM架构）

作用 Role	CPU寄存器 CPU Register
句柄寄存器 Capability register (in)	r0
标记寄存器 Badge register (out)	r0
Message tag (in/out)	r1
Message register 1-4 (in/out)	r2 - r5

通过下面的源码，更容易理解消息寄存器里都放了些啥（ARM体系结构）：

```
static inline void
seL4_Send(seL4_CPtr dest, seL4_MessageInfo_t msgInfo)
{
    register seL4_Word destptr asm("r0") = (seL4_Word)dest;
    register seL4_Word info asm("r1") = msgInfo.words[0];
```

```
    /* Load beginning of the message into registers. */
    register seL4_Word msg0 asm("r2") = seL4_GetMR(0);
    register seL4_Word msg1 asm("r3") = seL4_GetMR(1);
    register seL4_Word msg2 asm("r4") = seL4_GetMR(2);
    register seL4_Word msg3 asm("r5") = seL4_GetMR(3);

    /* Perform the system call. */
    register seL4_Word scno asm("r7") = seL4_SysSend;
    asm volatile ("swi %[swi_num]"
                  : "+r" (destptr), "+r" (msg0), "+r" (msg1), "+r" (msg2),
                  "+r" (msg3), "+r" (info)
                  : [swi_num] "i" __SWINUM(seL4_SysSend), "r"(scno)
                  : "memory");
}
```

消息的长度和句柄的数量决定了消息寄存器的使用及发送线程想要传递的句柄的数量。内核不解释label字段。label字段只是被当作消息的第一个数据被传送。用户程序可以对label字段进行自己的定义。capsUnwrapped字段只被接收方使用，用来指明哪个句柄被接收。

表4.3 seL4_IPCBuffer数据结构，注意badges和caps使用共同的一块内存区

类型 Type	字段名 Name	描述 Description
seL4_MessageInfo_t	tag	Message tag
seL4_Word[]	msg	Message contents
seL4_Word	userData	Base address of the structure, used by supporting user libraries
seL4_CPtr[] (in)	caps	Capabilities to transfer
seL4_CapData_t[] (out)	badges	Badges for endpoint capabilities received
seL4_CPtr	receiveCNode	CPTR to a CNode from which to nd the receive slot
seL4_CPtr	receiveIndex	CPTR to the receive slot relative to re- ceiveCNode
seL4_Word	receiveDepth	Number of bits of receiveIndex to use

```
/* message_info_t defined in api/types.bf */

enum seL4_MsgLimits {
    seL4_MsgLengthBits = 7,
    seL4_MsgExtraCapBits = 2
};

enum {
```

```
    seL4_MsgMaxLength = 120,
};
#define seL4_MsgMaxExtraCaps (BIT(seL4_MsgExtraCapBits)-1)

typedef struct seL4_IPCBuffer_ {
    seL4_MessageInfo_t tag;
    seL4_Word msg[seL4_MsgMaxLength];
    seL4_Word userData;
    seL4_Word caps_or_badges[seL4_MsgMaxExtraCaps];
    seL4_CPtr receiveCNode;
    seL4_CPtr receiveIndex;
    seL4_Word receiveDepth;
} seL4_IPCBuffer __attribute__ ((__aligned__ (sizeof(struct seL4_IPCBuffer_))));
```

-

下面是与seL4_MessageInfo_t 相关的源码，根据这些源码，你才能很好地理解tag，因为它是被放在一个32位的寄存器里的。

从源码里，能看到IPC消息的tag中的各个部分：label、length等。从数据的存储上能看出，label最大值0xfffff（1048575），length最大值0x7f（127），capsUnwrapped占3 bit，extraCaps占2 bit。

```
struct seL4_MessageInfo {
    uint32_t words[1];
};
typedef struct seL4_MessageInfo seL4_MessageInfo_t;

static inline seL4_MessageInfo_t CONST
seL4_MessageInfo_new(uint32_t label, uint32_t capsUnwrapped, uint32_t extraCaps,
uint32_t length) {
    seL4_MessageInfo_t seL4_MessageInfo;

    seL4_MessageInfo.words[0] = 0;

/* fail if user has passed bits that we will override */
    assert((label & ~0xfffff) == 0);

    seL4_MessageInfo.words[0] |= (label & 0xfffff) << 12;
/* fail if user has passed bits that we will override */
    assert((capsUnwrapped & ~0x7) == 0);

    seL4_MessageInfo.words[0] |= (capsUnwrapped & 0x7) << 9;
/* fail if user has passed bits that we will override */
    assert((extraCaps & ~0x3) == 0);

    seL4_MessageInfo.words[0] |= (extraCaps & 0x3) << 7;
/* fail if user has passed bits that we will override */
    assert((length & ~0x7f) == 0);

    seL4_MessageInfo.words[0] |= (length & 0x7f) << 0;

    return seL4_MessageInfo;
}
```

内核假定IPC缓存中包含一个表4.3定义的数据结构。内核使用尽量多的物理寄存器来传递IPC消息，当寄存器不够用时，通过IPC缓存的msg字段来传递。但是，既使有些消息内容是在寄存器中，msg中依然为这些寄存器内容留有空间。假如下面的情形，一个IPC消息需要4个消息寄存器，但是当前平台（指IA-32、ARM这样的platform）只有2个寄存器可用，则，参数1、2通过寄存器传递，参数3、4通过msg[2]、msg[3]传递，msg[0]、msg[1]留着，这方便应用程序通过把msg拷贝走的方式保留消息，注意，这时的msg[0]、msg[1]中是没有参数数据的。同样的情况也适用于tag字段。有空间在msg字段，但内核忽略这些保留的空间。

★应用程序编程时，IPC缓存中对应寄存器的内容是不能直接操作的，象这样的函数：seL4_SetUserData、seL4_SetCap，应用程序编程时是用不到的。

实际编程中，seL4_MessageInfo_t这个数据结构可能是seL4_MessageInfo_new()之后，一点一点把内容放进去的。

```
seL4_MessageInfo_t info;
seL4_Word badge;
info = seL4_Wait(aep, &badge);
assert(badge != 0);
if (sep != seL4_CapNull) {
    /* Synchronous endpoint registered. Send IPC */
    info = seL4_MessageInfo_new(label, 0, 0, 2);
    seL4_SetMR(0, badge);
    seL4_SetMR(1, node_ptr);
    seL4_Send(sep, info);
} else {
    /* No synchronous endpoint. Call the handler directly */
    irq_server_node_handle_irq(st->node, badge);
}
```

用来操作seL4_MessageInfo_t这个数据结构的函数，还有（这里以ARM平台为例列出）：
$/libs/libsel4/arch_include/arm/sel4/arch/functions.h

```
static inline seL4_IPCBuffer* seL4_GetIPCBuffer(void)
static inline seL4_MessageInfo_t seL4_GetTag(void)
static inline void seL4_SetTag(seL4_MessageInfo_t tag)
static inline seL4_Word seL4_GetMR(int i)
static inline void seL4_SetMR(int i, seL4_Word mr)
static inline seL4_Word seL4_GetUserData(void)
static inline void seL4_SetUserData(seL4_Word data)
static inline seL4_CapData_t seL4_GetBadge(int i)
static inline void seL4_SetCap(int i, seL4_CPtr cptr)
static inline void seL4_SetCapReceivePath(seL4_CPtr receiveCNode, seL4_CPtr
receiveIndex, seL4_Word receiveDepth)
```

4.2 同步端点 Synchronous Endpoint

同步端点（synchronous endpoint，simply endpoint）允许少量的数据及句柄在线程间同步传递。

所谓同步，就是信息发出去了，发送者要等待应答。如果发送者没处于要发送的状态，而且接收者正好处于接收状态，这个消息就不被传递。

同步消息传递的API是seL4_Send()或seL4_Call()。如果发出IPC同步请求，没有接收者准备好，则阻塞，排队等待，直到有接收者，就把消息发送给第一个准备好了的接收者。

同步消息接收的API是seL4_Wait()或seL4_ReplyWait()。如果发出等待请求给内核了，没人向它发消息，阻塞，排队等待，直到有发送者，就接收第一个发送者的消息。

源码（$/libs/libsel4vka/include/vka/object.h）中，用vka创建同步端点、异步端点。

```
static inline int vka_alloc_endpoint(vka_t *vka, vka_object_t *result)
{
    return   vka_alloc_object(vka,   seL4_EndpointObject,   seL4_EndpointBits,
result);
}
static inline int vka_alloc_async_endpoint(vka_t *vka, vka_object_t *result)
{
    return vka_alloc_object(vka, seL4_AsyncEndpointObject, seL4_EndpointBits,
result);
}
```

4.2.1 端点标记 Endpoint Badge

同步端点可以被锻造Mint出新的端点，如何区分出这两个端点呢，端点标记（endpoint badge）。端点标记是附属在端点句柄上的一个32位无符号整数（word）。当带标记的端点句柄发出消息时，这个标记被传递到接收线程的标记寄存器。

如果一个端点的标记badge是0，则表示它没有被标记（unbadged）。标记一个端点的API是seL4_CNode_Mutate()或seL4_CNode_Mint()。已经被标记的端点不能撤销标记、重置标记或创建具有不同标记的子句柄。

程序示例，给一个端点加上标记。

```
static seL4_CPtr
badge_endpoint(env_t env, int badge, seL4_CPtr ep)
{

    seL4_CapData_t cap_data = seL4_CapData_Badge_new(badge);
    seL4_CPtr slot = get_free_slot(env);
    int error = cnode_mint(env, ep, slot, seL4_AllRights, cap_data);
    test_assert(!error);
    return slot;
}
```

4.2.2 句柄传输 Capability Transfer

同步消息中可以有句柄，消息中的句柄必须有Grant权限，没有Grant权限的句柄，不会随消息的传递而传递。

句柄存在于消息的caps字段，caps中的每一项被当作CPTR解释为线程句柄空间中的一个句柄。消息中句柄的数量存于消息标记（message tag）中的extraCaps字段中。

IPC消息的接收方要指定存放句柄的槽slot，假如通过seL4_SetCapReceivePath()指定接收槽信息，seL4_SetCapReceivePath()的源码如下：

```
seL4_SetCapReceivePath(seL4_CPtr    receiveCNode,    seL4_CPtr    receiveIndex,
seL4_Word receiveDepth)
{
    seL4_IPCBuffer* ipcbuffer = seL4_GetIPCBuffer();
    ipcbuffer->receiveCNode = receiveCNode;     //root CNode
    ipcbuffer->receiveIndex = receiveIndex;     //句柄地址
    ipcbuffer->receiveDepth = receiveDepth;     //地址中要被解析的比特位数
}
```

除非接收方端点（receiving endpoint capability）没有Write权限，否则接收句柄与发送方线程中的句柄具有同样的权限。接收方端点没有Write权限这种情况下，接收来的句柄的写权限将被移除。

接收方只能接收一个句柄的槽，可是IPC消息中可能包含多个句柄。IPC消息中多于1个句柄的情形，由内核根本不同情况进行处理。

Note that receiving threads may specify only one receive slot, whereas a sending thread may include multiple capabilities in the message. Messages containing more than one capability may be interpreted by kernel objects. They may also be sent to receiving threads in the case where some of the extra capabilities in the message can be unwrapped.

如果消息中第n个句柄是消息发送端点句柄endpoint，那么这个句柄就是unwrapped，它的标记badge（即消息发送端点句柄endpoint的badge）将被放在接收标记的第n个位置上，而且消息的tag的capsUnwrapped字段的第n个比特位被置为1，句柄本身不再被传输，所以接收槽可以用来接收另外一个句柄。

如果接收方收到一个消息的tag中的extraCaps是2，capsUnwrapped被设为2，那么，消息中第1个句柄将被传送到指定的槽slot，第二个句柄没被包装（unwrapped），它的标记badge被放在badges[1]。如果IPC消息中有3个句柄，那这第三个句柄就不能被解包装（unwrap）

4.2.3 错误 Errors

句柄传输有两种出错的场合：发送阶段、接收阶段。

发送阶段，所有发送API都会先检查目的地的句柄存在，这个检查通过了，内核才能发送send这个行为初始化。无论什么原因，这个检查没通过，seL4_Send()和seL4_Call()系统调用将立即终止，没有IPC或句柄传递行为发生。系统将返回一个查找错误（lookup

failure error），见§9.1。

接收阶段，seL4传送句柄的顺序是发送线程IPC缓存区中的caps数组，也就是消息中的所有句柄都要检查一下，这时可能发生下列错误：

- 源句柄找不到。虽然发送IPC消息时，发送方已经检查源句柄的存在，但还是可能发生错误。在发送方发送完IPC消息后，它可能被阻塞，发生了N多事情之后，你才收到这个消息，这期间，发送方的CSpace可能有变化，源句柄可能已经失效。

- 目的槽slot找不到。不象发送消息的系统调用，seL4_Wait()在调用时，不检查目的槽slot是否存在，并且为空。seL4_Wait()没试图把句柄写入槽slot前，象传输标记了的句柄（badged capability）都已经做了。

- 被传输的句柄不能被继承，细节见§3.1.4。

即使IPC过程中发生了错误，也不会使整个传输失效，只是使得传输过早地终止了。失败前处理的句柄处理还是传输，接收IPC缓存（IPC Buffer）extraCaps字段被设为出错前传输的句柄的数量。

上面提到的哪种情况下，接收线程都不会得到一个出错返回。

4.3 异步端点 Asynchronous Endpoint

异步端点（Asynchronous endpoint，AsyncEP）允许非阻塞异步发送消息，但异步消息中不能有句柄。

每个异步端点存有一个32位无符号整数，这个整数可以通过seL4_Notify()写入，seL4_Notify()是把第一个消息寄存器里面的内容与异步端点中的整数位或（bitwise or）运算。

要注意seL4_Notify()不是一个正常的内核功能调用，它是被seL4用态态程序库包装出来的。它只通过一个参数调用seL4_Send()。它对向异步端点发送消息很实用。

seL4_Wait()获取异步端点上的异步消息，异步消息是按位（bit）定义的，所以你主位解释这个异步消息的32位无符号整数吧。seL4_Wait()接收完异步消息后，把这个异步端点清除。

如果调用seL4_Notify()前，没有人向当前线程发送异步消息，则阻塞，直到有人通过seL4_Notify()发送消息。

4.3.1 异步端点标记 Asynchronous Endpoint Badges

象同步端点一样，异步端点也可以通过Mint锻造出一个附着了标记（badge）的新端点。标记是附着在端点句柄上的一个32位无符号整数字，当一个消息被有标记的句柄发送出来，标记成为接收消息的一部分，标记与之前通过这个端点接收来的标记被做位或（bitwise or）运算。

如果一个异步端点的标记badge是0，则表示它没有被标记（unbadged）。标记一个端点的API是seL4_CNode_Mutate()或seL4_CNode_Mint()。已经被标记的端点不能撤销标记、

重置标记或创建具有不同标记的子句柄。

4.4 同步端点与 IPC（Synchronous Endpoints and IPC）

基本的IPC（inter-process communication）概念是端点（endpoint）之间的事。端点是一个内核小对象，其中包含了一个线程队列，和一个状态（state）标志，该标志用来记录队列中的线程是处于要发送（Send）消息的状态，还是处于等待接收消息（Wait）的状态。它扮演了一个“一个或多个发送者”与“一个或多个接收者”之间单向通信的作用。不象其它版本的L4，这个全局的标识符是不暴露给参与IPC的各方的，无论是发送者，还是接收者，都是使用一个本地端点地址来干活的。

一个端点上，可能有多个线程用来发送或接收消息。当一个线程对调用端点，要发送与接收消息，内核将扮演在它们之间传递消息的角色。没有伙伴的线程，就是你要发没人收，或你要收没人发，这样的孤单线程，将被挂起（suspend），算法中并未规定，哪些线程首先被唤醒（resume），一般是先进先出（FIFO）。

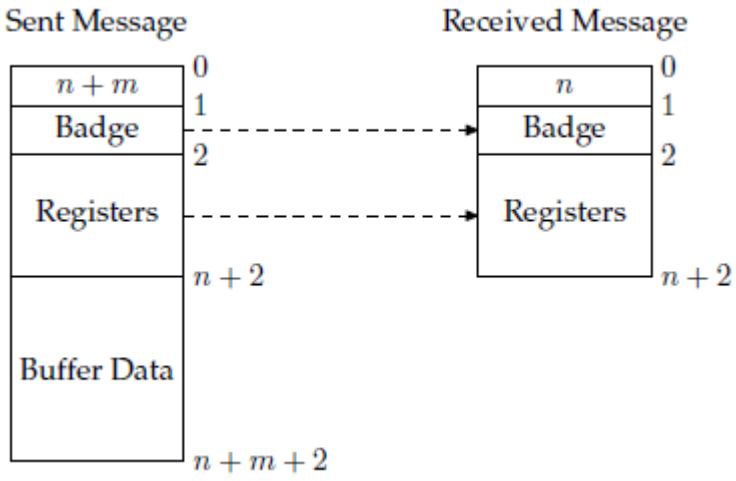

图4.1 当一个缓冲区丢掉了时的IPC，消息被截断

因为同一时间一个端点（endpoint）只能接收一个消息，而一个服务者希望为N多客户端提供服务，于是每个客户端都拿到了这个端点的一个引用（指向那个端点的在自己的句柄空间的句柄），咋标记这些句柄呢？内核提供了一个标记（badge）的办法，有标记的句柄就是标记句柄（badged endpoint capability）。一个标记就是一个数，由产生它的那个线程（我们可以称它为服务线程，Server）定义，因为这个句柄就是服务线程通过seL4_CNode_Mint()之类的手段得到的。

当消息被标记过的端点发送出去时，这个标记将同步传输出去。

有时，一个动作涉及到多个对象，如果这时权限又合适，这些对象就会同时发出同样的消息，可是，只有一个句柄是真的应该调用的，所以服务者，当接通收到消息后，要检查一下标记，看是哪个对象发给他的。也就是你作为服务者，真的提供服务时，要通过标记的方式检查一下来源，那个消息是不是真的应该被提供服务者发出来的。

内核提供了辨识（identify）系统，它允许你同时调N个对象

The kernel provides the Identify system call to allow such multiple-object invocations.
To use it, a server provides two capability references: one to the server’s endpoint capability,
which must have the right to mint new badged capabilities; and one to the

additional capability provided by the client. If the capability provided by the client is valid and refers to the server's endpoint, then the call will return the badge on the client's capability; otherwise an error code will be returned. This allows servers to provide badged capabilities to clients which may be identified later, without allowing untrusted clients to read the badges.

4.4 系统功能调用（Sysyem Calls）

图4.2展示了一个由客户端线程发起的对内核的请求。客户线程向内核对象发送一个句柄和指定的应答句柄，这个句柄是一个端点（endpoint），它负责接收内核对客户端的请求信息。它向内核对象发送一个请求，这个请求被发送到内核指定的对象，让内核指定对象替客户端做指定的事情。

接到客户线程的请求消息后，内核分析这个请求包含的具体操作，然后执行它。如果这些请求是这个内核对象无能为力的，就返回出错信息。

客户线程发出请求后，阻塞，等待从应答句柄（Reply Capability）接收到消息。内核发出的应答消息是非阻塞的，就是，当前如果没有线程等待接收这个应答，则把消息丢弃。SendWait系统功能调用就是让客户线程发出一个请求后，立即等着接收消息的原子操作。

Figure 2.3 on the following page shows a request being made of the kernel by a client thread. The client possesses a capability to a kernel object, and a designated reply capability, which is an endpoint capability that the client (or a system thread controlling the client) has nominated to be used for receiving replies to requests. It performs a Send operation to the kernel object, which transfers a message to the kernel along with a reference to the invoked kernel object.

After receiving a message from a client thread, the kernel determines which operation is being requested, based on the invoked object's type and the contents of the message; it then performs the operation if possible, and sends a message to the thread's reply

endpoint giving the result of the operation. The client is expected to have performed a Wait operation on its reply capability to receive this message—the reply is treated as if

the kernel performed a non-blocking Send operation, and will be dropped if neither the client nor any other thread is waiting to receive it. To allow the client to immediately perform the Wait operation after sending the request, the kernel provides an atomic Send andWait operation called SendWait.

图4.3展示了客户线程请求的服务是由用户态服务提供的情形。客户请求一个端点，这个端点代表了这个服务，服务线程接到了这个请求消息，通过发送句柄中的标记（badge）区分是哪个客户线程发来的请求。通常情况下，一个服务线程只通过一个句柄接收请求，每个客户线程一个应答句柄。

Figure 2.4 shows the procedure followed when a client requests a service provided by a user-level server thread. Rather than invoke a kernel object capability, the client invokes an endpoint which represents the service;

the server thread receives the message,

determines the identity of the client using the sending capability's badge, performs the operation, and sends the result to the client's reply capability. Generally, a server will possess a single capability used to receive requests, and also one reply capability for each of its clients; a client will have a single reply capability, and one capability for each object or server it is permitted to access.

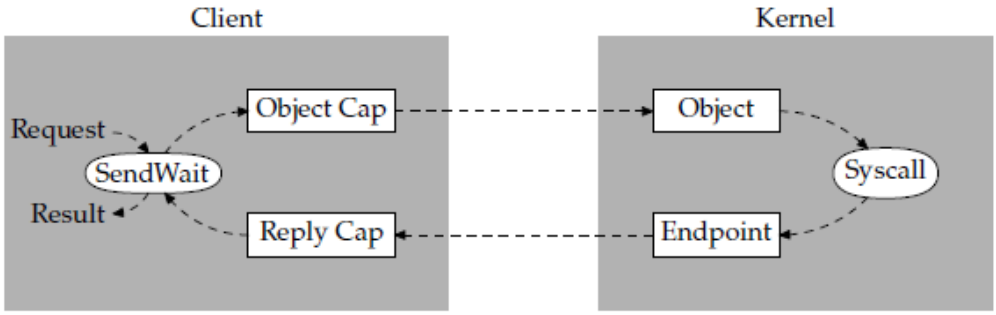

图4.2 调用内核服务，线段表示调用路径及调用过程中的数据传输

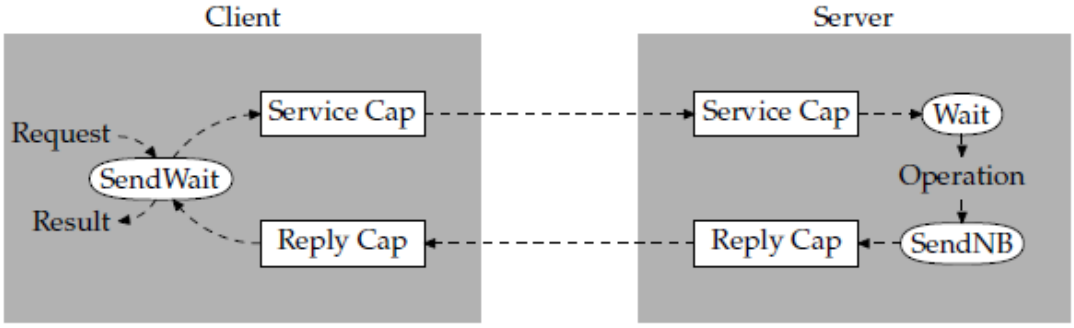

图4.3 调用在用户态空间提供的服务

客户、服务器间通信与客户与内核间通信很相似。实际上，客户线程发出请求后，调度程序根本不给它执行的机会。从客户线程的角度，一个服务是由内核提供的，还是由其它线程提供的，它无从感知。

利用这种机制，可以做一些系统监控（monitor）的事，让内核得到请求后，先发给monitor线程，monitor线程决定这个请求是否要被拒绝服务、写log、还是其它一些行为。monitor线程处理完后，再把请求交给最终的服务者。因为有时候句柄是通过系统调用的参数传递的，所以monitor要把这类句柄替换为它的句柄（即在它的句柄空间是合理值的句柄）。

5 线程及其执行

线程与执行 Threads and Execution

5.1 Threads（线程）

线程（Thread）是seL4的程序执行单位，程序通过管理线程来管理CPU时间及程序执行的上下文环境（context）。线程的管理机构（对应的是seL4内核对象，这个对象里有对线程描述所需要的数据结构）是线程控制块（thread control block，TCB）。

每个线程控制块都有一个句柄空间CSpace和虚拟地址空间VSpace与之关联。虚拟地址空间VSpace可以同时被多个线程共享。

每个线程控制块有一个IPC通信用的缓存区buffer，用来在IPC或内核对象调用（seL4的内核对象调用，方式上也是个IPC）时传递额外的参数，所谓额外的参数，就是没法放在寄存器里的数据，不同的平台，可以使用的寄存器不同，所以这个定义与平台相关。

每个线程属于且只属于一个域（Domain）。

用户态程序中，是通过一个普通的内核句柄与TCB打交道的，这个句柄的数据类型是：seL4_CPtr。

seL4内核中TCB数据结构：

```
/* TCB CNode: size = 256 bytes */
/* typedef cte_t[16] tcb_cnode; */

/* TCB: size = 652 bytes, alignment = 256 bytes */
struct tcb {
    /* Saved user-level context of thread, 592 bytes */
    user_context_t tcbContext;

    /* Thread state, 12 bytes */
    thread_state_t tcbState;

    /* Current fault, 8 bytes */
    fault_t tcbFault;

    /* Current lookup failure, 8 bytes */
    lookup_fault_t tcbLookupFailure;

    /* Domain, 1 byte (packed to 4) */
    uint32_t tcbDomain;
```

```
    /* Priority, 1 byte (packed to 4) */
    uint32_t tcbPriority;

    /* Timeslice remaining, 4 bytes */
    word_t tcbTimeSlice;

    /* Capability pointer to thread fault handler, 4 bytes */
    cptr_t tcbFaultHandler;

    /* userland virtual address of thread IPC buffer, 4 bytes */
    word_t tcbIPCBuffer;

    /* Previous and next pointers for endpoint & scheduler queues, 16 bytes */
    struct tcb* tcbSchedNext;
    struct tcb* tcbSchedPrev;
    struct tcb* tcbEPNext;
    struct tcb* tcbEPPrev;
};
typedef struct tcb tcb_t;

/* A TCB CNode and a TCB are always allocated together, and adjacently,
 * such that they fill a 1024-byte aligned block. The CNode comes first. */
enum tcb_cnode_index {
    /* CSpace root, 16 bytes */
    tcbCTable = 0,

    /* VSpace root, 16 bytes */
    tcbVTable = 1,

    /* Reply cap slot, 16 bytes */
    tcbReply = 2,

    /* TCB of most recent IPC sender, 16 bytes */
    tcbCaller = 3,

    /* IPC buffer cap slot, 16 bytes */
    tcbBuffer = 4,

    tcbCNodeEntries
};
typedef uint32_t tcb_cnode_index_t;
```

5.1.1 线程创建 Thread Creation

象其它seL4对象一样，TCB通过seL4_Untyped_Retype()创建。

程序示例：

```
int error;
vka_object_t cnode;

error = vka_alloc_cnode_object(&env->vka, 2, &cnode);
test_assert(error == 0);

error = seL4_Untyped_Retype(untyped.cptr,
                            seL4_TCBObject, 0,
                            env->cspace_root, cnode.cptr, seL4_WordBits,
                            (1 << 2) - 1, 2);
test_assert(error == seL4_RangeError);
```

一个新创建的TCB是不活动（inactive）的状态。

seL4_TCB_SetSpace()、seL4_TCB_Configure()函数实现对TCB的CSpace、VSpace的定制。然后通过seL4_TCB_WriteRegisters()定制初始栈（initial stack pointer）和指令初始地址（instruction pointer）。

线程通过调用seL4_TCB_WriteRegisters()或seL4_TCB_Resume()激活。

5.1.2 线程停止 Thread Deactivation

线程停止，就是不让线程活动了。seL4_TCB_Suspend()的作用就是让线程停止。

已经停止了的线程，可以通过seL4_TCB_ReadRegisters()和seL4_TCB_CopyRegisters()与它打交道。

线程可以重定义，并被重新使用，也可以永久挂起。

当指向一个线程的最后一个TCB被删除，它将被自动挂起。

5.1.3 调度 Scheduling

seL4线程的几种状态如下列原代码中所示：

```
/* Thread state */
enum _thread_state {
    ThreadState_Inactive = 0,
    ThreadState_Running,
    ThreadState_Restart,
    ThreadState_BlockedOnReceive,
    ThreadState_BlockedOnSend,
    ThreadState_BlockedOnReply,
    ThreadState_BlockedOnAsyncEvent,
    ThreadState_IdleThreadState
```

```
};
typedef uint32_t _thread_state_t;
```

seL4 使用抢占式preemptive的round-robin调度算法，具有256级优先级。

一个线程操作另外一个线程，最多只能把另外一个线程优先级设得与自己一样高，就是说，线程不能造个比自己优先级还高的子线程。

设置线程优先级的方法为：seL4_TCB_Configure()和seL4_TCB_SetPriority()。

5.1.4 异常 Exceptions

每个线程有一个与它关联的处理异常的端点（exception-handler endpoint）。如果线程引起了一个异常，内核将创造一个IPC消息，并把这个IPC消息连同相关异常的细节分发到线程处理端点。至于线程如何处理它的异常，那就是程序设计者的事了。

线程异常消息的定义见§5.2。

为了处理异常，线程要在与线程关联的CSpace中申请一个endpoint端点。端点通过seL4_TCB_SetSpace()或seL4_TCB_Configure()把异常处理端点设为异常处理者，这些方法首先检查CSpace中是否有端点句柄，如果没有，那显然设置不会成功。

谁产生的异常，谁负责处理。异常处理端点未设，或者设的端点在CSpace中找不到，异常消息将不被发送。

异常处理端点必须有Send和Grant权限，应答异常，将引起线程重新开始。异常应答时，可能包含重设线程的寄存器的值。

程序示例（faults.c）：

```
seL4_CPtr fault_ep = vka_alloc_endpoint_leaky(&env->vka);

int error = seL4_TCB_Configure(faulter_thread.thread.tcb.cptr,
            fault_ep,
            prio,
            faulter_cspace,
            seL4_CapData_Guard_new(0, seL4_WordBits - env->cspace_size_bits),
            faulter_vspace, seL4_NilData,
            faulter_thread.thread.ipc_buffer_addr,
            faulter_thread.thread.ipc_buffer);
test_assert(!error);
```

5.1.5 寄存器读写方法中的消息结构

Message Layout of the Read-/Write-Registers Methods

seL4_TCB_ReadRegisters()、seL4_TCB_WriteRegisters()方法可以读到的寄存器。

寄存器内容是通过IPC缓存（IPC buffer）传输的。

IA-32

寄存器 Register	IPC缓存区位置 IPC Buffer location
EIP	IPCBuffer[0]
ESP	IPCBuffer[1]
EFLAGS	IPCBuffer[2]
EAX	IPCBuffer[3]
EBX	IPCBuffer[4]
ECX	IPCBuffer[5]
EDX	IPCBuffer[6]
ESI	IPCBuffer[7]
EDI	IPCBuffer[8]
EBP	IPCBuffer[9]
TLS_BASE	IPCBuffer[10]
FS	IPCBuffer[11]
GS	IPCBuffer[12]

ARM

Register	IPC Buffer location
PC	IPCBuffer[0]
SP	IPCBuffer[1]
CPSR	IPCBuffer[2]
R0-R1	IPCBuffer[3-4]
R8-R12	IPCBuffer[5-9]
R2-R7	IPCBuffer[10-15]
R14	IPCBuffer[16]

5.2 故障 Faults

线程干活时可能有故障，故障的处理方法与异常类似，也是由内核向线程发送异常消息。

故障类型定义在消息的label，有如下种类的故障：

```
typedef enum {
    seL4_NoFault = 0,
    seL4_CapFault,
    seL4_VMFault,
    seL4_UnknownSyscall,
    seL4_UserException,
    seL4_Interrupt,
    SEL4_FORCE_LONG_ENUM(seL4_FaultType),
} seL4_FaultType;
```

5.2.1 句柄故障 Capability Faults

有两种场合可能有句柄故障。

第一种，查找被seL4_Call()或seL4_Send()引用的句柄时。思考一下这样的场合：你向某个句柄发送一个消息，这个消息就要有接收者，内核如何知道这个接收者，在CSpace里面查啊，查不到，你说我这时把错误信息发给谁？也就是由谁负责处理这个错误？不能是人家请求者啊，人家只是给你发了个消息，只能是接收线程，可接收者不是故障了吗，于是，故障以消息的形式发给了请求者。消息中包含请求者的句柄及IPC buffer中的caps字段中的句柄。

seL4_NBSend()异常发送消息，当其使用的是无效的句柄时，不会有故障发生。

第二种，seL4_Wait()被调用时，它引用的句柄不存在、不是一个端点句柄或没有接收消息的权限。

应答故障IPC（fault IPC）将引导起线程重起（restart）。IPC消息的定义见表5.1。

表5.1 IPC消息内容

意思 Meaning	IPC缓存区位置 IPC buffer Location
Program counter to restart execution at	IPCBuffer[0]
Capability address	IPCBuffer[1]
In receive phase (1 if the fault happened	IPCBuffer[2]
during a wait system call, 0 otherwise)	
Lookup failure description. As described	IPCBuffer[3..]
in Section 3.4	

5.2.2 不明系统调用 Unknown Syscall

当线程进行系统调用异常的调用功能号（syscall number）seL4不认识，它就是一个不明系统调用。

异常线程的寄存器将被传递给线程异常处理程序，这样，线程可以“伪造”一个系统调用。

应答不明系统调用时，允许线程被重启，并且可以（不是必须）设置线程的寄存器。如果应答label是0，则线程重启。其它情况，消息长度不是0，线程的寄存器将被按照IPC的缓存区修改，被修改的寄存器的数量是消息的标记tag中的长度字段中的值。

通过下面的源码了解一下消息中的tag中的长度信息。本书第4章有对IPC消息的数据结构的详细的讨论。

```
struct seL4_MessageInfo {
    uint32_t words[1];
};
```

```
typedef struct seL4_MessageInfo seL4_MessageInfo_t;

static inline seL4_MessageInfo_t CONST
seL4_MessageInfo_new(uint32_t label, uint32_t capsUnwrapped, uint32_t extraCaps,
uint32_t length) {
    seL4_MessageInfo_t seL4_MessageInfo;

    seL4_MessageInfo.words[0] = 0;

/* fail if user has passed bits that we will override */
    assert((label & ~0xfffff) == 0);

    seL4_MessageInfo.words[0] |= (label & 0xfffff) << 12;
/* fail if user has passed bits that we will override */
    assert((capsUnwrapped & ~0x7) == 0);

    seL4_MessageInfo.words[0] |= (capsUnwrapped & 0x7) << 9;
/* fail if user has passed bits that we will override */
    assert((extraCaps & ~0x3) == 0);

    seL4_MessageInfo.words[0] |= (extraCaps & 0x3) << 7;
/* fail if user has passed bits that we will override */
    assert((length & ~0x7f) == 0);

    seL4_MessageInfo.words[0] |= (length & 0x7f) << 0;

    return seL4_MessageInfo;
}
```

ARM

表 5.2: 不明系统调用结果（ARM架构）

发送的值 Value sent	应答寄存器 Register set by reply	IPC缓存区位置 IPC buffer location
R0-R7	(same)	IPCBuffer[0-7]
FaultInstruction	(same)	IPCBuffer[8]
SP	(same)	IPCBuffer[9]
LR	(same)	IPCBuffer[10]
CPSR	(same)	IPCBuffer[11]
Syscall number	--	IPCBuffer[12]

IA-32

表 5.3: 不明系统调用结果（IA-32架构）

发送的值 Value sent	应答寄存器 Register set by reply	IPC缓存区位置 IPC buffer location
EAX	(same)	IPCBuffer[0]
EBX	(same)	IPCBuffer[1]

ECX	(same)	IPCBuffer[2]
EDX	(same)	IPCBuffer[3]
ESI	(same)	IPCBuffer[4]
EDI	(same)	IPCBuffer[5]
EBP	(same)	IPCBuffer[6]
EIP	(same)	IPCBuffer[7]
ESP	(same)	IPCBuffer[8]
EFLAGS	(same)	IPCBuffer[9]
Syscall number	--	IPCBuffer[10]

5.2.3 用户异常 User Exception

用户异常用来分发系统硬件体系结构相关的异常，例如，被0除异常。

应答用户异常时，允许线程被重启，并且可以（不是必须）设置线程的寄存器。如果应答label是0，则线程重启。其它情况，消息长度不是0，线程的寄存器将被按照IPC的缓存区修改，被修改的寄存器的数量是消息的标记tag中的长度字段中的值。

ARM

表5.4：用户异常调用结果（ARM架构）

发送的值 Value sent	应答寄存器 Register set by reply	IPC缓存区位置 IPC buffer location
FaultInstruction	(same)	IPCBuffer[0]
SP	(same)	IPCBuffer[1]
CPSR	(same)	IPCBuffer[2]
Exception number	--	IPCBuffer[3]
Exception code	--	IPCBuffer[4]

IA-32

表5.5：用户异常调用结果（IA-32架构）

发送的值 Value sent	应答寄存器 Register set by reply	IPC缓存区位置 IPC buffer location
EIP	(same)	IPCBuffer[0]
ESP	(same)	IPCBuffer[1]
EFLAGS	(same)	IPCBuffer[2]
Exception number	--	IPCBuffer[3]
Exception code	--	IPCBuffer[4]

5.2.4 缺页 VM Fault

线程引起的缺页中断而发出的异常，应答这个异常将引起线程重启。

表5.6：缺页异常调用结果

意思 Meaning	IPC缓区存 IPC buffer location
Program counter to restart execution at.	IPCBuffer[0]
Address that caused the fault.	IPCBuffer[1]
Instruction fault (1 if the fault was caused by an instruction fetch).	IPCBuffer[2]
Fault status register (FSR). Contains information about the cause of the fault. Architecture dependent.	IPCBuffer[3]

5.3 域（Domains）

域（domain）是用来隔离不相关的子系统，这样就限制了信息在它们之间的流动。系统按固定的、时间触发的机制在域间调度。固定的域在编译时被编译进内核里，通过常量：NUM_DOMAINS、和全局环境变量ksDomSchedule。

一个线程严格属于一个域，只有当那个域是活动的时它才运行。seL4_DomainSet_Set()有来改变线程的域。

初始化线程开始于域（Domain）seL4_CapDomain。见初始化线程环境部分的介绍，定义于bootingo.h。

```
/* caps with fixed slot potitions in the root CNode */

enum {
    seL4_CapNull                =  0, /* null cap */
    seL4_CapInitThreadTCB       =  1, /* initial thread's TCB cap */
    seL4_CapInitThreadCNode     =  2, /* initial thread's root CNode cap */
    seL4_CapInitThreadPD        =  3, /* initial thread's PD cap */
    seL4_CapIRQControl          =  4, /* global IRQ controller cap */
    seL4_CapASIDControl         =  5, /* global ASID controller cap */
    seL4_CapInitThreadASIDPool  =  6, /* initial thread's ASID pool cap */
    seL4_CapIOPort               =  7, /* global IO port cap (null cap if not
supported) */
    seL4_CapIOSpace             =  8, /* global IO space cap (null cap if no IOMMU
support) */
    seL4_CapBootInfoFrame       =  9, /* bootinfo frame cap */
    seL4_CapInitThreadIPCBuffer = 10, /* initial thread's IPC buffer frame cap
*/
    seL4_CapDomain              = 11  /* global domain controller cap */
```

```
};
```

从上面的程序定义可以看出，seL4_CapDomain只是一个枚举enum方式定义出来的一个整型值，但它被当作一个seL4_CPtr用，可以直接通过下面这个函数：

```
seL4_CPtr simple_default_init_cap(void *data, seL4_CPtr cap_pos) {
    return (seL4_CPtr) cap_pos;
}
```

得到一个真的句柄，而且这个句柄就是它自身这个整型值强制类型转换而得到。

5.4 线程控制块 TCB

seL4中，TCB（线程管制块，Thread Control Block）对象代表的是执行状态的线程。线程是seL4是基本调试单位，它是否阻塞、执行等等状态取决于它与其它应用间的互操作。

图5.1演示了与一个线程相关的CSpace、VSpace等之间的关系。

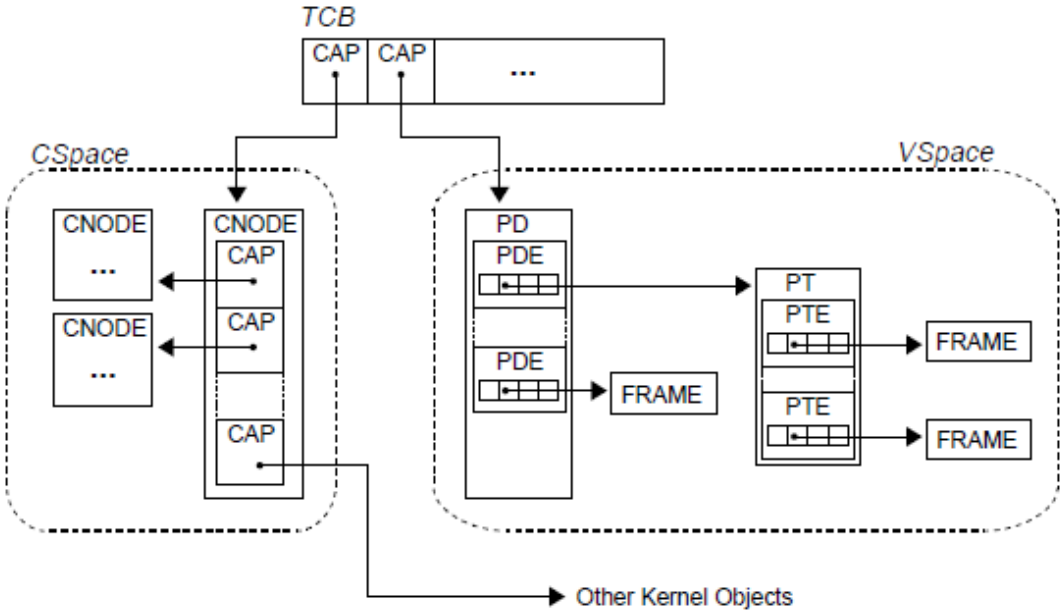

图5.1 与一个应用相关的内部数据结构图

一个TCB对象有下列方法（参sel4.xml中的定义）：

```
<interface name="seL4_TCB">
    <method id="TCBReadRegisters" name="ReadRegisters">
        <param dir="in" name="suspend_source" type="bool"/>
        <param dir="in" name="arch_flags" type="uint8_t"/>
        <param dir="in" name="count" type="seL4_Word"/>
        <param dir="out" name="regs" type="seL4_UserContext"/>
    </method>
    <method id="TCBWriteRegisters" name="WriteRegisters">
        <param dir="in" name="resume_target" type="bool"/>
        <param dir="in" name="arch_flags" type="uint8_t"/>
        <param dir="in" name="count" type="seL4_Word"/>
        <param dir="in" name="regs" type="seL4_UserContext"/>
```

```
        </method>
        <method id="TCBCopyRegisters" name="CopyRegisters">
            <param dir="in" name="source" type="seL4_TCB"/>
            <param dir="in" name="suspend_source" type="bool"/>
            <param dir="in" name="resume_target" type="bool"/>
            <param dir="in" name="transfer_frame" type="bool"/>
            <param dir="in" name="transfer_integer" type="bool"/>
            <param dir="in" name="arch_flags" type="uint8_t"/>
        </method>
        <method id="TCBConfigure" name="Configure">
            <param dir="in" name="fault_ep" type="seL4_Word"/>
            <param dir="in" name="priority" type="uint8_t"/>
            <param dir="in" name="cspace_root" type="seL4_CNode"/>
            <param dir="in" name="cspace_root_data" type="seL4_CapData_t"/>
            <param dir="in" name="vspace_root" type="seL4_CNode"/>
            <param dir="in" name="vspace_root_data" type="seL4_CapData_t"/>
            <param dir="in" name="buffer" type="seL4_Word"/>
            <param dir="in" name="bufferFrame" type="seL4_CPtr"/>
        </method>
        <method id="TCBSetPriority" name="SetPriority">
            <param dir="in" name="priority" type="uint8_t"/>
        </method>
        <method id="TCBSetIPCBuffer" name="SetIPCBuffer">
            <param dir="in" name="buffer" type="seL4_Word"/>
            <param dir="in" name="bufferFrame" type="seL4_CPtr"/>
        </method>
        <method id="TCBSetSpace" name="SetSpace">
            <param dir="in" name="fault_ep" type="seL4_Word"/>
            <param dir="in" name="cspace_root" type="seL4_CNode"/>
            <param dir="in" name="cspace_root_data" type="seL4_CapData_t"/>
            <param dir="in" name="vspace_root" type="seL4_CNode"/>
            <param dir="in" name="vspace_root_data" type="seL4_CapData_t"/>
        </method>
        <method id="TCBSuspend" name="Suspend"/>
        <method id="TCBResume" name="Resume"/>
    </interface>
```

- -

6 地址空间与虚拟内存

读本章前，建议把IA-32、ARM等体系结构的存储管理、MMU、页表管理之类的说法弄弄懂，否则，很难一上来就把虚拟地址的东西弄懂。

虚拟内存必须与物理内存映射起来才能使用，操作未经映射的虚拟内存，将引起Page Fault，缺页中断。

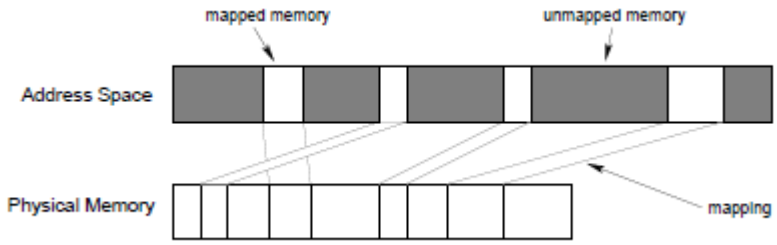

图6.1 虚拟内存与物理内存的映射关系

seL4内核没有内存申请器（allocator），所有内核对象都要在用户程序中显示申请。

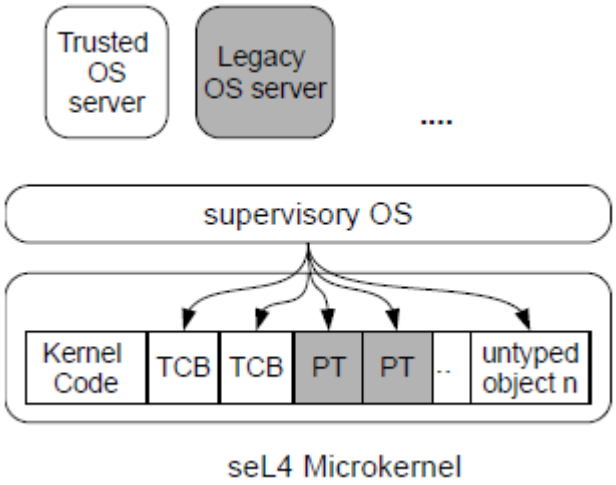

图6. 系统引导时的seL4内存布局

在操作系统这样的系统软件设计中，有一个可靠性的设计原则，就是下层软件不信任上层软件，而上层软件要无条件信任下层软件。那么，下层软件如何把自己的实现细节隐藏，不让上层软件通过“简单”方法可以直接操作下层软件的资源呢，句柄是一个常用的设计。

一个句柄指向一个内核对象，内核对象也只能通过与其绑定的句柄来操作它，无法直接访问。大多数的内核对象都有方法，用来操作这个对象的数据结构，就象是 c++中的封装中的 protect 数据结构，方法自身可以访问数据结构中的数据，但是，使用这个对象的其它程序，只能通过 API 方法与这个对象打交道。

举例来说，用户来以通过调用一个原始内存 untyped-memory 句柄的 invoke 方法，用来在这个原始内存上创建内核对象。

句柄存储在 CNodes 中，CNodes 本身也是一个内核对象。

有些内核对象只有方法，但自身没有什么状态（state）信息，这类对象的目的是为了让应用程序与内核打交道，这类对象没有自身的存储实例。

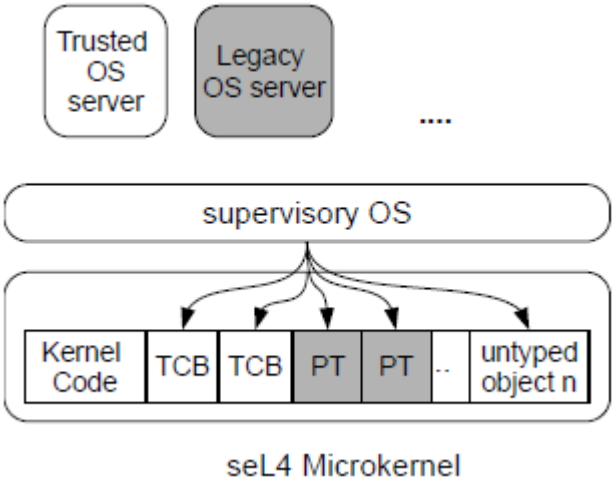

图6.内核服务被创建后的seL4内存布局

地址空间与虚拟内存Address Spaces and Virtual Memory。seL4中的一段虚拟地址空间，称为VSpace。象CSpace一样，VSpace是由内核提供的对象组成的，这些对象管理着对应的硬件的内存。页字典管理着页表，页表就是硬件MMU中说的那个页表，内核还包含ASID池（ASID Pool）和ASID控制（ASID Control）对象，用来跟踪地地址空间的使用情况。

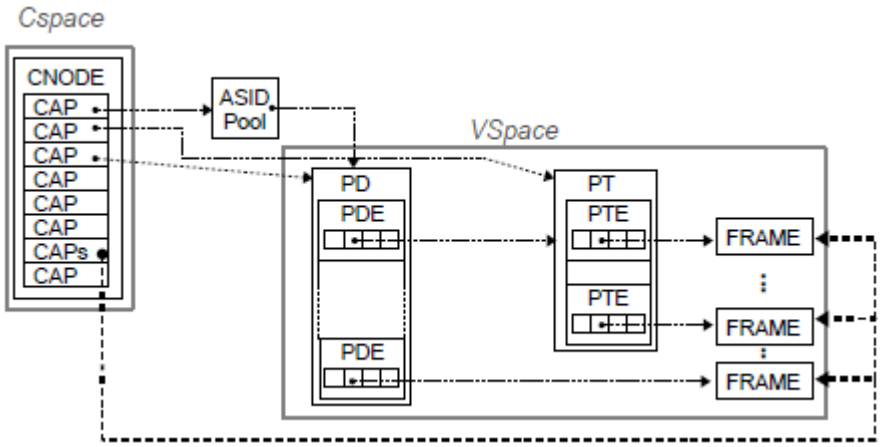

图6.1　seL4中的内存管理机制示意图

只是VSpace相关的对象就足够实现硬件相关的创建、操纵、删除虚拟地址空间等操作所需要的数据结构了。

这里要理解一个软件如何操作硬件的概念，其实软件指挥硬件，都是给硬件发数据，这些数据是硬件硬约定的，如页表、端口，软件想要了解硬件的状态，也只能是从相应的内存或端口读数据。这些情况下，硬件是老大，一切听人家的。

6.1 概述（Overview）

IA-32

IA-32架构有两级页表，最上层的页目录覆盖一个4GB的区间，每个页表覆盖4MB区间。帧可以是4KB或4MB。在4KM帧被影射前，

processors have a two-level page-table structure. The top-level page directory covers a 4 GiB range and each page table covers a 4 MiB range. Frames can be 4 KiB or 4 MiB. Before a 4KiB frame can be mapped, a page table covering the range that the frame will be mapped into must have been mapped, otherwise seL4 will return an error. 4 MiB frames are mapped directly into the page directory, thus, a page table does not need to be mapped rst.

ARM

ARM架构有两级页表，最上层的页目录覆盖一个4GB的区间，每个页表覆盖1MB区间。4个允许的页尺寸是: 4 KiB, 64 KiB, 1 MiB 和 16 MiB。4 KiB and 64 KiB pages are mapped into the second-level page table. Before they can be mapped, a page table covering the range that they will be mapped into must have been installed. 1 MiB and 16 MiB pages are installed directly into the page directory such that it is not necessary to map a page table rst. Pages of 4 KiB and 1 MiB size occupy one slot in a page table and the page directory, respectively. Pages of 64 KiB and 16 MiB size occupy 16 slots in a page table and the page directory, respectively.

6.2 内核对象（Objects）

Page Directory (PD)

页目录（Page Directory，PD）是两级页表结构的最上层页表。它的数据结构是硬件约定的，但概念上，大家是相通的。

页目录里面包含一定数量的页表入口（page directory entries，PDE)。页目录自身没有方法。它通常是访问其它虚拟内存内核对象时作为参数使用。

Page Table (PT)

页表（Page Table，PT）对象定义了页表的第二级，它包含一些槽（Slot)，每个槽包含一个页表项（page-table entry，PTE)。

页表对象有两个方法：

- seL4_ARM_PageTable_Map()
- seL4_IA32_PageTable_Map()

Takes a Page Directory capability as an argument, and installs a reference to the invoked Page Table in a speci ed slot in the Page Directory.

- seL4_ARM_PageTable_Unmap()
- seL4_IA32_PageTable_Unmap()

Removes the reference to the invoked Page Table from its containing Page Directory.

Page

一个页（Page）对象是一段物理内存区域，用来在虚拟地址空间实现虚拟地址。Page对象有下列方法：

- seL4_ARM_Page_Map()
- seL4_IA32_Page_Map()

拿页字典（Page Directory） 当成一个参数，把给定页加到页字典PD或页表PT的槽中。seL4源码中，它们都被宏定义为：seL4_ARCH_Page_Map()。

原型定义为：

```
static inline int
seL4_IA32_Page_Map(seL4_IA32_Page    service,    seL4_IA32_PageDirectory    pd,
seL4_Word vaddr, seL4_CapRights rights, seL4_IA32_VMAttributes attr)
```

- seL4_ARM_Page_Remap()
- seL4_IA32_Page_Remap()

改变一个已经存在的影射的权限。

- seL4_ARM_Page_Unmap()
- seL4_IA32_Page_Unmap()

Removes an existing mapping.

The virtual address for a Page mapping must be aligned to the size of the Page and must be mapped to a suitable Page Directory or Page Table. To map a page readable, the capability to the page that is being invoked must have read permissions. To map the page writable, the capability must have write permissions. The requested mapping permissions are speci ed with an argument of type seL4 CapRights given to the seL4 ARM Page Map() or seL4 IA32 Page Map() method. seL4 CanRead and seL4 CanWrite are the only valid permissions on both ARM and IA-32 architectures. If the capability does not have su cient permissions to authorise the given mapping,

then the mapping permissions are silently downgraded.

ASID Control

地址空间标识符控制。

Address Space ID，是一个CPU硬件上的术语，也有人把它说成Application Space ID，应用程序空间ID。

假如象在传统的Linux这样的操作系统中一样，你拿一个独立的地址空间就当成一个进程，并把它对应成一个应用，那么，一个系统能支持的应用数量是有限的。为了管理这个有限的资源，内核提供了地址空间标识符控制（ASID Control）句柄。这个句

柄用来生成一个句柄认证地址空间子集，这个被创建的句柄被称为：ASID Pool。

为什么说“一个系统能支持的应用数量是有限的”呢？
参考图ARMv6体系结构

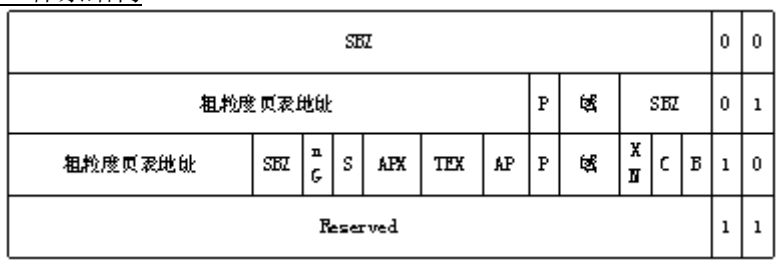

图6.2　ARMv6体系结构示意图

协处理器CP15中的XP-bit可以指定是否使用这种新的页表格式。如果不设置该位，则系统继续使用ARMv5架构的页表格式。

从图可以看出，页表格式中有以下特性：

- XN：从不执行位（execute never bit）。
- nG：非全局地址映射位（not Global bit for address matching）。

应用程序空间指示ASID（Application Space Identifier）是ARMv6体系中增加的又一关键特性。当nG位置位时，地址转换使用虚拟地址和ASID相结合的方法以减少上下文切换的时间。同时，应用程序空间指示提供了一种任务可知调试方法（task-aware debugging）。

ASID只有一个方法：

- seL4_ARM_ASIDControl_MakePool()
- seL4_IA32_ASIDControl_MakePool()

与指向原始内存Untyped Memory句柄一起，创建一个ASID池。
传给seL4 ARM ASIDControl MakePool()的原始内存句柄必须有4KB内存，足够空间容纳1024个VSpace。

ASID Pool

一个ASID Pool被授权为最多应用程序的子集。对于一个应用程序的VSpace，必须赋给ASID。这很好理解，VSpace是虚拟地址空间，ASID是应用程序标识，两者必须结合到一起来，MMU才能给你进行虚实转换。
ASID pool只有一个方法：

- seL4_ARM_ASIDPool_Assign()
- seL4_IA32_ASIDPool_Assign()

把一个与页目录关联的VSpace与一个ASID Pool结合起来，

6.3 映射属性（Mapping Attributes）

seL4_ARM_VMAttributes或seL4_IA32_VMAttributes属性用来指定页映射时，缓存的行为

表6.1： ARM 页表入口的虚拟内存属性

属性 Attribute	意思 Meaning
seL4_ARM_PageCacheable	Enable data in this mapping to be cached
seL4_ARM_ParityEnabled	Enable parity checking for this mapping

```
typedef enum {
    seL4_ARM_PageCacheable = 0x01,
    seL4_ARM_ParityEnabled = 0x02,
    seL4_ARM_Default_VMAttributes = 0x03,
    /* seL4_ARM_PageCacheable | seL4_ARM_ParityEnabled */
    SEL4_FORCE_LONG_ENUM(seL4_ARM_VMAttributes),
} seL4_ARM_VMAttributes;
```

6.4 共享内存（Sharing Memory）

seL4的内存共享是基于页（Page）的，也就是你哪怕想共享一个字节，也要把这个字节所在的页共享了。因为要共享的数据结构通常不会刚好是以页为单位的，所以，要对合适的页区域进行映射以实现共享。

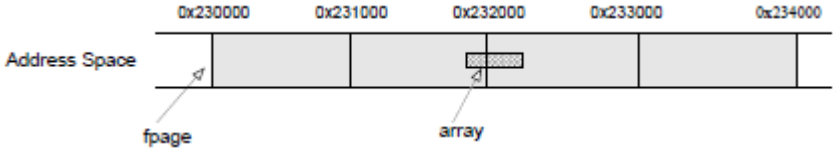

图6. 共享的页范围太大

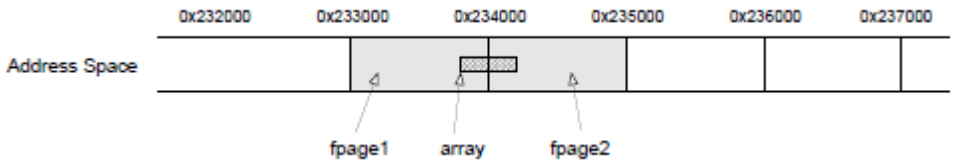

图6. 共享的页范围比较合适

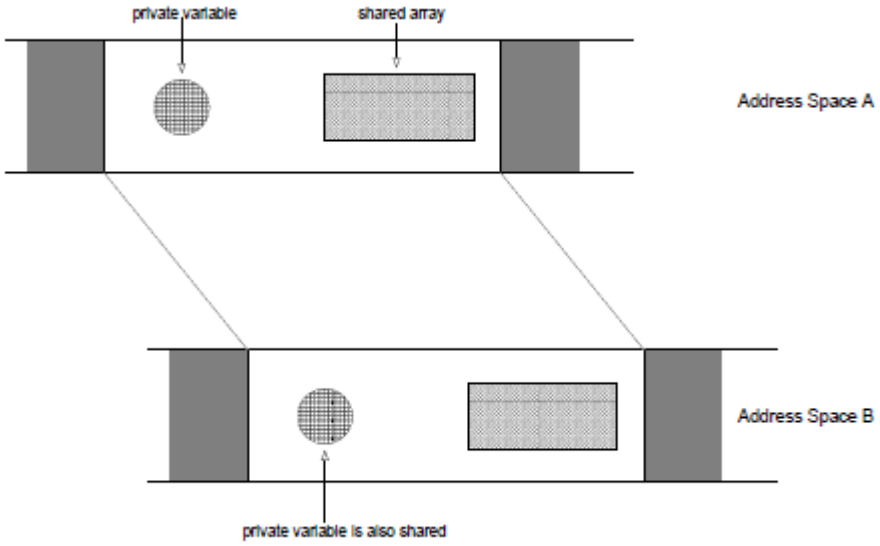

图6. 把不该共享数据结构也共享出去了，图中圆的数据结构是私有数据结构

seL4不允许页表（Page Table）共享，但允许页（page）共享。要想共享一页，指向页的句柄必须先要被复制，使用seL4_CNode_Copy()。

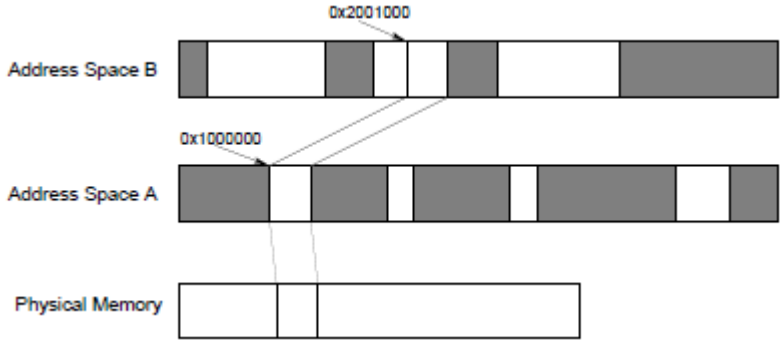

图6 两个虚拟地址空间共享一段物理内存

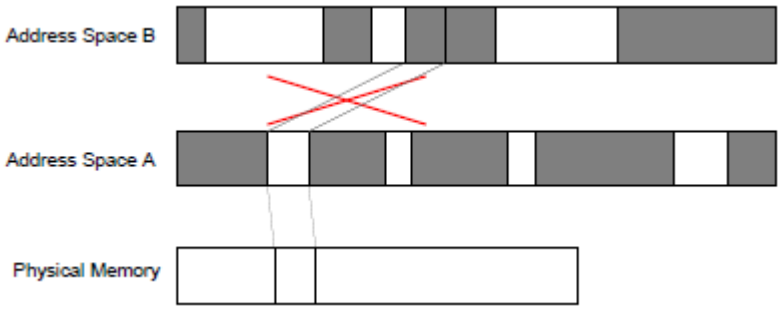

图6. 一段共享内存区间没有被映射（map）

表6.2: IA-32 页表入口的虚拟内存属性

属性 Attribute	意思 Meaning
seL4_IA32_CacheDisabled	Prevent data in this mapping from being cached
seL4_IA32_WriteThrough	Enable write through cacheing for this mapping
seL4_IA32_WriteCombining	Enable write combining for this mapping

```
typedef enum {
    seL4_IA32_Default_VMAttributes = 0,
    seL4_IA32_WriteBack = 0,
    seL4_IA32_WriteThrough = 1,
    seL4_IA32_CacheDisabled = 2,
    seL4_IA32_Uncacheable = 3,
    seL4_IA32_WriteCombining = 4,
    SEL4_FORCE_LONG_ENUM(seL4_IA32_VMAttributes),
} seL4_IA32_VMAttributes;
```

被复制得到的页的句柄，必须通过seL4_ARM_Page_Map()或seL4_IA32_Page_Map()加入到第二个地址空间。在一个地址空间内试图两次把同一页加入，将引起出错。

一个程序把自己的一页传给了你，你不把它加入你自己的页表中，显然是不能访问的。一个页，被N个程序加入了自己的页表，这就是共享内存。

6.5 页出错（Page Faults）

页出错的异常将被报告给执行程序的异常句柄，见§5.2.4。

7 硬件 IO

硬件 IO，Hardware I/O。

7.1 中断指派（Interrupt Delivery）

seL4 是一个微内核操作系统，它没有工作在内核态的驱动，它的所谓驱动（Driver）也是一个用户程序，甚至都不是一个完整的程序，只是别人程序的一部分。那这个用户态的程序，注意这里的用户态这个词，表明这些程序是工作在用户态的，不是工作在内核态（特权态），是响应物理的中断（处理中断请求，请求完后要给应答）的呢？

IRQHandler、IRQControl 这两个句柄登场了。它们就是普通的句柄 seL4_CPtr，定义上没啥特殊的。

```
typedef seL4_CPtr seL4_IRQHandler;
typedef seL4_CPtr seL4_IRQControl;
```

seL4 中，中断是通过异步端点（AsyncEP）对象指派的。一个线程可以告诉内核把一次中断请求传给它，由它负责处理。通报完后，线程就通过在对象 AsyncEP 上调用 seL4_Wait()等待中断的发生。消息中的比特位分别代表了不同中断。比特 n 代表中断 IRQ n。这种方式使得一个线程可以通过一个 AsyncEP 对象处理多个中断。

IRQHandler 句柄代表一个线程应答哪些中断，它有三个方法：

seL4_IRQHandler_SetEndpoint()
定义当中断发生时，内核应该通报notify()的AsyncEP。在这个句柄上seL4_Wait()，等待中断的发生。

seL4_IRQHandler_Ack()
通报内核，用户态中的驱动程序结束了中断的处理，可以让内核继续向其发送被挂起的其它中断了。

seL4_IRQHandler_Clear()
取消IRQHandler对象上注册的AsyncEP。

当系统刚起动时，所有中断全部悬空，没人处理。也就是没有 IRQHandler 句柄存在。系统导时，初始化线程的 CSpace 中包含一个 IRQControl 句柄，这个句柄可以及用来生成响应不同中断的不同 IRQHandler 句柄。

一般的做法是，初始化线程决定哪些中断由哪些程序处理，并生成好 IRQControl，传给相应的程序。

IRQControl 有一个方法：

seL4_IRQControl_Get()

为指定的中断创建一个IRQHandler句柄。

7.2 IA-3 特定 I/O

7.2.1 I/O 端口（I/O Ports）

IA-32 平台才有端口的说法，所谓端口，就是控制硬件的寄存器，它有自己独立的地址空间，就叫端口，如果它的地址空间与内存的地址空间是一样的，并且占用一段物理地址空间，它就叫 I/O 空间。

IA-32 平台上，seL4 允许在用户态程序中读、写端口，句柄是 IO Port，每个 IO port 代表一段端口区间。

读 I/O 端口的有三个函数：seL4_IA32_IOPort_In8()、seL4_IA32_IOPort_In16()和 seL4_IA32_IOPort_In32()。看名字就能看出来，就是从端口中读出 1 个字节（8 bit）、2 个字节（16 bit）、4 个字节（32_bit）的内容。

写 I/O 端口的有三个函数：seL4_IA32_IOPort_Out8()、seL4_IA32_IOPort_Out16()和 seL4_IA32_IOPort_Out32()。看名字就能看出来，就是往端口中写入 1 个字节（8 bit）、2 个字节（16 bit）、4 个字节（32_bit）的内容。

每个函数除 IO Port 这个参数外，还有一个 port 参数，代表具体端口，具体端口显然必须在 IO Port 中约定的 IO 地址范围。

系统初始化时，初始化线程的 CSpace 中包含主要 IO Port 句柄，可以通过它访问所有 I/O 端口。其它 IO Port 句柄，也就是访问其它端口的句柄，通过对这个句柄做 seL4_CNode_Mint()而得。新创建句柄的 I/O 端口通过传递给 seL4_CNode_Mint()的参数，32 比特的标记 badge 提供。第一个端口地址占 badge 的高 16 位，最后一个端口的地址占 badge 的低 16 位。合理的地址区间就是这两个地址之间的地址。

I/O 端口方法出错返回错误码。如果访问的端口地址超过 IO port 句柄约定的地址区间（创建这个句柄时指定的，上面不是提到过吗？），则返回 seL4_IllegalOperation。

因为读 I/O 端口要返回两个无符号整数值，一个是读出的内容，一个是错误码，所以这些函数返回的是一个 c++的结构体：

```
/*
 * Return types for generated methods.
 */
struct seL4_IA32_Page_GetAddress {
    int error;
    seL4_Word paddr;
};
typedef struct seL4_IA32_Page_GetAddress seL4_IA32_Page_GetAddress_t;
```

```
struct seL4_IA32_IOPort_In8 {
    int error;
    uint8_t result;
};
typedef struct seL4_IA32_IOPort_In8 seL4_IA32_IOPort_In8_t;

struct seL4_IA32_IOPort_In16 {
    int error;
    uint16_t result;
};
typedef struct seL4_IA32_IOPort_In16 seL4_IA32_IOPort_In16_t;

struct seL4_IA32_IOPort_In32 {
    int error;
    uint32_t result;
};
typedef struct seL4_IA32_IOPort_In32 seL4_IA32_IOPort_In32_t;
```

7.2.2 I/O 空间（I/O Space）

I/O devices capable of DMA present a security risk because the CPU's MMU is by-passed when the device accesses memory. In seL4, device drivers run in user space to keep them out of the trusted computing base. A malicious or buggy device driver may, however, program the device to access or corrupt memory that is not part of its address space, thus subverting security. To mitigate this threat, seL4 provides support for the IOMMU on Intel IA-32-based platforms. An IOMMU allows memory to be remapped from the device's point of view. It acts as an MMU for the device, restricting the regions of system memory that it can access. More information can be obtained from Intel's IOMMU documentation [Int11].

seL4-based systems that wish to utilise DMA must have an IOMMU. This restriction results from the fact that seL4 provides no way to obtain the physical address of a Page from its capability. Hence, applications are unable to accurately instruct devices, at which address they should directly address the physical memory. Instead, frames of memory must be mapped into the device's address space using seL4's IOMMU primitives.

Two new objects are provided by the kernel to abstract the IOMMU:

IOSpace This object represents the address space associated with a hardware device on the PCI bus. It represents the right to modify a device's memory mappings.

IOPageTable This object represents a node in the multilevel page-table structure used by IOMMU hardware to translate hardware memory accesses.

Page capabilities are used to represent the actual frames that are mapped into the I/O address space. A Page can be mapped into either a VSpace or an IOSpace but never into both at the same time.

IOSpace and VSpace fault handling di er signi cantly. VSpace page faults are redirected to the thread's exception handler (see Section 5.2), which can take the appropriate action and restart the thread at the faulting instruction. There is no concept of an exception handler for an IOSpace. Instead, faulting transactions are simply aborted; the device driver must correct the cause of the fault and retry the DMA transaction.

An initial master IOSpace capability is provided in the initial thread's CSpace. An IOSpace capability for a speci c device is created by using the seL4 CNode Mint() method, passing the PCI identi er of the device as the low 16 bits of the badge ar-gument, and a Domain ID as the high 16 bits of the badge argument. PCI identi ers are explained fully in the PCI speci cation [SA99], but are brie y described here. A PCI identi er is a 16-bit quantity. The rst 8 bits identify the bus that the device is on. The next 5 bits are the device identi er: the number of the device on the bus. The last 3 bits are the function number. A single device may consist of several inde-pendent functions, each of which may be addressed by the PCI identi er. Domain IDs are explained fully in the Intel IOMMU documentation [Int11]. There is presently no way to query seL4 for how many Domain IDs are supported by the IOMMU and the seL4 CNode Mint() method will fail if an unsupported value is chosen.

The IOMMU page-table structure has three levels. Page tables are mapped into an IOSpace using the seL4 IA32 IOPageTable Map() method. This method takes the IOPageTable to map, the IOSpace to map into and the address to map at. Three levels of page tables must be mapped before a frame can be mapped successfully. A frame is mapped with the seL4 IA32 Page MapIO() method whose parameters are analogous to the corresponding method that maps Pages into VSpaces (see Chapter 6), namely seL4_IA32_Page_Map().

Unmapping is accomplished with the usual unmap (see Chapter 6) API call, seL4_- IA32 Page Unmap().

More information about seL4's IOMMU abstractions can be found in [Pal09].

8 系统启动过程

系统启动过程，或称系统引导过程，就是从加电开始，一直到系统就续的过程，System Bootstrapping。

本章步及到的数据结构，定义在：

c++头文件：/libs/libsel4/include/sel4/bootinfo.h

8.1 初始线程环境（Initial Thread's Environment）

一个电子系统，只要是加了电，它的 CPU 就一直跑指令，也就是从内存里拿一条指令，在执行单元执行，然后再拿下一条指令，如此往复。所谓内核，所谓用户态程序，不过是当前 CPU 跑在不同态（ring）中的一个抽象说法而已。也就是一上来，这个电子设备只有指令，是你的这些指令，把有此东西（用户程序之类）摆在了用户态（ring3），把有些东西（内核）摆在了内核态（ring0），等等等等，内存里的世界摆好了，开始跳到你的第一个用户程序的第一个线程，让它开运转起来就是了。

你觉得相对于电子设备里的程序，你是不是有点象上帝？

seL4 为初始线程（initial thread）创建一个最小引导环境（boot environment）。这个环境中包含：

- TCB，你的初始线程的控制块，Thread Control Block
- CSpace，你让内核为你工作，内核自己又不管理动态内存，于是，只好你自己来管。
  这个 CSpace 初始时包含可操控系统全局资源的所有句柄。
  CNode 的尺寸可以在系统编译时配置，缺省是 2^12 个槽。但通常要选择“守卫（guard)”，以使得占足 32 比特。2^12 个槽，用 12 比特位可以表示一个句柄，那不足 32 位的咋办？填充呗。那些填充位，再配上点用处、说法，就叫守卫。这样设置后，第一个槽的 CPTR 就是 0x0，第二个槽的 CPTR 就是 0x1，以此类推。
- VSpace，虚拟地起址空间，你通过 seL4 的内存对象使用虚拟地址空间，这个线性地址空间，还没有象 32 位 Linux 那样约定了好多（比如用户态空间占低内存 3GB，内核空间占高内存 1GB 等）。
- 一块包含系统起动影像（userland image，code/data of the initial thread）的帧（frame）
- IPC buffer

表 8.1 初始化线程的 CNode 内容

CPTR	常量 Enum Constant	句柄 Capability
0x0	seL4_CapNull	null
0x1	seL4_CapInitThreadTCB	initial thread's TCB
0x2	seL4_CapInitThreadCNode	initial thread's CNode
0x3	seL4_CapInitThreadPD	初始线程目录词典 initial thread's page directory
0x4	seL4_CapIRQControl	全局中断控制器global IRQ controller (见§7.1)
0x5	seL4_CapASIDControl	全局地址空间标识符控制器global ASID controller (见§6)
0x6	seL4_CapInitThreadASIDPool	初始线程的地址空间标识符池initial thread's ASID pool(见§6)
0x7	seL4_CapIOPort	全局IO端口global I/O port cap, null cap if unsup- ported (见§7.2.1)
0x8	seL4_CapIOSpace	全局IO地址空间 global I/O space cap, null cap if unsup- ported (见§7.2.2)
0x9	seL4_CapBootInfoFrame	起动影像BootInfo frame (§8.2)
0xa	seL4_CapInitThreadIPCBuffer	initial thread's IPC bu er (§4.1)
0xb	seL4_CapDom	domain cap (§5.3)

bootinfo.h 中的一个定义：

```
/* caps with fixed slot potitions in the root CNode */

enum {
    seL4_CapNull                =  0, /* null cap */
    seL4_CapInitThreadTCB       =  1, /* initial thread's TCB cap */
    seL4_CapInitThreadCNode     =  2, /* initial thread's root CNode cap */
    seL4_CapInitThreadPD        =  3, /* initial thread's PD cap */
    seL4_CapIRQControl          =  4, /* global IRQ controller cap */
    seL4_CapASIDControl         =  5, /* global ASID controller cap */
    seL4_CapInitThreadASIDPool  =  6, /* initial thread's ASID pool cap */
    seL4_CapIOPort               =  7, /* global IO port cap (null cap if not
supported) */
    seL4_CapIOSpace             =  8, /* global IO space cap (null cap if no IOMMU
support) */
    seL4_CapBootInfoFrame       =  9, /* bootinfo frame cap */
    seL4_CapInitThreadIPCBuffer = 10, /* initial thread's IPC buffer frame cap
*/
    seL4_CapDomain              = 11  /* global domain controller cap */
};
```

8.2 起动信息帧（BootInfo Frame）

CNode 槽 CPTR 0xb 及之上的是在系统起动的过程中自动填上去的，它们的确切的值依赖于启动操作系统影像（就是你编 seL4 得到的那个影像文件）的大小、平台配置、设备配置等。

为了告诉启动线程它的启动环境，哪些句柄都存在哪里，内核为启动线程提供了起动信息帧 BootInfo Frame，并把它映射到起动线程的地址空间。

要知道，这个起动线程才是引起系统干活的第一个程序，内核在做事情上，时时处于被动状态，微内核操作系统又没有驱动，你就是让系统一切的工作只是被动服务（听中断啊），都没有把中断程序给你加载并把中断向量给你设上的。

起动信息帧放在内核的哪里由内核决定，并把起始地址（这个地址是启动线程认识的地址）通过一个寄存器传给启动线程。启动线程通过函数 seL4_GetBootInfo()得到这个起动信息帧。

起动信息帧的数据结构见表 8.2，这个描述采用 c++结构类型，这个数据结构在 seL4 开发环境中有定义。除了上面提到的句柄之类的内容，这个数据结构里还有系统的其它配置信息。

seL4_SlotRegion 是一个 c++结构类类型，它里面有开始、结束 CPTR。它用来表示一段在启始线程的 CNode 中的一个槽（Slot）区间。

```
typedef struct {
    seL4_Word start; /* first CNode slot position OF region */
    seL4_Word end;   /* first CNode slot position AFTER region */
} seL4_SlotRegion;
```

表 8.2: BootInfo Struct

Field Type	Field Name	Description
seL4_Word	nodeID	node ID (see Section 8.4)
seL4_Word	numNodes	number of nodes (see Sec- tion 8.4)
seL4_Word	numIOPTLevels	number of I/O page-table levels (0 if no IOMMU)
seL4_IPCBuffer*	ipcBuffer	指向初始线程的IPC缓冲区 pointer to the initial thread's IPC buffer
seL4_SlotRegion	empty	empty slots (null caps)
seL4_SlotRegion	sharedFrames	see Section 8.4
seL4 SlotRegion	userImageFrames	frames containing the user-land image
seL4_SlotRegion	userImagePTs	page tables covering the userland image
seL4_SlotRegion	untyped	untyped-memory capabili-ties
seL4_Word[]	untypedPaddrList	array of untyped-memory physical addresses
uint8_t[]	untypedSizeBitsList	array of untyped-memory sizes (2^n bytes)

uint8		t	initThreadCNodeSizeBits	CNode size (2^n slots)
seL4_Word			numDeviceRegions	number of device memory regions
seL4_DeviceRegion[]			deviceRegions	device memory regions (see Table 8.3)

userImageFrame和userImagePT中的句柄是有序的，也就是说，第一个句柄对应启动影像的第一帧，如此类推。用户程序永远知道它自己的代码和数据被映射到虚拟地址的哪个空间了，这样，通过GCC编译，并通过标准链接角本链接的程序，它的_executable_start和_end是可用的。应用程序知道每个用户帧和页表句柄后面的程序是谁。

这里给出的原始内存是无序的。第 i 个原始内存句柄所表示的原始内存的大小存在于 untypedSizeBitsList[i]中（2^n 字节）。所以，数组的大小是：untyped.end - untyped.start。这个长度是硬编码的，对应用程序来说不可改变。同样的情况也适用于 untypedPaddrList。对每个原始内存句柄，它存储了它的物理内存，这将使得当有 IOMMU 硬件时，用户程序可以用原始内存句柄初始化 DMA 传输，因为这时的操作对象只能是物理内存。

内核不监视物理内存的使用，不管理你应用程序抢内存，把内存还剩多少，除非你在编译时指定了 4K 原始内存的最小数量，缺省是 12。

表8.3 DeviceRegion数据结构

Field Type	Field Name	Description
seL4_Word	basePaddr	physical base address of the device region
seL4_Word	frameSizeBits	size (2^n bytes) of the frames used
seL4_SlotRegion	frames	capabilities to the frames covering the region

内核针对内存映射设备管理的每个物理内存区域都创建了帧。这些设备不只是嵌入式硬编码的设备，也包括内核启动时通过PCI扫描检测到的设备。

每个设备的基地址存储在basePaddr中，frames数组变量中存的是所有用来访问这些存储的句柄，这些句柄是有序的，数组的第一个元素（即第一个句柄）对应的物理地址就是basePaddr，所有怕区域大小固定为：2^frameSizeBits。

所以，整个区域的大小为：

(frames.end - frames.start) << frameSizeBits

BootInfo中的数组deviceRegions存有所有可以备区域，设备数量为：numDeviceRegions

如果平台有一个seL4支持的IOMMU，numIOPTLevels包含IOMMU-page-table levels。这个信息在用户程序构造IOMMU地址空间（IOSpace）时有用。如果平台没有IOMMU支持，numIOPTLevels被设为0。

8.3 引导用令行（Boot Command-line Arguments）

IA-32平台上，multiboot兼容的bootloader（如：GRUB、syslinux）引导seL4时，可以把命令行传给内核。多个参数通过空白隔开。支持两种类型的命令行：

- key-value形式，如key=value，value可以是字符串、整数0x开头的十六进制数，或者用逗号分隔开的列表。等号前后，逗号前后不可以有空白。
- key形式，即只有key

key和value中间不能有空白。

表 8.4: IA-32 boot命令行参数

Key	Value	Default
控制台端口 console port	I/O基地址，如果内核有输出，将输出到这个串行口中。 内核在debug模式有调试信息输出 I/O-port base of the serial port that the kernel prints to (if compiled in debug mode)	0x3f8
调试端口 debug port	I/O-port base of the serial port that is used for kernel debugging (if compiled in debug mode)	0x3f8
禁止IOMMU disable_iommu	none	The IOMMU is enabled by default on VT-d-capable plat-forms
最大结点数 maxnum nodes	Maximum number of seL4 nodes that can be started up (see Section 8.4)	1
最大共享帧数 num_sh_frames	Number of frames shared be- tween seL4 nodes (see Sec- tion 8.4)	0

8.4 多核起动（Multikernel Bootstrapping）

ARM平台现在不支持多核，此时BootInfo的nodeID值永远为0，numNodes为1，sharedFrames是一个空的区域。

IA-32平台，seL4支持多核，多核起动时，通过命令行传送max_num_nodes大于1，每个可用CPU核将运行一份独立的seL4结点。可用物理内存平均分区给各个结点，所有结点都能得到物理设备，得到device frame。初始化线程必须协调这些设备帧，规定哪个设备归哪个结点响应。IOMMU只能被初始线程操作，其它结点没有全局的IOSpace句柄。

支持的最多硬件核数缺省为8，可以被调整到256.

除了共享帧（shared frames）结点间互相独立。当系统启动时，内核创建一些4KB大小的共享帧，这些帧所有结点都可以访问。共享帧的数量可以通过命令行num_sh_frames来规定。

所谓共享帧所有结点都可以访问，那当然是指这些结点的初始化线程可以见到它们。在BootInfo数据结构中，sharedFrames存储了这些槽区域。

应用程序通过共享帧共享数据结构、消息传输、同步机制等。共享帧的句柄同样支持铀柄的锻造（Mint）等操作，于是应用程序就可以做更细粒度的一些事了。

每个结点有自己的唯一标识ID，BootInfo中的nodeID，BootInfo中的numNodes是结点数量。

multiboot兼容的bootloader（如：GRUB、syslinux）引导seL4时，用户空间的程序影像通过boot modules交给内核。每个启动模块包含一个ELF文件。如果只有一个程序，则每个结得到一份它们自己的影像复本。如果系统起动影像中包含多个程序，则第一个结点得到第一个程序影像，第二个程序得到第二个，以此类推，如果程序影象不够结点们分，比如你有8个结点，但只3个影像，则最后一个程序影像将被重复使用。

如果内核是debug模式，则每个结点可以被赋予独立的监控串口和调试串口（见表8.4）。方法就是传入consoe_port一个串口基地址列表，如下例：

console port=0x3f8,0x2f8,0x3e8,0x2e8

将把端口0x3f8 给 结点0，端口0x2f8给 结点1，余此类推。余下的没赋给端口的结点，没控制台输出。

同理适用于调试端口debug_port。

9 编程接口 API

9.1 出错代码

Error Codes

Invoking a capability with invalid parameters will result in an error. seL4 system calls return an error code in the message tag and a short error description in the message registers to aid the programmer in determining the cause of errors.

9.1.1 Invalid Argument

A non-capability argument is invalid.

Field	Meaning
Label	seL4_InvalidArgument
IPCBuffer[0]	Invalid argument number

9.1.2 Invalid Capability

A capability argument is invalid.

Field	Meaning
Label	seL4_InvalidCapability
IPCBuffer[0]	Invalid capability argument number

9.1.3 Illegal Operation

The requested operation is not permitted.

Field	Meaning
Label	seL4 IllegalOperation

9.1.4 Range Error

An argument is out of the allowed range.

Field	Meaning
Label	seL4_RangeError
IPCBuffer[0]	Minimum allowed value
IPCBuffer[1]	Maximum allowed value

9.1.5 Alignment Error

A supplied argument does not meet the alignment requirements.

Field	Meaning
Label	seL4_AlignmentError

9.1.6 Failed Lookup

A capability could not be looked up.

Field	Meaning
Label	seL4_FailedLookup
IPCBuffer[0]	1 if the lookup failed for a source capability, 0 otherwise
IPCBuffer[1]	Type of lookup failure
IPCBuffer[2..]	Lookup failure description as described in Section 3.4

9.1.7 Delete First

A destination slot speci ed in the syscall arguments is occupied.

Field	Meaning
Label	seL4_DeleteFirst

9.1.8 Revoke First

The object currently has other objects derived from it and the requested invocation cannot be performed until either these objects are deleted or the revoke invocation is performed on the capability.

Field	Meaning
Label	seL4_RevokeFirst

9.1.9 Not Enough Memory

The Untyped Memory object does not have enough unallocated space to complete the seL4_Untyped_Retype() request.

Field	Meaning
Label	seL4_NotEnoughMemory
IPCBuffer[0]	Amount of memory available in bytes

9.2 基础系统 API

这些 API 由内核直接提供。
定义在 syscall.xml 中的系统调用

```
<syscalls>
    <!-- official API syscalls -->
    <api>
        <config>
            <syscall name="Call"      />
            <syscall name="ReplyWait" />
            <syscall name="Send"      />
```

```
            <syscall name="NBSend"    />
            <syscall name="Wait"      />
            <syscall name="Reply"     />
            <syscall name="Yield"     />
        </config>
    </api>
    <!-- Syscalls on the unknown syscall path. These definitions will be wrapped
in #ifdef name -->
    <debug>
        <config name="DEBUG">
            <syscall name="DebugPutChar"  />
            <syscall name="DebugHalt"     />
            <syscall name="DebugCapIdentify"   />
            <syscall name="DebugSnapshot" />
        </config>
        <config name="DANGEROUS_CODE_INJECTION">
            <syscall name="DebugRun"/>
        </config>
        <config name="CONFIG_BENCHMARK">
            <syscall name="BenchmarkResetLog" />
            <syscall name="BenchmarkDumpLog"  />
            <syscall name="BenchmarkLogSize"  />
        </config>
    </debug>
</syscalls>
```

seL4_SysCall
System Calls

9.2.1 Send

原型：

static inline void seL4_Send

```
static inline void
seL4_Send(seL4_CPtr dest, seL4_MessageInfo_t msgInfo)
```

功能：

向一个句柄发送一个消息，阻塞。

参数说明：

Type	Name	Description
seL4_CPtr	dest	要调用的句柄
seL4_MessageInfo_t	msgInfo	IPC消息结构

返回值：

本函数无返回值。

9.2.2 Wait

原型：

static inline seL4_MessageInfo_t seL4_Wait

```
static inline seL4_MessageInfo_t
seL4_Wait(seL4_CPtr src, seL4_Word* sender)
```

功能：

等待别人的应答。

参数说明：

Type	Name	Description
seL4_CPtr	src	要调用的句柄
seL4_Word*	sender	端点句柄的标记badge，通过这个地址接收其内容；如果为NULL，表示不接收消息的标记

返回值：

一个 seL4_MessageInfo_t 结构。

9.2.3 Call

原型：

static inline seL4_MessageInfo_t seL4_Call

```
static inline seL4_MessageInfo_t
seL4_Call(seL4_CPtr dest, seL4_MessageInfo_t msgInfo)
```

功能：

调用一个句柄。

参数说明：

Type	Name	Description
seL4_CPtr	dest	要调用的句柄
seL4_MessageInfo_t	msgInfo	IPC消息结构

返回值：

一个 seL4_MessageInfo_t 结构。

程序示例：

```
static void
signal_helper_finished(seL4_CPtr local_endpoint, int val)
{
    seL4_MessageInfo_t info = seL4_MessageInfo_new(0, 0, 0, 1);
```

```
    seL4_SetMR(0, val);
    seL4_Call(local_endpoint, info);
}
```

9.2.4 Reply

原型：

static inline void seL4_Reply

```
static inline void
seL4_Reply(seL4_MessageInfo_t msgInfo)
```

功能：

发一个应答。应答句柄存在于线程最后一次收到的同步消息。

参数说明：

Type	Name	Description
seL4_MessageInfo_t	msgInfo	IPC消息结构

返回值：

本函数无返回值。

9.2.5 Non-blocking Send

原型：

static inline seL4_MessageInfo_t seL4_NBSend

```
static inline void
seL4_NBSend(seL4_CPtr dest, seL4_MessageInfo_t msgInfo)
```

功能：

向一个句柄发送一个消息，非阻塞。

参数说明：

Type	Name	Description
seL4_CPtr	dest	要调用的句柄
seL4_MessageInfo_t	msgInfo	IPC消息结构

返回值：

本函数无返回值。

9.2.6 Reply Wait

原型：

static inline seL4_MessageInfo_t seL4_ReplyWait

```
static inline seL4_MessageInfo_t
seL4_ReplyWait(seL4_CPtr src, seL4_MessageInfo_t msgInfo, seL4_Word *sender)
```

功能：

在一个系统调用内，发一个应答并等待别人的应答。

参数说明：

Type	Name	Description
seL4_CPtr	src	要调用的句柄
seL4_MessageInfo_t	msgInfo	IPC消息结构
seL4_Word*	sender	端点句柄的标记badge，通过这个地址接收其内容；如果为NULL，表示不接收消息的标记

返回值：

一个 seL4_MessageInfo_t 结构。

程序示例：

```
static int
bouncer_func(seL4_CPtr ep, seL4_Word arg1, seL4_Word arg2, seL4_Word arg3)
{
    seL4_MessageInfo_t tag = seL4_MessageInfo_new(0, 0, 0, 0);
    seL4_Word sender_badge;
    seL4_Wait(ep, &sender_badge);
    while (1) {
        seL4_ReplyWait(ep, tag, &sender_badge);
    }
    return 0;
}
```

9.2.7 Yield

原型：

static inline void seL4_Yield

```
static inline void
seL4_Yield(void)
```

功能：

放弃余下的 CPU 时间给具有相同优先级的线程。

返回值：

本函数无返回值

9.2.8 Notify

原型：

static inline void seL4_Notify

```
static inline void
seL4_Notify(seL4_CPtr dest, seL4_Word msg)
```

功能：

发送一个字的消息。

参数说明：

Type	Name	Description
seL4_CPtr	dest	要调用的句柄
seL4_Word	msg	要发送的一个单字

返回值：

本函数无返回值

说明：

从下面 seL4_Notify()的源码能看出，seL4_Notify()不是一个正常的内核提供的系统调用，它是用户态程序库对 seL4_Send 的一个包装。它对向一个异步端点发消息有用。

它的实现与平台相关，是用汇编写的。

```
static inline void
seL4_Notify(seL4_CPtr dest, seL4_Word msg)
{
    register seL4_Word destptr asm("r0") = (seL4_Word)dest;
    register seL4_Word info asm("r1") = seL4_MessageInfo_new(0, 0, 0, 1).words[0];
    register seL4_Word msg0 asm("r2") = msg;

    /* Perform the system call. */
    register seL4_Word scno asm("r7") = seL4_SysSend;
    asm volatile ("swi %[swi_num]"
                  : "+r" (destptr), "+r" (msg0), "+r" (info)
                  : [swi_num] "i" __SWINUM(seL4_SysSend), "r"(scno)
                  : "memory");
}
```

9.3 体系结构无关的对象方法

体系结构无关的对象方法 Architecture-Independent Object Methods ，这些方法通过内核对象的方式提供。

seL4.xml 中定义的 API

```
<api>
```

```
    <interface name="seL4_Untyped">
        <method id="UntypedRetype" name="Retype">
            <param dir="in" name="type" type="int"/>
            <param dir="in" name="size_bits" type="int"/>
            <param dir="in" name="root" type="seL4_CNode"/>
            <param dir="in" name="node_index" type="int"/>
            <param dir="in" name="node_depth" type="int"/>
            <param dir="in" name="node_offset" type="int"/>
            <param dir="in" name="num_objects" type="int"/>
        </method>
    </interface>
    <interface name="seL4_TCB">
        <method id="TCBReadRegisters" name="ReadRegisters">
            <param dir="in" name="suspend_source" type="bool"/>
            <param dir="in" name="arch_flags" type="uint8_t"/>
            <param dir="in" name="count" type="seL4_Word"/>
            <param dir="out" name="regs" type="seL4_UserContext"/>
        </method>
        <method id="TCBWriteRegisters" name="WriteRegisters">
            <param dir="in" name="resume_target" type="bool"/>
            <param dir="in" name="arch_flags" type="uint8_t"/>
            <param dir="in" name="count" type="seL4_Word"/>
            <param dir="in" name="regs" type="seL4_UserContext"/>
        </method>
        <method id="TCBCopyRegisters" name="CopyRegisters">
            <param dir="in" name="source" type="seL4_TCB"/>
            <param dir="in" name="suspend_source" type="bool"/>
            <param dir="in" name="resume_target" type="bool"/>
            <param dir="in" name="transfer_frame" type="bool"/>
            <param dir="in" name="transfer_integer" type="bool"/>
            <param dir="in" name="arch_flags" type="uint8_t"/>
        </method>
        <method id="TCBConfigure" name="Configure">
            <param dir="in" name="fault_ep" type="seL4_Word"/>
            <param dir="in" name="priority" type="uint8_t"/>
            <param dir="in" name="cspace_root" type="seL4_CNode"/>
            <param dir="in" name="cspace_root_data" type="seL4_CapData_t"/>
            <param dir="in" name="vspace_root" type="seL4_CNode"/>
            <param dir="in" name="vspace_root_data" type="seL4_CapData_t"/>
            <param dir="in" name="buffer" type="seL4_Word"/>
            <param dir="in" name="bufferFrame" type="seL4_CPtr"/>
        </method>
        <method id="TCBSetPriority" name="SetPriority">
            <param dir="in" name="priority" type="uint8_t"/>
        </method>
        <method id="TCBSetIPCBuffer" name="SetIPCBuffer">
            <param dir="in" name="buffer" type="seL4_Word"/>
            <param dir="in" name="bufferFrame" type="seL4_CPtr"/>
        </method>
        <method id="TCBSetSpace" name="SetSpace">
            <param dir="in" name="fault_ep" type="seL4_Word"/>
            <param dir="in" name="cspace_root" type="seL4_CNode"/>
```

```
            <param dir="in" name="cspace_root_data" type="seL4_CapData_t"/>
            <param dir="in" name="vspace_root" type="seL4_CNode"/>
            <param dir="in" name="vspace_root_data" type="seL4_CapData_t"/>
        </method>
        <method id="TCBSuspend" name="Suspend"/>
        <method id="TCBResume" name="Resume"/>
    </interface>
    <interface name="seL4_CNode">
        <method id="CNodeRevoke" name="Revoke">
            <param dir="in" name="index" type="seL4_Word"/>
            <param dir="in" name="depth" type="uint8_t"/>
        </method>
        <method id="CNodeDelete" name="Delete">
            <param dir="in" name="index" type="seL4_Word"/>
            <param dir="in" name="depth" type="uint8_t"/>
        </method>
        <method id="CNodeRecycle" name="Recycle">
            <param dir="in" name="index" type="seL4_Word"/>
            <param dir="in" name="depth" type="uint8_t"/>
        </method>
        <method id="CNodeCopy" name="Copy">
            <param dir="in" name="dest_index" type="seL4_Word"/>
            <param dir="in" name="dest_depth" type="uint8_t"/>
            <param dir="in" name="src_root" type="seL4_CNode"/>
            <param dir="in" name="src_index" type="seL4_Word"/>
            <param dir="in" name="src_depth" type="uint8_t"/>
            <param dir="in" name="rights" type="seL4_CapRights"/>
        </method>
        <method id="CNodeMint" name="Mint">
            <param dir="in" name="dest_index" type="seL4_Word"/>
            <param dir="in" name="dest_depth" type="uint8_t"/>
            <param dir="in" name="src_root" type="seL4_CNode"/>
            <param dir="in" name="src_index" type="seL4_Word"/>
            <param dir="in" name="src_depth" type="uint8_t"/>
            <param dir="in" name="rights" type="seL4_CapRights"/>
            <param dir="in" name="badge" type="seL4_CapData_t"/>
        </method>
        <method id="CNodeMove" name="Move">
            <param dir="in" name="dest_index" type="seL4_Word"/>
            <param dir="in" name="dest_depth" type="uint8_t"/>
            <param dir="in" name="src_root" type="seL4_CNode"/>
            <param dir="in" name="src_index" type="seL4_Word"/>
            <param dir="in" name="src_depth" type="uint8_t"/>
        </method>
        <method id="CNodeMutate" name="Mutate">
            <param dir="in" name="dest_index" type="seL4_Word"/>
            <param dir="in" name="dest_depth" type="uint8_t"/>
            <param dir="in" name="src_root" type="seL4_CNode"/>
            <param dir="in" name="src_index" type="seL4_Word"/>
            <param dir="in" name="src_depth" type="uint8_t"/>
            <param dir="in" name="badge" type="seL4_CapData_t"/>
        </method>
```

```
        <method id="CNodeRotate" name="Rotate">
            <param dir="in" name="dest_index" type="seL4_Word"/>
            <param dir="in" name="dest_depth" type="uint8_t"/>
            <param dir="in" name="dest_badge" type="seL4_CapData_t"/>
            <param dir="in" name="pivot_root" type="seL4_CNode"/>
            <param dir="in" name="pivot_index" type="seL4_Word"/>
            <param dir="in" name="pivot_depth" type="uint8_t"/>
            <param dir="in" name="pivot_badge" type="seL4_CapData_t"/>
            <param dir="in" name="src_root" type="seL4_CNode"/>
            <param dir="in" name="src_index" type="seL4_Word"/>
            <param dir="in" name="src_depth" type="uint8_t"/>
        </method>
        <method id="CNodeSaveCaller" name="SaveCaller">
            <param dir="in" name="index" type="seL4_Word"/>
            <param dir="in" name="depth" type="uint8_t"/>
        </method>
    </interface>
    <interface name="seL4_IRQControl">
        <method id="IRQIssueIRQHandler" name="Get">
            <param dir="in" name="irq" type="int"/>
            <param dir="in" name="root" type="seL4_CNode"/>
            <param dir="in" name="index" type="seL4_Word"/>
            <param dir="in" name="depth" type="uint8_t"/>
        </method>
        <method id="IRQInterruptControl" name="Control"/>
    </interface>
    <interface name="seL4_IRQHandler">
        <method id="IRQAckIRQ" name="Ack"/>
        <method id="IRQSetIRQHandler" name="SetEndpoint">
            <param dir="in" name="endpoint" type="seL4_CPtr"/>
        </method>
        <method id="IRQClearIRQHandler" name="Clear"/>
        <method id="IRQSetMode" name="SetMode">
            <param dir="in" name="level_trigger" type="uint32_t"/>
            <param dir="in" name="low_polarity" type="uint32_t"/>
        </method>
    </interface>
    <interface name="seL4_DomainSet">
        <method id="DomainSetSet" name="Set">
            <param dir="in" name="domain" type="uint8_t"/>
            <param dir="in" name="thread" type="seL4_TCB"/>
        </method>
    </interface>
</api>
```

9.3.1 CNode - Copy

原型：

static inline int seL4_CNode_Copy

```
static inline int
seL4_CNode_Copy(seL4_CNode service, seL4_Word dest_index, uint8_t dest_depth,
seL4_CNode src_root, seL4_Word src_index, uint8_t src_depth, seL4_CapRights
rights)
```

功能：

拷贝一个句柄，同时设置它的权限。

参数说明：

Type	Name	Description
seL4_CNode	service	用来指向目标CSpace根的CNode的CPTR，深度必须为32
seL4_Word	dest_index	目标CSpace中的槽，CPTR类型
uint8_t	dest_depth	dest_index中用来表示目标槽Slot的比特位数
seL4_CNode	src_root	用来指向源CSpace根的CNode的CPTR，深度必须为32
seL4_Word	src_index	源CSpace中的槽，CPTR类型
uint8_t	src_depth	src_index中用来表示目标槽Slot的比特位数
seL4_CapRights	rights	新句柄将继承的权限

类型 seL4_CapRights 定义：

```
typedef enum {
    seL4_CanWrite = 0x01,
    seL4_CanRead = 0x02,
    seL4_CanGrant = 0x04,
    seL4_AllRights = 0x07, /* seL4_CanWrite | seL4_CanRead | seL4_CanGrant */
    seL4_Transfer_Mint = 0x100,
    SEL4_FORCE_LONG_ENUM(seL4_CapRights),
} seL4_CapRights;
```

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

9.3.2 CNode - Delete

原型：

inline int seL4_CNode_Delete

```
static inline int
seL4_CNode_Delete(seL4_CNode service, seL4_Word index, uint8_t depth)
```

功能：

删除一个句柄。

参数说明：

Type	Name	Description
seL4_CNode	service	用来指向目标CSpace根的CNode的CPTR，深度必须为32

seL4_Word	dindex	目标CSpace中的槽，CPTR类型
uint8_t	depth	index中用来表示目标槽Slot的比特位数

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

9.3.3 CNode - Mint

这个函数与 seL4_CNode_Copy()的差别在于 seL4_CNode_Mint 多了个参数：标记（badge）。

原型：

static inline int seL4_CNode_Mint

```
static inline int
seL4_CNode_Mint(seL4_CNode service, seL4_Word dest_index, uint8_t dest_depth,
seL4_CNode src_root, seL4_Word src_index, uint8_t src_depth, seL4_CapRights
rights, seL4_CapData_t badge)
```

功能：

拷贝一个句柄，同时设置它的权限。

参数说明：

Type	Name	Description
seL4_CNode	service	用来指向目标CSpace根的CNode的CPTR，深度必须为32
seL4_Word	dest_index	目标CSpace中的槽，CPTR类型
uint8_t	dest_depth	dest_index中用来表示目标槽Slot的比特位数
seL4_CNode	src_root	用来指向源CSpace根的CNode的CPTR，深度必须为32
seL4_Word	src_index	源CSpace中的槽，CPTR类型
uint8_t	src_depth	src_index中用来表示目标槽Slot的比特位数
seL4_CapRights	rights	新句柄将继承的权限
seL4_CapData_t	badge	将用在新句柄上的标记

类型 seL4_CapRights 定义：

```
typedef enum {
    seL4_CanWrite = 0x01,
    seL4_CanRead = 0x02,
    seL4_CanGrant = 0x04,
    seL4_AllRights = 0x07, /* seL4_CanWrite | seL4_CanRead | seL4_CanGrant */
    seL4_Transfer_Mint = 0x100,
    SEL4_FORCE_LONG_ENUM(seL4_CapRights),
} seL4_CapRights;
```

seL4_CapData_t 的定义：

```
struct seL4_CapData {
    uint32_t words[1];
};
typedef struct seL4_CapData seL4_CapData_t;
```

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

9.3.4 CNode - Move

原型：

static inline int seL4_CNode_Move

```
static inline int
seL4_CNode_Move(seL4_CNode service, seL4_Word dest_index, uint8_t dest_depth,
seL4_CNode src_root, seL4_Word src_index, uint8_t src_depth)
```

功能：

移动一个句柄。

参数说明：

Type	Name	Description
seL4_CNode	service	用来指向目标CSpace根的CNode的CPTR，深度必须为32
seL4_Word	dest_index	目标CSpace中的槽，CPTR类型
uint8_t	dest_depth	dest_index中用来表示目标槽Slot的比特位数
seL4_CNode	src_root	用来指向源CSpace根的CNode的CPTR，深度必须为32
seL4_Word	src_index	源CSpace中的槽，CPTR类型
uint8_t	src_depth	src_index中用来表示目标槽Slot的比特位数

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

9.3.5 CNode - Mutate

原型：

static inline int seL4_CNode_Mutate

```
static inline int
seL4_CNode_Mutate(seL4_CNode service, seL4_Word dest_index, uint8_t dest_depth,
seL4_CNode src_root, seL4_Word src_index, uint8_t src_depth, seL4_CapData_t
badge)
```

功能：

句柄突变。移动一个句柄，并设置它的权限。

参数说明：

Type	Name	Description
seL4_CNode	service	用来指向目标CSpace根的CNode的CPTR，深度必须为32
seL4_Word	dest_index	目标CSpace中的槽，CPTR类型
uint8_t	dest_depth	dest_index中用来表示目标槽Slot的比特位数
seL4_CNode	src_root	用来指向源CSpace根的CNode的CPTR，深度必须为32
seL4_Word	src_index	源CSpace中的槽，CPTR类型
uint8_t	src_depth	src_index中用来表示目标槽Slot的比特位数
seL4_CapData_t	badge	将用在新句柄上的标记

seL4_CapData_t 的定义：

```
struct seL4_CapData {
    uint32_t words[1];
};
typedef struct seL4_CapData seL4_CapData_t;
```

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

9.3.6 CNode - Recycle

原型：

inline int seL4_CNode_Recycle

```
static inline int
seL4_CNode_Recycle(seL4_CNode service, seL4_Word index, uint8_t depth)
```

功能：

再生一个句柄，就是把一个句柄身上附着的有些资源解除绑定，重用它。它在功能上，有点类似于 seL4_CNode_Revoke，只是它多了把句柄初始化为原始状态的功能。

在相同的保护方式（protection domain）下，重置一个句柄，首先重置这个句柄，然后把与这个句柄关联的无关紧要的属性移除，使得它回归自然状态。

再生有标记的端点（badged endpoint）将取消有关这个标记的 IPC。

再生帧（frame）、页表（page table）和页目录（page directorie），应该在相同的保护域，再生后不是把所有的权力（authority）取消。

参数说明：

Type	Name	Description
seL4_CNode	service	用来指向目标CSpace根的CNode的CPTR，深度必须为32
seL4_Word	dindex	目标CSpace中的槽，CPTR类型
uint8_t	depth	index中用来表示目标槽Slot的比特位数

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

9.3.7 CNode - Revoke

原型：

inline int seL4_CNode_Revoke

```
static inline int
seL4_CNode_Revoke(seL4_CNode service, seL4_Word index, uint8_t depth)
```

功能：

删除所有子句柄。

参数说明：

Type	Name	Description
seL4_CNode	service	用来指向目标CSpace根的CNode的CPTR，深度必须为32
seL4_Word	dindex	目标CSpace中的槽，CPTR类型
uint8_t	depth	index中用来表示目标槽Slot的比特位数

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

9.3.8 CNode - Rotate

原型：

static inline int seL4_CNode_Rotate

```
static inline int
seL4_CNode_Rotate(seL4_CNode service, seL4_Word dest_index, uint8_t dest_depth,
seL4_CapData_t dest_badge, seL4_CNode pivot_root, seL4_Word pivot_index, uint8_t
pivot_depth, seL4_CapData_t pivot_badge, seL4_CNode src_root, seL4_Word
src_index, uint8_t src_depth)
```

功能：

假设有三个句柄：目标 destination、轴 pivot 和源 source，把轴移到目标，把源移到轴。

参数说明：

Type	Name	Description
seL4_CNode	service	用来指向目标CSpace根的CNode的CPTR，深度必须为32
seL4_Word	dest_index	目标CSpace中的槽，CPTR类型
uint8_t	dest_depth	dest_index中用来表示目标槽Slot的比特位数
seL4_CapData_t	dest_badge	将用在目标句柄上的标记
seL4_CNode	pivot_root	用来指向轴CSpace根的CNode的CPTR，深度必须为32
seL4_Word	pivot_index	轴CSpace中的槽，CPTR类型
uint8_t	pivot_depth	src_index中用来表示轴槽Slot的比特位数

seL4_CapData_t	pivot_badge	将用在轴句柄上的标记
seL4_CNode	src_root	用来指向源CSpace根的CNode的CPTR，深度必须为32
seL4_Word	src_index	源CSpace中的槽，CPTR类型
uint8_t	src_depth	src_index中用来表示目标槽Slot的比特位数
seL4_CapData_t	badge	将用在新句柄上的标记

seL4_CapData_t 的定义：

```
struct seL4_CapData {
    uint32_t words[1];
};
typedef struct seL4_CapData seL4_CapData_t;
```

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

9.3.9 CNode - Save Caller

原型：

inline int seL4_CNode_SaveCaller

```
static inline int
seL4_CNode_SaveCaller(seL4_CNode service, seL4_Word index, uint8_t depth)
```

功能：

保存线程最后应答句柄。

seL4_Call() 阻塞式调用时，它实质上是一个 seL4_Send()调用，发出请求的线程将被阻塞直到消息被接收。当消息被发送给接收者（通过端点 Endpoint），一个附加的应答句柄被同时发送，这使得消息接收者可以有权发应答信息。应答句柄存储于接收线程的 TCB 的特殊槽（slot）中。当通过 seL4_Call()调用内核对象句柄时，内核对象在返回消息中返回错误码或其它应答数据。

seL4_Reply() 应答阻塞式调用，用来应答 seL4_Call()，用 seL4_Call()生成并存储于 TCB 中的句柄，向调用者发送消息，唤醒它的线程。

TCB 中只有一个用作应答的句柄，所以 seL4_Reply 只能用来应答最近的消息。seL4_CNode_SaveCaller()可以用来在句柄空间中保存应答句柄，然后通过 seL4_Send 向这个句柄发送消息向调用者传送消息。

参数说明：

Type	Name	Description
seL4_CNode	service	用来指向目标CSpace根的CNode的CPTR，深度必须为32
seL4_Word	dindex	目标CSpace中的槽，CPTR类型
uint8_t	depth	index中用来表示目标槽Slot的比特位数

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

9.3.10 Debug - Halt

static inline void seL4_DebugHalt

系统停机。只有调试模式（debugging is turned on）有效。
没参数，也没有返回值。

9.3.11 Debug - Put Character

原型：

static inline void seL4_DebugPutChar

```
static inline void
seL4_DebugPutChar(char c)
```

功能：

输出一个字符到串口。只有调试模式（debugging is turned on）有效。
没有返回值。

参数说明：

Type	Name	Description
char	c	字符

9.3.12 Debug - Snapshot

原型：

```
static inline void seL4_SysDebugSnapshot
```

系统快照。只有调试模式（debugging is turned on）有效。
没参数，也没有返回值。

9.3.13 Debug - CapIdentify

原型：

```
static inline uint32_t
seL4_DebugCapIdentify(seL4_CPtr cap)
```

{

```
        register seL4_Word arg1 asm("r0") = cap;
        register seL4_Word scno asm("r7") = seL4_SysDebugCapIdentify;
        asm volatile ("swi %[swi_num]"
                      : "+r"(arg1)
                      :   [swi_num]   "i"   __SWINUM(seL4_SysDebugCapIdentify),
"r"(scno));
        return (uint32_t)arg1;
    }
```

9.3.14 Debug - Run

定义了宏 DANGEROUS_CODE_INJECTION 才有效。

原型：

```
static inline void
seL4_DebugRun(void (* userfn) (void *), void* userarg)
```

```
    {
        register seL4_Word arg1 asm("r0") = (seL4_Word)userfn;
        register seL4_Word arg2 asm("r1") = (seL4_Word)userarg;
        register seL4_Word scno asm("r7") = seL4_SysDebugRun;
        asm volatile ("swi %[swi_num]"
                      : /* no outputs */
                      : [swi_num] "i" __SWINUM(seL4_SysDebugRun), "r" (arg1),
"r" (arg2), "r"(scno)
                      : "memory"
                     );
    }
```

9.3.15 Benchmark - ResetLog

原型：

```
static inline void
seL4_BenchmarkResetLog(void)
```

```
    /* set the log index back to 0 */

    {
        register seL4_Word scno asm("r7") = seL4_SysBenchmarkResetLog;
        asm volatile ("swi %[swi_num]"
                      : /* no outputs */
                      :   [swi_num]   "i"   __SWINUM(seL4_SysBenchmarkResetLog),
"r"(scno)
                     );
    }

    /* read size words from the log starting from start into the ipc buffer.
```

```
     * @return the amount sucessfully read. Will cap at ipc buffer size and at
size of
     * recorded log */
    static inline uint32_t
```

9.3.16 Benchmark - DumpLog

原型：

```
seL4_BenchmarkDumpLog(seL4_Word start, seL4_Word size)
```

```
    {

        register seL4_Word arg1 asm("r0") = (seL4_Word) start;
        register seL4_Word arg2 asm("r1") = (seL4_Word) size;
        register seL4_Word scno asm("r7") = seL4_SysBenchmarkDumpLog;
        asm volatile ("swi %[swi_num]"
                      : "+r" (arg1)
                      : [swi_num] "i" __SWINUM(seL4_SysBenchmarkDumpLog), "r"
(arg1), "r" (arg2), "r"(scno));

        return (uint32_t) arg1;

    }
```

9.3.17 Benchmark - LogSize

原型：

```
static inline uint32_t
seL4_BenchmarkLogSize(void)
```

```
    /* Return the amount of things we tried to log. This could be greater than
     * the size of the log itself */

    {

        register seL4_Word arg1 asm("r0") = 0; /* required for retval */
        register seL4_Word scno asm("r7") = seL4_SysBenchmarkLogSize;
        asm volatile ("swi %[swi_num]"
                      : "+r" (arg1)
                      :   [swi_num]   "i"   __SWINUM(seL4_SysBenchmarkLogSize),
"r"(scno));

        return (uint32_t) arg1;

    }
```

9.3.18 DomainSet - Set

原型：

static inline int seL4_DomainSet_Set

```
static inline int
seL4_DomainSet_Set(seL4_DomainSet service, uint8_t domain, seL4_TCB thread)
```

功能：

改变一个线程的 domain。

参数说明：

类型 Type	参数名称 Name	描述 Description
seL4_IRQControl	service	允许domain定义的句柄
uint8_t	domain	线程的新domain
seL4_TCB	thread	指向线程的线程控制块TCB，这个线程是你操作的目标

数据类型定义：

```
typedef seL4_CPtr seL4_TCB;
typedef seL4_CPtr seL4_DomainSet;
```

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

9.3.19 IRQ Control - Get

原型：

static inline int seL4_IRQControl_Get

```
static inline int
seL4_IRQControl_Get(seL4_IRQControl service, int irq, seL4_CNode root, seL4_Word
index, uint8_t depth)
```

- -

功能：

创建一个中断处理句柄。

参数说明：

类型 Type	参数名称 Name	描述 Description
seL4_IRQControl	service	一个IRQControl句柄
int	irq	你想让你的句柄处理的中断
seL4_CNode	root	指向CNode的CPTR，用来标明你的CSpace，深度必须是32
seL4_Word	index	指向你的CSpace的目标槽slot的CPTR

uint8_t	depth	用来找目标槽slot的index中的比特位数

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

系统起动时，初始线程的 CNode 中有一个句柄：seL4_CapIRQControl，一切中断句柄，都是从此派生出来的。你如果在初始线程中，就可以这样写：

```
seL4_Error simple_stable_get_irq(void *data, int irq, seL4_CNode root, seL4_Word index, uint8_t depth) {
    return seL4_IRQControl_Get(seL4_CapIRQControl, irq, root, index, depth);
}
```

9.3.20 IRQ Handler - Acknowledge

原型：

static inline int seL4_IRQHandler_Ack

```
static inline int
seL4_IRQHandler_Ack(seL4_IRQHandler service)
```

功能：

使能应答一个中断。告知硬件中断处理装置，某个中断可以接收了。

参数说明：

类型 Type	参数名称 Name	描述 Description
seL4_IRQHandler	service	一个IRQControl句柄

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

9.3.21 IRQ Handler - Clear

原型：

static inline int seL4_IRQHandler_Clear

```
static inline int
seL4_IRQHandler_Clear(seL4_IRQHandler service)
```

- -

功能：

清除一个中断句柄。就是让这个句柄不再与那个中断发生关联。

参数说明：

类型 Type	参数名称 Name	描述 Description

seL4_IRQHandler	service	一个IRQControl句柄

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

9.3.22 IRQ Handler - Set Endpoint

原型：

static inline int seL4_IRQHandler_SetEndpoint

```
static inline int
seL4_IRQHandler_SetEndpoint(seL4_IRQHandler service, seL4_CPtr endpoint)
```

功能：

设置处理中断的端点（endpoint）。

参数说明：

类型 Type	参数名称 Name	描述 Description
seL4_IRQHandler	service	一个IRQControl句柄
seL4_CPtr	endoint	接收中断的端点

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

程序示例：

```
……
error = vka_alloc_async_endpoint(&env->vka, &aep);
……
timer_data->irq = timer_common_get_irq(vka, simple, irq_number);
……
    /* set the end point */
    error = seL4_IRQHandler_SetEndpoint(timer_data->irq, aep);
    if (error != seL4_NoError) {
        LOG_ERROR("seL4_IRQHandler_SetEndpoint failed with error %d\n", error);
        goto error;
    }
```

9.3.23 TCB - Configure

原型：

static inline int seL4_TCB_Configure

```
static inline int
```

```
seL4_TCB_Configure(seL4_TCB service, seL4_Word fault_ep, uint8_t priority,
seL4_CNode cspace_root, seL4_CapData_t cspace_root_data, seL4_CNode vspace_root,
seL4_CapData_t vspace_root_data, seL4_Word buffer, seL4_CPtr bufferFrame);
```

功能：

设置 TCB 的参数。

参数说明：

类型 Type	参数名称 Name	描述 Description
seL4_TCB	service	要操作的TCB句柄
seL4_Word	fault_ep	当IPC出错时，接收出错信息的句柄，这个句柄在线程的CSpace中，它是一个CPTR
uint8_t	priority	新的线程优先级
seL4_CNode	cspace_root	新的CSpace根
seL4_CapData_t	cspace_root_data	（可选）设置新CSpace的守卫guard尺寸， 如果设为0，这个参数不起作用
seL4_CNode	vspace_root	新的VSpace根
seL4_CapData_t	vspace_root_data	在IA-32平台、ARM平台无作用
seL4_Word	buffer	线程IPC缓存区，必须512字节对齐，IPC缓存区不可以跨页
seL4_CPtr	bufferFrame	线程IPC缓存区所在的页的句柄

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

9.3.24 TCB - Copy Registers

原型：

static inline int seL4_TCB_CopyRegisters

```
static inline int
seL4_TCB_CopyRegisters(seL4_TCB service, seL4_TCB source, bool suspend_source,
bool resume_target, bool transfer_frame, bool transfer_integer, uint8_t
arch_flags)
```

功能：

把寄存器从一个线程拷贝到另外一个线程。具体拷贝哪些寄存器，与平台相关，比如，有些平台有浮点协处理器，也可以拷贝。本操作之后，源线程可以被挂起（suspend），目标线程可以被唤醒（resume）。

在这个函数的上下文中，帧寄存器（frame register）是系统需要的，比如系统调用时，读、写、保留的寄存器、指令寄存器 pc、栈寄存器；系统不需要的寄存器，就是整数寄存器（integer register），如 IA-32 中的 EAX、ARM 中的 r0、r1 等寄存器。

因为整数寄存器在函数调用时不被保护，所以有时，复制它们的意思不大。

参数说明：

类型 Type	参数名称 Name	描述 Description
seL4_TCB	service	要操作的目标线程的TCB句柄，
seL4_TCB	source	要操作的源线程的TCB句柄，
bool	suspend_source	调用者应该挂起（suspend）源线程
bool	resume_target	调用者应该唤醒（resume）目标线程
bool	transfer_integer	应该传送整数寄存器
uint8_t	arch_flags	在IA-32平台、ARM平台无作用

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

9.3.25 TCB - Read Registers

原型：

static inline int seL4_TCB_ReadRegisters

```
static inline int
seL4_TCB_ReadRegisters(seL4_TCB service, bool suspend_source, uint8_t
arch_flags, seL4_Word count, seL4_UserContext *regs)
```

功能：

读一个线程的寄存器。

参数说明：

类型 Type	参数名称 Name	描述 Description
seL4_TCB	service	要操作的目标线程的TCB句柄，
bool	suspend_source	调用者应该挂起（suspend）源线程
uint8_t	arch_flags	在IA-32平台、ARM平台无作用
seL4_Word	count	读回的寄存器数量
seL4_UserContext	regs	读到的寄存器数据

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

seL4_UserContext 的定义：

IA-32 平台：

```
typedef struct {
    /* frameRegisters */
    seL4_Word eip, esp, eflags, eax, ebx, ecx, edx, esi, edi, ebp;
    /* gpRegisters */
    seL4_Word tls_base, fs, gs;
```

```
} seL4_UserContext;
```

ARM 平台：

```
typedef struct {
    /* frame registers */
    seL4_Word pc, sp, cpsr, r0, r1, r8, r9, r10, r11, r12;
    /* other integer registers */
    seL4_Word r2, r3, r4, r5, r6, r7, r14;
} seL4_UserContext;
```

9.3.26 TCB - Resume

static inline int seL4 TCB Resume

```
static inline int
seL4_TCB_Resume(seL4_TCB service)
```

Resume a thread

Type	Name	Description
seL4_TCB	_service	Capability to the TCB which is being operated on.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Section 5.1

9.3.27 TCB - Set IPC_Buffer

原型：

static inline int seL4 TCB SetIPCBuffer

```
static inline int
seL4_TCB_SetIPCBuffer(seL4_TCB service, seL4_Word buffer, seL4_CPtr bufferFrame)
```

功能：

设置一个线程的IPC缓存区（IPC Buffer）。

参数说明：

类型 Type	参数名称 Name	描述 Description
seL4_TCB	service	要操作的目标线程的TCB句柄，
seL4_Word	buffer	线程IPC缓存区的位置，必须512字节对齐，不可以跨页边界

seL4_CPtr	bufferFrame	包含IPC缓存区的页的句柄（Capability to a page）

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

9.3.28 TCB - Set Priority

static inline int seL4 TCB SetPriority

```
static inline int
seL4_TCB_SetPriority(seL4_TCB service, uint8_t priority)
```

Change a thread's priority

Type	Name	Description
seL4_TCB	_service	Capability to the TCB which is being operated on.
uint8_t	priority	The thread's new priority.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Section 5.1.3

9.3.29 TCB - Set Space

static inline int seL4 TCB SetSpace

```
static inline int
seL4_TCB_SetSpace(seL4_TCB service, seL4_Word fault_ep, seL4_CNode cspace_root,
seL4_CapData_t cspace_root_data, seL4_CNode vspace_root, seL4_CapData_t
vspace_root_data)
```

- -

Set the fault endpoint, CSpace and VSpace of a thread

Type	Name	Description
seL4_TCB	_ service	Capability to the TCB which is being operated on.
seL4_Word	fault _ ep	CPTR to the endpoint which receives IPCs when this thread faults. This capability is in the CSpace of the thread being con gured.

seL4_CNode	cspace_root	The new CSpace root.
seL4_CapData_t	cspace_root_data	Optionally set the guard and guard size of the new root CNode. If set to zero, this parameter has no e ect.
seL4_CNode	vspace_root	The new VSpace root.
seL4_CapData_t	vspace_root_data	Has no e ect on IA-32 or ARM proces-sors.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Section 5.1

9.3.30 TCB - Suspend

static inline int seL4 TCB Suspend

```
static inline int
seL4_TCB_Suspend(seL4_TCB service)
```

Suspend a thread

Type	Name	Description
seL4_TCB	_service	Capability to the TCB which is being operated on.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Section 5.1.2

9.3.31 TCB - Write Registers

原型：

static inline int seL4 TCB WriteRegisters

```
static inline int
seL4_TCB_WriteRegisters(seL4_TCB service, bool resume_target, uint8_t
arch_flags, seL4_Word count, seL4_UserContext *regs)
```

功能：

写一个线程的寄存器。

参数说明：

类型 Type	参数名称 Name	描述 Description
seL4_TCB	service	要操作的目标线程的TCB句柄
bool	resume_target	调用者唤醒（resume）源线程
uint8_t	arch_flags	在IA-32平台、ARM平台无作用
seL4_Word	count	要写的寄存器数量
seL4_UserContext	regs	要写的寄存器数据

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

9.3.32 Untyped - Retype

原型：

static inline int seL4_Untyped_Retype

```
static inline int
seL4_Untyped_Retype(seL4_Untyped service, int type, int size_bits, seL4_CNode root, int node_index, int node_depth, int node_offset, int num_objects)
```

功能：

赋型一块原始内存，也就是从物理内存里申请一内存，因为申请到的这块内存是有类型的，所以有了这样的说法。

参数说明：

类型 Type	参数名称 Name	描述 Description
seL4_Untyped	service	指向一个原始内存对象的CPTR
int	type	我们要创建的对象的类型 typedef enum api_object { seL4_UntypedObject, seL4_TCBObject, seL4_EndpointObject, seL4_AsyncEndpointObject, seL4_CapTableObject, seL4_NonArchObjectTypeCount, } seL4_ObjectType;
int	size_bits	只对可变大小对象有效
seL4_CNode	root	在目标CSpace中的CNode根CPTR
int	node_index	相对于root的偏移
int	node_depth	node_index中句柄的有效位数
int	node_offset	存储句柄的结点中的槽slot的偏移

int	num_objects	要创建的对象的数量

返回值：

成功则返回 0，不成功返回非 0 值，出错信息存于寄存器，见§9.1。

程序示例：

```
/* Try to drop two caps in, at the end of the cnode, overrunning it. */
error = seL4_Untyped_Retype(untyped.cptr,
                            seL4_TCBObject, 0,
                            env->cspace_root, cnode.cptr, seL4_WordBits,
                            (1 << 2) - 1, 2);
test_assert(error == seL4_RangeError);
```

补充说明：

这个函数的功能就是给你一个句柄 service，它指向一块原始内存区，即无类型的原始内存，创建 num_objects 个需要类型的对象，同时创建 num_objects 个句柄，这些句柄存在 node_offset 开始的 CNode 中，与对象一一对应。

关于象 CNode 这样的对象具有可变大小的问题，下面详细描述一下。

很多内核对象的大小是确定了的，所以创建时，只要告知类型就可以，但象 CNode、Untyped Memory 这样的对象，大小是可变的，就需要通过参数 size_bits 告知你要创建的对象的大小。对 CNode，每个 CNode 中槽 slot 的数量，表述为：2^size_bits，每个槽占 16 字节，所以整个这个对象点用的内存是 16*2^size_bits 字节。对 Untyped Memory 的情形，这个函数是要申请很多 Untyped Memory 小块，每个小块的大小为 2^size_bits。

如果 service 句柄所指的原始内存的大小足够分配，记住，总的内存大小是对象大小乘以对象数量 num_objects，函数返回 0，并正确处理相应逻辑，即申请内存、句柄，并把句柄指向相应内存，如果原始内存的大小不足够分配，则函数失败。

申请原始内存的算法，是在指定的原始内存区的开始部分，找到未分配，同时与要分配的对象尺寸对齐的内存，这种算法会在原始内存中，申请了的对象间有缝隙（gap），即虽然未分配给谁，但没法用。

申请得到的句柄在 CNode 中是连续存储的。CNode 通过 root、node_index 和 node_depth 参数指定。node_offset 参数指定 CNode 中第一个句柄的存放位置。num_objects 参数指定要创建的对象的数量，所有槽 slot 必须是空的，否则将出错。

所有结果对象的句柄存在于同一个 CNode 中。

表 9.1: 平台无关的对象大小

Object	Object Size
n-bit Untyped	2^n bytes (where n 4)
n-slot CNode	16n bytes (where n 2)
Synchronous Endpoint	16 bytes
Asynchronous Endpoint	16 bytes
IRQ Control	--
IRQ Handler	--

9.3.33 对象大小汇总

当你在原始内存中申请内存，也就是对原始内存进行赋型时，知道你要申请的对象的大小，是挺重要的。表9.2和表9.3列出了常用内核对象的大小。

表9.2: IA-32 平台相关对象大小

IA-32 Object	Object Size
Thread Control Block	1KiB
IA32 4K Frame	4KiB
IA32 4M Frame	4MiB
IA32 Page Directory	4KiB
IA32 Page Table	4KiB
IA32 ASID Control	--
IA32 ASID Pool	4KiB
IA32 Port	--
IA32 IO Space	--
IA32 IO Page table	4KiB

表9.3: ARM平台相关对象

ARM Object	Object Size
Thread Control Block	512 bytes
ARM Small Frame	4KiB
ARM Large Frame	64KiB
ARM Section	1MiB
ARM Supersection	16MiB
ARM Page Directory	16KiB
ARM Page Table	1KiB
ARM ASID Control	--
ARM ASID Pool	4KiB

9.4 IA-32 特定对象方法（IA-32-Specific Object Methods）

9.4.1 IA32 ASID Control - Make Pool

static inline int seL4_IA32_ASIDControl_MakePool

Create an IA-32 ASID pool

Type	Name	Description
seL4_IA32_ASIDControl	_service	The master ASIDControl capability.
seL4_Untyped	untyped	Capability to an untyped memory object that will become the pool. Must be 4K bytes.
seL4_CNode	root	CPTR to the CNode that forms the root of the destination CSpace. Must be at a depth of 32.
seL4_Word	index	CPTR to the destination slot. Resolved from the root of the destination CSpace.
uint8_t	depth	Number of bits of index to resolve to nd the destination slot.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Chapter 6

9.4.2 IA32 ASID Pool - Assign

static inline int seL4_IA32_ASIDPool_Assign

Assign an ASID pool

Type	Name	Description
seL4_IA32_ASIDPool	service	The ASID pool which is being assigned to. Must not be full. Each ASID pool can contain 1024 entries.
seL4_IA32_PageDirectory	vroot	The page directory that is being assigned to an ASID pool. Must not already be assigned to an ASID pool.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Chapter 6

9.4.3 IA32 IO Port - In 8

static inline seL4_IA32_IOPort_In8_t seL4_IA32_IOPort_In8

Read 8 bits from an IO port

Type	Name	Description
seL4_IA32_IOPort	service	An IO port capability.
uint16_t	port	The port to read from.

Return value: A seL4_IA32_IOPort_In8_t structure as described in Section 7.2.1

Description: See Section 7.2.1

9.4.4 IA32 IO Port - In 16

static inline seL4_IA32_IOPort_In16_t seL4_IA32_IOPort_In16

Read 16 bits from an IO port

Type	Name	Description
seL4_IA32_IOPort	service	An IO port capability.
uint16_t	port	The port to read from.

Return value: A seL4_IA32_IOPort_In16_t structure as described in Section 7.2.1

Description: See Section 7.2.1

9.4.5 IA32 IO Port - In 32

static inline seL4_IA32_IOPort_In32_t seL4_IA32_IOPort_In32

Read 32 bits from an IO port

Type	Name	Description
seL4_IA32_IOPort	service	An IO port capability.
uint16_t	port	The port to read from.

Return value: A seL4_IA32_IOPort_In32_t structure as described in Section 7.2.1

Description: See Section 7.2.1

9.4.6 IA32 IO Port - Out 8

static inline int seL4_IA32_IOPort_Out8

Write 8 bits to an IO port

Type	Name	Description
seL4_IA32_IOPort	service	An IO port capability.
uint16_t	port	The port to write to.
uint8_t	data	Data to write to the IO port.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Section 7.2.1

9.4.7 IA32 IO Port - Out 16

static inline int seL4_IA32_IOPort_Out16

Write 16 bits to an IO port

Type	Name	Description
seL4_IA32_IOPort	service	An IO port capability.
uint16_t	port	The port to write to.
uint16_t	data	Data to write to the IO port.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Section 7.2.1

9.4.8 IA32 IO Port - Out 32

static inline int seL4_IA32_IOPort_Out32_

Write 32 bits to an IO port

Type	Name	Description
seL4_IA32_IOPort_t	service	An IO port capability.
uint16_t	port	The port to write to.
uint32_t	data	Data to write to the IO port.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Section 7.2.1

9.4.9 IA32 IO Page Table - Map

static inline int seL4_IA32_IOPageTable_Map_

Map a page table into an IOSpace

Type	Name	Description
seL4_IA32_IOPageTable_t	service	The page table that is being mapped.

seL4_IA32_IOSpace	iospace	The IOSpace that the page table is being mapped into.
seL4_Word	ioaddr	The address that the page table is being mapped at.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Section 7.2.2

9.4.10 IA32 Page - Map IO

static inline int seL4_IA32_Page_MapIO

Map a page into an IOSpace

Type	Name	Description
seL4_IA32_Page	service	The frame that is being mapped.
seL4_IA32_IOSpace	iospace	The IOSpace that the frame is being mapped into.
seL4_CapRights	rights	Rights for the mapping. Possible values for this type are given in Section 3.1.3.
seL4_Word	ioaddr	The address that the frame is being mapped at.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Section 7.2.2

9.4.11 IA32 Page - Map

static inline int seL4_IA32_Page_Map

Map a page into an address space

Type	Name	Description
seL4_IA32_Page	service	Capability to the page to map.
seL4_IA32_PageDirectory	pd	Capability to the VSpace which will contain the mapping.
seL4_Word	vaddr	Virtual address to map the page into.
seL4_CapRights	rights	Rights for the mapping. Possible values for this type are given in Section 3.1.3.
seL4_IA32_VMAttributes	attr	VM Attributes for the mapping. Possible values for this type are given in Chapter 6.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Chapter 6

9.4.12 IA32 Page - Remap

static inline int seL4_IA32_Page_Remap

Remap a page

Type	Name	Description
seL4_IA32_Page	service	Capability to the page to map.

seL4_IA32_PageDirectory	pd	Capability to the VSpace which will contain the mapping.
seL4_CapRights	rights	Rights for the mapping. Possible values for this type are given in Section 3.1.3.
seL4_IA32_VMAttributes	attr	VM Attributes for the mapping. Possible values for this type are given in Chapter 6.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Chapter 6

9.4.13 IA32 Page - Unmap

static inline int seL4_IA32_Page_Unmap

Unmap a page

Type	Name	Description
seL4_IA32_Page	_service	Capability to the page to unmap.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Chapter 6

9.4.14 IA32 Page - Get Address

static inline seL4 IA32 Page GetAddress t seL4 IA32 Page GetAddress

Get the physical address of the underlying frame

Type	Name	Description
seL4_IA32_Page	service	Capability to the page to lookup.

Return value: A seL4_IA32_Page_GetAddress_t structure as described in TODO

Description: See Chapter 6

9.4.15 IA32 Page Table - Map

static inline int seL4 IA32 PageTable Map _

Map a page table into an address space

Type	Name	Description
seL4_IA32_PageTable	service	Capability to the page table to map.
seL4_IA32_PageDirectory	pd	Capability to the VSpace which will contain the mapping.
seL4_Word	vaddr	Virtual address to map the page into.
seL4_IA32_VMAttributes	attr	VM Attributes for the mapping. Possible values for this type are given in Chapter 6.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Chapter 6

9.4.16 IA32 Page Table - Unmap

static inline int seL4_IA32_PageTable_Unmap

Unmap a page table from its address space and zero it out

Type	Name	Description
seL4_IA32_PageTable_t	service	Capability to the page table to unmap.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Chapter 6

9.5 ARM-Specific Object Methods

9.5.1 ARM ASID Control - Make Pool

static inline int seL4_ARM_ASIDControl_MakePool

Create an ASID Pool

Type	Name	Description

seL4_ARM_ASIDControl	service	The master ASIDControl capability.
seL4_Untyped	untyped	Capability to an untyped memory object that will become the pool. Must be 4K bytes.
seL4_CNode	root	CPTR to the CNode that forms the root of the destination CSpace. Must be at a depth of 32.
seL4_Word	index	CPTR to the destination slot. Resolved from the root of the destination CSpace.
uint8_t	depth	Number of bits of index to resolve to find the destination slot.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Chapter 6

9.5.2 ARM ASID Pool - Assign

static inline int seL4_ARM_ASIDPool_Assign

Assign an ASID Pool

Type	Name	Description
seL4_ARM_ASIDPool	_service	The ASID pool which is being assigned to. Must not be full. Each ASID pool can contain 1024 entries.
seL4_ARM_PageDirectory	vroot	The page directory that is being assigned to an ASID pool. Must not already be assigned to an ASID pool.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Chapter 6

9.5.3 ARM Page - Flush Caches

static inline int seL4_ARM_Page_FlushCaches

Flush a cache range

Type	Name	Description
seL4_ARM_Page	service	The page whose contents will be flushed.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Chapter 6

9.5.4 ARM Page - Map

static inline int seL4_ARM_Page_Map

Map a page into an address space

Type	Name	Description
seL4_ARM_Page	service	Capability to the page to map.

seL4_ARM_PageDirectory	pd	Capability to the VSpace which will contain the mapping.
seL4_Word	vaddr	Virtual address to map the page into.
seL4_CapRights	rights	Rights for the mapping. Possible values for this type are given in Section 3.1.3.
seL4_ARM_VMAttributes	attr	VM Attributes for the mapping. Possible values for this type are given in Chapter 6.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Chapter 6

9.5.5 ARM Page - Remap

static inline int seL4 ARM Page Remap

Remap a page

Type	Name	Description
seL4_ARM_Page	service	Capability to the page to remap.
seL4_ARM_PageDirectory	pd	Capability to the VSpace which will contain the mapping.
seL4_CapRights	rights	Rights for the mapping. Possible values for this type are given in Section 3.1.3.
seL4_ARM_VMAttributes	attr	VM Attributes for the mapping. Possible values for this type are given in Chapter 6.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag

contents upon error.

Description: See Chapter 6

9.5.6 ARM Page - Unmap

static inline int seL4_ARM_Page_Unmap

Unmap a page

Type	Name	Description
seL4_ARM_Page	service	Capability to the page to unmap.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Chapter 6

9.5.7 ARM Page - Get Address

static inline seL4_ARM_Page_GetAddress_t seL4_ARM_Page_GetAddress

Get the physical address of the underlying frame

Type	Name	Description
seL4_ARM_Page	service	Capability to the page to lookup.

Return value: A seL4_ARM_Page_GetAddress_t structure as described in TODO

Description: See Chapter 6

9.5.8 ARM Page Table - Map

static inline int seL4_ARM_PageTable_Map_

Map a page table into an address space

Type	Name	Description
seL4_ARM_PageTable	_service	Capability to the page table that will be mapped.
seL4_ARM_PageDirectory	pd	Capability to the VSpace which will contain the mapping.
seL4_Word	vaddr	Virtual address to map the page into.
seL4_ARM_VMAttributes	attr	VM Attributes for the mapping. Possible values for this type are given in Chapter 6.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Chapter 6

9.5.9 ARM Page Table - Unmap

static inline int seL4_ARM_PageTable_Unmap

Unmap a page table from its address space and zero it out

Type	Name	Description
seL4_ARM_PageTable	service	Capability to the page table that will be unmapped.

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See Section 9.1 for a description of the message register and tag contents upon error.

Description: See Chapter 6

10 muslc c 库

需要如下两个 lib 库，才能使用 seL4 的 c 库：

- libmuslc http://www.musl-libc.org/ musl is lightweight, fast, simple, free, and strives to be correct in the sense of standards-conformance and safety.
- libsel4muslcsys， libsel4muslcsys 提供了 libmuslc 所需要系统调用

11 内存相关的库

- libsel4allocman, 通过 vka 实现的内存、对象、句柄申请器，seL4 上申请一块内存，是个挺复杂的事情，需要管理虚拟地址、虚实映射、内核对象使用等等，在上面包装上一层 API，方便程序员使用。
- libsel4vka, vka 是通用虚拟内存申请器（Generic Virtual Kernel Allocator）的意思，这个库只定义了最基础的接口，所有内存申请器都要实现这些接口
- libsel4vspace, Interface for virtual memory management on seL4.

概念解释：

- CSpace，Capability Space 的缩写，seL4 中的数据结构：seL4_CNode
- VSpace， Virtual Space 的缩写
- 申请器标识 cookie，seL4 内核自身不管理动态数据，但是在系统运行过程中，会有很多动态的东西，比如你申请了一块原始内存，总得有地方记录一下这块内存的信息吧？这些东西，在 seL4 中，都是记录在被称为 CSpace（Capability Space）的地方，一个系统，运行起来，很有 N（N>1）个 CSpace，你每次委托内核给你干点事，是不是要告诉内核，我当前存放管理信息的这个 CSpace 啊？在申请内存时，这个 CSpace 通常就是“申请器标识 cookie”。
- Generic Virtual Kernel Allocator (VKA)

11.1 libsel4allocman

seL4 通过内核申请来的资源五花八门，各种申请途径，各种资源，如果没有一个统一的管理机制，很容易让各种资源变得不一致，于是，设计了 allocman 这个库。

释放操作的时候，也可能递归地有申请操作。

```
 * It is generally desirable that a free operation does not have any
 * allocation calls in it. If an allocator does wish to allocate a
 * resource when performing a free it must accept that its allocation
 * function could be called as a result. In a similar manner if your
 * allocation function frees resources your free function may be recursively
 * called.
 *
 * There are (generally) two different types of allocators. Those that are
 * linked to an allocation manager, and those that are not. Typically the
 * only sort of manager you would not want linked to an allocation manager
 * is a cspace manager (if you are managing a clients cspace). Although you
 * could also create an untyped manager if you do not want to give clients
```

```
 * untypeds directly, but still want to have a fixed untyped pool reserved
 * for it.
 *
 * Possibility exists for much foot shooting with any allocators. A typical
 * desire might be to create a sub allocator (such as a cspace manager),
 * use an already existing allocation manager to back all of its allocations,
 * and then destroy that cspace manager at some point to release all its resources.
 * There are no guarantees that this will work. If all requests to the sub allocator
 * use the same allocation manager to perform book keeping requests, and the
 * sub allocator is told to free using that same allocation manager then all
 * should work. But this is strictly up to using your allocators correctly,
 * and knowing how they work.
 */
```

11.1.1 allocman 数据结构

首先看这个函数的原型：

```
allocman_t *bootstrap_use_bootinfo(seL4_BootInfo *bi, uint32_t pool_size, char *pool);
```

在系统起动阶段，bootstrap 就把 allocman_t 这个数据结构传给应用程序了。

下面是 allocman_t 这个数据结构的定义：

```
/**
 * The allocman itself. This is generally the only type you will need to pass around
 * to deal with allocation. It is declared in full here so that the compiler is able
 * to calculate its size so it can be allocated on stacks/globals etc as required
 */
typedef struct allocman {
    /* link to our underlying allocators. some are lazily added. the mspace will always be here,
     * and have_mspace can be used to check if the allocman is initialized at all */
    int have_mspace;
    struct mspace_interface mspace;
    int have_cspace;
    struct cspace_interface cspace;
    int have_utspace;
    struct utspace_interface utspace;

    /* Flag that tracks whether any alloc/free/other function has been entered yet */
```

```
    int in_operation;

    /* Counts that track re-entry into each specific alloc/free function */
    uint32_t cspace_alloc_depth;
    uint32_t cspace_free_depth;
    uint32_t utspace_alloc_depth;
    uint32_t utspace_free_depth;
    uint32_t mspace_alloc_depth;
    uint32_t mspace_free_depth;

    /* Track whether the watermark is currently refilled so we don't recursively
do it */
    int refilling_watermark;
    /* Has a watermark resource been used. This is just an optimization */
    int used_watermark;

    /* track resources that we have not yet been able to free due to circular
dependencies */
    uint32_t desired_freed_slots;
    uint32_t num_freed_slots;
    cspacepath_t *freed_slots;

    uint32_t desired_freed_mspace_chunks;
    uint32_t num_freed_mspace_chunks;
    struct allocman_freed_mspace_chunk *freed_mspace_chunks;

    uint32_t desired_freed_utspace_chunks;
    uint32_t num_freed_utspace_chunks;
    struct allocman_freed_utspace_chunk *freed_utspace_chunks;

    /* cspace watermark resources */
    uint32_t desired_cspace_slots;
    uint32_t num_cspace_slots;
    cspacepath_t *cspace_slots;

    /* mspace watermark resources */
    uint32_t num_mspace_chunks;
    struct allocman_mspace_chunk *mspace_chunk;
    uint32_t *mspace_chunk_count;
    void ***mspace_chunks;

    /* utspace watermark resources */
    uint32_t num_utspace_chunks;
    struct allocman_utspace_chunk *utspace_chunk;
```

```
    uint32_t *utspace_chunk_count;
    struct allocman_utspace_allocation **utspace_chunks;
} allocman_t;
```

11.1.2 allocman 函数

allocman_mspace_alloc

```
void *allocman_mspace_alloc(allocman_t *alloc, uint32_t bytes, int *_error);
```

功能：

申请实实在在的内存

参数：

[输入] alloc 指向 allocman_t 的指针
[输入] bytes 要申请的内存的字节数
[输出] _error 申请成功被设为 0

返回值：

指向申请得到的内存的指针

allocman_mspace_free

```
void allocman_mspace_free(allocman_t *alloc, void *ptr, uint32_t bytes);
```

功能：

释放 allocman_mspace_alloc()得到的内存

参数：

[输入] alloc 指向 allocman_t 的指针
[输入] ptr 指向要释放内存的指针
[输入] bytes 要释放的内存的字节数

返回值：

（无）

allocman_cspace_alloc

```
int allocman_cspace_alloc(allocman_t *alloc, cspacepath_t *slot);
```

功能：

申请一个槽 Slot

参数：

[输入] alloc 指向 allocman_t 的指针

[输入] slot 指向存放 Slot 的 cspacepath_t 的指针

返回值：

成功则返回 0

allocman_cspace_free

```
void allocman_cspace_free(allocman_t *alloc, cspacepath_t *slot);
```

功能：

释放一个槽 Slot

参数：

[输入] alloc 指向 allocman_t 的指针

[输入] slot 指向存放 Slot 的 cspacepath_t 的指针，为了避免 slot 中的数据不是一个正确的 Slot，约定：(*slot) == allocman_cspace_make_path(alloc, slot->capPtr)

返回值：

成功则返回 0

allocman_cspace_make_path

```
static inline cspacepath_t allocman_cspace_make_path(allocman_t *alloc, seL4_CPtr slot) {
    assert(alloc->have_cspace);
    return alloc->cspace.make_path(alloc->cspace.cspace, slot);
}
```

功能：

转换 seL4_CPtr 为 cspacepath_t，如果这个槽不在那个 CSpace 中，则返回结果不确定

参数：

[输入] alloc 指向 allocman_t 的指针

[输入] slot seL4_CPtr 指针

返回值：

成功则返回存有 Slot 的 cspacepath_t

allocman_utspace_alloc

```
uint32_t allocman_utspace_alloc(allocman_t *alloc, uint32_t size_bits, seL4_Word type, cspacepath_t *path, int *_error);
```

功能：

申请一块原始内存（untyped memory）

参数：

[输入] alloc 指向 allocman_t 的指针

[输入] size_bits 大小，这个大小有别于申请 seL4_CapTableObjects 时传给 seL4_Untyped_Retype 的大小

[输入] type seL4 数据类型

[输入] path 用来存放句柄的 cspacepath_t 数据指针

[输入] _error 成功则被赋值为 0

返回值：

成功则返回一个标记，以后可以通过这个标记释放这个内存

allocman_utspace_free

```
void allocman_utspace_free(allocman_t *alloc, uint32_t cookie, uint32_t size_bits);
```

功能：

释放一块原始内存（untyped memory），假定这块原始内存中的句柄等资源已经被正确释放

参数：

[输入] alloc 指向 allocman_t 的指针

[输入] cookie 指向标记

[输入] size_bits 大小，这个大小有别于申请 seL4_CapTableObjects 时传给 seL4_Untyped_Retype 的大小

返回值：

（无）

allocman_create

```
int allocman_create(allocman_t *alloc, struct mspace_interface mspace);
```

功能：

初始化一个新的 allocman_t，所有信息是在系统启动（boot）时的设置

参数：

[输入] alloc 指向将要被初始化的 allocman_t 的指针，

[输入] mspace 内存申请器，mspace_interface 结构，这个申请器必须是一直存在的，否则调用一个不存在的函数能带来什么后果，完全不可知

返回值：

（无）

allocman_fill_reserves

```
/**
 * Attempts to fill the reserves of the allocator. This can be used if the underlying allocators have been modified,
 * for instance by having resources added, or as a way to query the health of the allocman
 *
 * @param alloc The allocman to fill reserves of
 *
 * @return returns 0 if reserves are full
 */
int allocman_fill_reserves(allocman_t *alloc);
```

功能：

初

参数：

[输入] alloc 指向将要被初始化的 allocman_t 的指针，

返回值：

成功返回 0

allocman_attach_utspace

```
int allocman_attach_utspace(allocman_t *alloc, struct utspace_interface utspace);
```

功能：

把一个原始内存（untyped memory）申请器附着在 allocman 上

参数：

[输入] alloc 指向 allocman_t 的指针

[输入] utspace_interface utspace

返回值：

成功返回 0

allocman_attach_cspace

```
int allocman_attach_cspace(allocman_t *alloc, struct cspace_interface cspace);
```

功能：

把一个 CSpace 管理器附着在 allocman 上

参数：

[输入] alloc 指向 llocman_t 的指针，

[输入] cspace_interface cspace

返回值：

成功返回 0

allocman_configure_mspace_reserve

```
int allocman_configure_mspace_reserve(allocman_t *alloc, struct allocman_mspace_chunk chunk);
```

功能：

设置 allocman 中的大块保留数据区

参数：

[输入] alloc 指向 allocman_t 的指针，

[输入] allocman_mspace_chunk chunk

返回值：

成功返回 0

allocman_configure_utspace_reserve

```
int allocman_configure_utspace_reserve(allocman_t *alloc, struct allocman_utspace_chunk chunk);
```

功能：

设置 allocman 中的大块原始保留数据区

参数：

[输入] alloc 指向 allocman_t 的指针，

[输入] allocman_utspace_chunk chunk

返回值：

成功返回 0

allocman_configure_cspace_reserve

```
int allocman_configure_cspace_reserve(allocman_t *alloc, uint32_t num);
```

功能：

设置 allocman 中的大块 CSpace 保留数据区

参数：

[输入] alloc 指向 allocman_t 的指针，

[输入] num 保留数据区中准备存储的 cslots 数量

返回值：

成功返回 0

allocman_configure_max_freed_slots

```
int allocman_configure_max_freed_slots(allocman_t *alloc, uint32_t num);
```

功能：

设置最大已经释放的 cptrs 存储数量，当一个内存申请器 allocator 不能递归调用时使用，当然我们不希望有内存泄漏

参数：

[输入] alloc allocman_t 的指针，

返回值：

成功返回 0

allocman_configure_max_freed_memory_chunks

```
int allocman_configure_max_freed_memory_chunks(allocman_t *alloc, uint32_t num);
```

功能：

设置最大已经释放了的对象的存储数量，当一个内存申请器 allocator 不能递归调用时使用，当然我们不希望有内存泄漏

参数：

[输入] alloc 指向 allocman_t 的指针，

返回值：

成功返回 0

allocman_configure_max_freed_untyped_chunks

```
int allocman_configure_max_freed_untyped_chunks(allocman_t *alloc, uint32_t num);
```

功能：

设置最大已经释放了的原始内存对象的存储数量，当一个内存申请器 allocator 不能递归调用时使用，当然我们不希望有内存泄漏

参数：

[输入] alloc 指向 allocman_t 的指针

[输入] num 数量

返回值：

成功返回 0

allocman_utspace_add_uts

```
/**
 * Add additional untyped objects to the underlying untyped manager. This allows additional
 * resources to be injected after the allocman has started
 *
 * @param alloc The allocman to add to
 * @param num Number of untypeds to add
 * @param uts Path to each of the untyped to add. untyped is assumed to be at depth 32 from this threads cspace_root
 * @param size_bits Size, in bits, of each of the untypeds
 * @param paddr Optional parameter specifying the physical address of each of the untypeds
 *
 * @return returns 0 on success
 */
static inline int allocman_utspace_add_uts(allocman_t *alloc, uint32_t num, cspacepath_t *uts, uint32_t *size_bits, uint32_t *paddr) {
    int error;
    assert(alloc->have_utspace);
    error = alloc->utspace.add_uts(alloc, alloc->utspace.utspace, num, uts, size_bits, paddr);
    if (error) {
        return error;
    }
    allocman_fill_reserves(alloc);
    return 0;
```

```
}
```

功能：

初

参数：

[输入] alloc 指向将要被初始化的 allocman_t 的指针，

返回值：

成功返回 0

allocman_utspace_paddr

```
static inline uint32_t allocman_utspace_paddr(allocman_t *alloc, uint32_t cookie,
uint32_t size_bits) {
    assert(alloc->have_utspace);
    return alloc->utspace.paddr(alloc->utspace.utspace, cookie, size_bits);
}
```

功能：

求原始内存对象类型的物理指针

参数：

[输入] alloc 指向 allocman_t 的指针，

[输入] cookie 原始内存类型，

[输入] size_bits 大小

返回值：

物理内存地址

11.1.3 bootstrap 数据结构

bootstrap.h 中是从 0 开始进行系统引导的辅助函数。微内核系统，对内存的使用、内存管理等等信息全不管，只是生成了一个简单的 BootInfo（见§8），那是完全不够用的。这些 bootstrap 中的函数调用完后得到的就是 allocman，你的内存管理就靠 allocman 往下做吧。想自己发明一个 allocman 那不是个简单的事。

引导一个系统是挺困难的，这些辅助函数试图让你只是通常一些配置，就可以把系统引导起来。

通常情况下，引导一个系统要做下列事情：

- 描述当前句柄空间 CSpace；

- 描述哪里可以得到资源，如 untypeds、slots 等；
- 可能的情况下，还要切换到一个新的句柄空间 CSpace 中。

引导系统需要一些初始池内存（pool memory），这是在你启动系统后**永远**不可释放的内存。它不能放在你的程序的栈里，这里肯定知道，否则你写什么操作系统程序啊？实块内存也不能是你的程序的全局变量，你的程序可以挂了，可是挂了后擦屁股的活还得有人干吧？这里是微内核，没有超级管理者，这块内存你也不能碰。

```
struct bootstrap_info {
    allocman_t *alloc;
    int have_boot_cspace;
    /* a path to the current boot cnode */
    cspacepath_t boot_cnode;
    /* this is prefixed with 'maybe' since we are just using it as some preallocated
memory
     * if a temp bootstrapping cspace is needed. it may not actually be used if a
more
     * complicated cspace is passed in directly by the user */
    cspace_simple1level_t maybe_boot_cspace;
    cspace_interface_t boot_cspace;
    /* if we have switched cspaces, a patch to our old root cnode (based in the current
cnode) */
    cspacepath_t old_cnode;
    /* a path to the page directory cap, we need this to be able to change the cspace
root */
    cspacepath_t pd;
    cspacepath_t tcb;
    int uts_in_current_cspace;
    uint32_t num_uts;
    cspacepath_t *uts;
    uint32_t *ut_size_bits;
    uint32_t *ut_paddr;
    simple_t *simple;
};
```

seL4 启动时的内存结构：

```
/* For the initial vspace, we must always guarantee we have virtual memory available
 * for each bottom level page table. Future vspaces can then use the initial vspace
 * to allocate bottom level page tables until memory runs out.
 *
 * The initial vspace then looks like this:
 * 0xffffffff ↑
 *            | kernel code
```

```
 * 0xe0000000 ↓
 * 0xffffffff ↑
 *            | bottom level page tables
 * 0xdf800000 ↓
 * 0xdf7fffff ↑
 *            | top level page table
 * 0xdf7ff000 ↓
 * 0xdf7fefff ↑
 *            | available address space
 * 0x00001000 ↓
 * 0x00000fff ↑
 *            | reserved (null page)
 * 0x00000000 ↓
 *
 * The following constants come from the above layout.
 */
```

看看所有测试程序启动之入口程序$/apps/sel4test-driver/src/main.c 中的源码。

```
struct env {
    /* An initialised vka that may be used by the test. */
    vka_t vka;
    /* virtual memory management interface */
    vspace_t vspace;
    /* abtracts over kernel version and boot environment */
    simple_t simple;
    /* path for the default timer irq handler */
    cspacepath_t irq_path;
#ifdef CONFIG_ARCH_ARM
    /* frame for the default timer */
    cspacepath_t frame_path;
#elif CONFIG_ARCH_IA32
    /* io port for the default timer */
    seL4_CPtr io_port_cap;
#endif
    /* init data frame vaddr */
    test_init_data_t *init;
    /* extra cap to the init data frame for mapping into the remote vspace */
    seL4_CPtr init_frame_cap_copy;
};
```

```

/* static memory for the allocator to bootstrap with */
#define ALLOCATOR_STATIC_POOL_SIZE ((1 << seL4_PageBits) * 10)
```

```
static char allocator_mem_pool[ALLOCATOR_STATIC_POOL_SIZE];

/* initialise our runtime environment */
static void
init_env(env_t env)
{
    allocman_t *allocman;
    UNUSED reservation_t virtual_reservation;
    UNUSED int error;

    // 在静态内存 allocator_mem_pool（这块内存是程序加载器 elfLoader 加载程序时开辟的)中,创建内存管理器 allocman,这个函数会把你的程序需要的句柄空间 CSpace、VSpace 等一关处理好
    allocman            =            bootstrap_use_current_simple(&env->simple, ALLOCATOR_STATIC_POOL_SIZE, allocator_mem_pool);
    assert(allocman);

    //创建一个虚拟内存管理界面 virtual kernel allocator
    /* create a vka (interface for interacting with the underlying allocator) */
    allocman_make_vka(&env->vka, allocman);

    /* create a vspace (virtual memory management interface). We pass
     * boot info not because it will use capabilities from it, but so
     * it knows the address and will add it as a reserved region */
    error = sel4utils_bootstrap_vspace_with_bootinfo_leaky(&env->vspace,
                                                           &data,
simple_get_pd(&env->simple), &env->vka, seL4_GetBootInfo());

    /* fill the allocator with virtual memory */
    void *vaddr;
    virtual_reservation = vspace_reserve_range(&env->vspace,
                                               ALLOCATOR_VIRTUAL_POOL_SIZE,
seL4_AllRights, 1, &vaddr);
    assert(virtual_reservation.res);
    bootstrap_configure_virtual_pool(allocman, vaddr,
                                     ALLOCATOR_VIRTUAL_POOL_SIZE,
simple_get_pd(&env->simple));
}
```

sel4utils_bootstrap_vspace_with_bootinfo()的实现体在 bootstrap.c 中。试想一下 seL4 系统的启动过程，当用户态被摆上了东西，马上就开始有分页（Page）机制工作了，你是不是要把相应已经在内存里的东西转换到页面里面去,这就要求首先有合适的内存虚实映射，然后是虚拟内存地址管理机制要知道你已经用了哪些内存，对 seL4 来说，管理句柄及句柄空间，你也得处理一下吧？很多工作要做的。

11.1.4 bootstrap 函数

bootstrap_configure_virtual_pool

```
void bootstrap_configure_virtual_pool(allocman_t *alloc, void *vstart, uint32_t
vsize, seL4_CPtr pd);
```

功能：

* Every allocation manager created by these bootstrapping functions has a dual_pool
 * as its memory manager. For the purposes of bootstrapping only the fixed pool (as
 * passed to the boot strapping functions) is used. If you want to use a virtual
pool
 * (you almost certainly do so you don't run out of memory or have a stupidly large
 * static pool) then this function will initial the virtual pool after the fact.
The
 * reason for this ordering is that it is expected that the creation of a vspace
manager
 * might expect an allocator, which you won't have yet if you are boot strapping.
 *
 * Note that there is no protection against calling this function multiple times
or
 * from trying to call it on an allocman that does not have a dual_pool as its
underlying
 * memory manager. DO NOT FUCK IT UP

参数：

- alloc Allocman whose memory manager to configure
- vstart Start of a virtual address range that will be allocated from.
- vsize Size of the virtual address range
- pd Page directory to invoke when mapping frames/page tables

返回值：

（无）

bootstrap_use_bootinfo

```
/**
 * Simplest bootstrapping method that uses all the information in seL4_BootInfo
 * assumes you are the rootserver. This keeps using whatever cspace you are currently
in.
 *
 * @param bi BootInfo as passed to the rootserver
 * @param pool_size Size of the initial pool. See file comments for details
 * @param pool Initial pool. See file comments for details
 *
 * @return returns NULL on error
 */
allocman_t *bootstrap_use_bootinfo(seL4_BootInfo *bi, uint32_t pool_size, char
*pool);
```

bootstrap_new_1level_bootinfo

```
/**
 * Bootstraps using all the information in bootinfo, but switches to a new single
 * level cspace. All untypeds specified in bootinfo will be moved to the new cspace,
 * any other capabilities will be left in the old cspace. If you wish to refer to
the
 * boot cspace (most likely since it probably has capabilities you still want), then
 * a cspace description of the old cspace can also be returned.
 *
 * @param bi BootInfo as passed to the rootserver
 * @param cnode_size Number of slot bits (cnode_slots = 2^cnode_size) for the new
cnode
 * @param pool_size Size of the initial pool. See file comments for details
 * @param pool Initial pool. See file comments for details
 * @param old_cspace Optional location to store a description of the original cspace.
You
 * can free this memory back to the allocman when are done with it
 *
 * @return returns NULL on error
 */
allocman_t *bootstrap_new_1level_bootinfo(seL4_BootInfo *bi, int cnode_size,
uint32_t pool_size, char *pool, cspace_simple1level_t **old_cspace);
```

bootstrap_new_2level_bootinfo

```
/**
 * Bootstraps using all the information in bootinfo, but switches to a new two
 * level cspace. All untypeds specified in bootinfo will be moved to the new cspace,
 * any other capabilities will be left in the old cspace. If you wish to refer to
the
 * boot cspace (most likely since it probably has capabilities you still want), then
 * a cspace description of the old cspace can also be returned.
 *
 * @param bi BootInfo as passed to the rootserver
 * @param l1size Number of slot bits (l1_slots = 2^l1size) for the level 1 cnode
 * @param l2size Number of slot bits (l2_slots = 2^l2size) for the level 2 cnode
 * @param pool_size Size of the initial pool. See file comments for details
 * @param pool Initial pool. See file comments for details
 * @param old_cspace Optional location to store a description of the original cspace.
You
 * can free this memory back to the allocman when are done with it
 *
 * @return returns NULL on error
 */
allocman_t *bootstrap_new_2level_bootinfo(seL4_BootInfo *bi, int l1size, int
l2size, uint32_t pool_size, char *pool, cspace_simple1level_t **old_cspace);
```

allocman_add_simple_untypeds

```
/**
 * Give an allocator all the untyped memory that simple knows about.
 *
 * This assumes that all the untyped caps are currently as simple thinks they are.
 * If there have been any cspace reshuffles simple will not give allocman useable
information
 */
int allocman_add_simple_untypeds(allocman_t *alloc, simple_t *simple);
```

bootstrap_new_2level_simple

```
/**
 * Bootstraps using all the information provided by simple, but switches to a new
```

```
two
 * level cspace. All capabilities specified by simple will be moved to the new cspace.
All untypeds specified by simple are given to the allocator
 *
 * @param simple simple pointer to the struct
 * @param l1size Number of slot bits (l1_slots = 2^l1size) for the level 1 cnode
 * @param l2size Number of slot bits (l2_slots = 2^l2size) for the level 2 cnode
 * @param pool_size Size of the initial pool. See file comments for details
 * @param pool Initial pool. See file comments for details
 *
 * @return returns NULL on error
 */
allocman_t *bootstrap_new_2level_simple(simple_t *simple, int l1size, int l2size,
uint32_t pool_size, char *pool);
```

bootstrap_use_current_simple

使用当前定义在 simple 内核抽象中的数据引导系统。同时使用 simple 中描述的句柄空间 CSpace、原始内存（untyped memory）。

参数：

- simple Pointer to simple interface, will not be retained
- pool_size 初始内存池大小
- pool 初始内存池

返回值：

出错则返回 NULL。

函数原型：

```
allocman_t *bootstrap_use_current_simple(simple_t *simple, uint32_t pool_size,
char *pool);
```

bootstrap_use_current_1level

使用当前单层句柄空间 CSpace 构造 allocman_t。

当系统引导成功后，你需要手动把原始内存加到返回的 allocman_t 上。

参数：

- root_cnode Location of the cnode that is the current cspace
- cnode_size Size in slot_bits of the current cnode
- start_slot First free slot in the current cspace
- end_slot Last free slot + 1 in the current cspace
- pool_size Size of the initial pool. See file comments for details

- pool Initial pool. See file comments for details

返回值：

出错返回 NULL

原型：

```
allocman_t *bootstrap_use_current_1level(seL4_CPtr root_cnode, int cnode_size, seL4_CPtr start_slot, seL4_CPtr end_slot, uint32_t pool_size, char *pool);
```

bootstrap_set_boot_cspace

```
/**
 * Provides a description of the boot cspace if you are doing a customized
 * bootstrapping. This MUST be set before using boostrap_new_[1|2]level
 *
 * @param bs Internal bootstrapping info as allocated/returned by {@link #bootstrap_create_info}
 * @param cspace CSpace that will be used for bootstrapping purposes. The cspace only needs to exist
 *  for as long as bootstrapping is happening, it will not be used afterwards
 * @param root_cnode Path to the root cnode of cspace. This is needed so that a cap to the old cspace
 *  can be provided in the new cspace
 *
 * @return returns 0 on success
 */
int bootstrap_set_boot_cspace(bootstrap_info_t *bs, cspace_interface_t cspace, cspacepath_t root_cnode);
```

bootstrap_add_untypeds

```
/**
 * Adds knowledge of untypeds to the bootstrapping information. These untypeds will
 * be moved to the new cspace and be given to the untyped manager  once bootstrapping
 * has completed.
 *
 * @param bs Internal bootstrapping info as allocated/returned by {@link #bootstrap_create_info}
 * @param num Number of untypeds to be added
 * @param uts Path to each of the untypeds
 * @param size_bits Size of each of the untypeds
 * @param paddr Optional physical address of each of the untypeds
```

```
 *
 * @return returns 0 on success
 */
int bootstrap_add_untypeds(bootstrap_info_t *bs, int num, cspacepath_t *uts,
uint32_t *size_bits, uint32_t *paddr);
```

bootstrap_add_untypeds_from_bootinfo

```
/**
 * Adds knowledge of all the untypeds of bootinfo to the bootstrapper. These will
 * be moved to the new cspace and given to the untyped manager once bootstrapping
has
 * completed
 *
 * @param bs Internal bootstrapping info as allocated/returned by {@link
#bootstrap_create_info}
 * @param bi BootInfo as passed to the rootserver
 *
 * @return returns 0 on success
 */
int bootstrap_add_untypeds_from_bootinfo(bootstrap_info_t *bs, seL4_BootInfo
*bi);
```

bootstrap_new_1level

```
/**
 * Completes bootstrapping into a new single level cspace.
 *
 * @param info Internal bootstrapping info as allocated/returned by {@link
#bootstrap_create_info}
 * @param cnode_size Size in slot bits of new cspace
 * @param tcb Path to the TCB of the current thread, need to perform an invocation
of seL4_TCB_SetSpace
 * @param pd Path to the PD of the current thread. This is needed to work around
seL4 restriction that
 *  requires the address space be set at the same time as the cspace
 * @param oldroot Optional location to store a path to a cnode that is root cnode
given in {@link #bootstrap_set_boot_cspace}
 *
 * @return returns NULL on error
 */
```

```
allocman_t *bootstrap_new_1level(bootstrap_info_t *info, int cnode_size,
cspacepath_t tcb, cspacepath_t pd, cspacepath_t *oldroot);
```

bootstrap_new_2level

```
/**
 * Completes bootstrapping into a new two level cspace.
 *
 * @param info Internal bootstrapping info as allocated/returned by {@link
#bootstrap_create_info}
 * @param l1size Number of slot bits (l1_slots = 2^l1size) for the level 1 cnode
 * @param l2size Number of slot bits (l2_slots = 2^l2size) for the level 2 cnode
 * @param tcb Path to the TCB of the current thread, need to perform an invocation
of seL4_TCB_SetSpace
 * @param pd Path to the PD of the current thread. This is needed to work around
seL4 restriction that
 *  requires the address space be set at the same time as the cspace
 * @param oldroot Optional location to store a path to a cnode that is root cnode
given in {@link #bootstrap_set_boot_cspace}
 *
 * @return returns NULL on error
 */
allocman_t *bootstrap_new_2level(bootstrap_info_t *info, int l1size, int l2size,
cspacepath_t tcb, cspacepath_t pd, cspacepath_t *oldroot);
```

bootstrap_create_info

```
/**
 * This function starts bootstrapping the system, and then 'breaks out' and
 * allows you to give a description of the boot cspace as well as provide any
 * untypeds. A new 1 or 2 level cspace can then be created.
 *
 * @param pool_size Size of the initial pool. See file comments for details
 * @param pool Initial pool. See file comments for details
 *
 * @return returns NULL on error
 */
bootstrap_info_t *bootstrap_create_info(uint32_t pool_size, char *pool);
```

bootstrap_create_allocman

```
/**
 * Creates an empty allocman from a starting pool. The returned allocman will not
 * have an attached cspace or utspace. This function provides the ultimate flexibility
 * in how you can boot strap the system (read: this does basically nothing for you).

 * @param pool_size Size of the initial pool. See file comments for details
 * @param pool Initial pool. See file comments for details
 *
 * @return returns NULL on error
 */
allocman_t *bootstrap_create_allocman(uint32_t pool_size, char *pool);
```

11.2 libsel4vka

CSpace（Capability Space）是一块内存区，这块内存里面存的是内核要用到的数据结构，这块内存是由用户态程序申请来的，里面放的数据是不可以被用户态程序直接操作的，这就造成一个复杂的语义，于是在原始 API 之上提供了这样面向编程的 API 函数。

vka，虚拟内存内存管理器，virtual kernel allocator

下面程序定义于$/libs/libsel4vka/include/vka/object.h，用这些 API，使得申请内核对象变得容易。

```
/*
 * A wrapper to hold all the allocation information for an 'object'
 * An object here is just combination of cptr and untyped allocation
 * The type and size of the allocation is also stored to make free
 * more convenient.
 */

typedef struct vka_object {
    seL4_CPtr cptr;
    uint32_t ut;
    seL4_Word type;
    seL4_Word size_bits;
} vka_object_t;

/*
```

```
 * Generic object allocator used by functions below, can also be used directly
 */
static inline int vka_alloc_object(vka_t *vka, seL4_Word type, seL4_Word size_bits, vka_object_t *result)
{
    seL4_CPtr cptr;
    uint32_t ut;
    int error;
    cspacepath_t path;
    assert(vka);
    assert(result);
    if ( (error = vka_cspace_alloc(vka, &cptr)) != 0) {
        return error;
    }
    vka_cspace_make_path(vka, cptr, &path);
    if ( (error = vka_utspace_alloc(vka, &path, type, size_bits, &ut)) != 0) {
        fprintf(stderr, "Failed to allocate object of size %lu, error %d\n", BIT(size_bits), error);
        vka_cspace_free(vka, cptr);
        return error;
    }
    result->cptr = cptr;
    result->ut = ut;
    result->type = type;
    result->size_bits = size_bits;
    return 0;
}

static inline seL4_CPtr vka_alloc_object_leaky(vka_t *vka, seL4_Word type, seL4_Word size_bits)
{
    vka_object_t result = {.cptr = 0, .ut = 0, .type = 0, size_bits = 0};
    return vka_alloc_object(vka, type, size_bits, &result) == -1 ? 0 : result.cptr;
}

static inline void vka_free_object(vka_t *vka, vka_object_t *object)
{
    cspacepath_t path;
    assert(vka);
    assert(object);
    vka_cspace_make_path(vka, object->cptr, &path);
    /* ignore any errors */
    seL4_CNode_Delete(path.root, path.capPtr, path.capDepth);
    vka_cspace_free(vka, object->cptr);
```

```
    vka_utspace_free(vka, object->type, object->size_bits, object->ut);
}
```

11.2.1 申请一个 Slot

- cspace_alloc

```
typedef int (*vka_cspace_alloc_fn)(void *data, seL4_CPtr *res);
```

功能：

在 CSpace 中申请一个 Slot

参数：

[输入] data 申请器标识 cookie

[输出] res 存有 Slot 的指向 cptr 的指针

返回值：

成功则返回 0

- cspace_make_path

```
typedef void (*vka_cspace_make_path_fn)(void *data, seL4_CPtr slot, cspacepath_t *res);
```

功能：

把一个指向 CSpace 中的内存的指针 cptr 转换为一个 cspacepath，以便在 Untyped_Retype 之类的地方应用。

参数：

[输入] data 申请器标识 cookie

[输入] slot 一个 Slot 指针

[输出] res 填好内容的存有 cspacepath 的指针

返回值：

（无）

- cspace_free

```
typedef void (*vka_cspace_free_fn)(void *data, seL4_CPtr slot);
```

功能：

释放一个 Slot。

参数：

[输入] data 申请器标识 cookie

[输入] slot 一个 Slot 指针

返回值：

（无）

11.2.2 申请原始内存

UserTypeSpace，用户态使用内存申请。

- utspace_alloc

```
typedef int (*vka_utspace_alloc_fn)(void *data, const cspacepath_t *dest, seL4_Word type, seL4_Word size_bits, uint32_t *res);
```

功能：

往一个对象中申请一块原始内存。

申请一块虚拟内存，把它映射为物理内存，类似于 Linux 的 mmap()，这块内存总得托管到一个地方吧？那块地方就是 CSpace 中的句柄。

参数：

[输入] data 申请器标识 cookie

[输入] dest 一个存放申请来的对象的句柄的 cspacepath_t 空间

[输入] type 申请的 seL4 对象类型，与传给 Untyped_Retype 的一致

[输入] size_bits 申请的 seL4 对象大小，与传给 Untyped_Retype 的一致

[输出] res 指针

返回值：

成功则返回 0

- utspace_free

```
typedef void (*vka_utspace_free_fn)(void *data, seL4_Word type, seL4_Word size_bits, uint32_t target);
```

功能：

释放一块原始内存。

这块原始内存中的句柄都释放了？

参数：

[输入] data 申请器标识 cookie

[输入] type 申请的 seL4 对象类型，与传给 Untyped_Retype 的一致

[输入] size_bits 申请的 seL4 对象大小，与传给 Untyped_Retype 的一致

[输出] target 指针，utspace_alloc()返回的那个

返回值：

（无）

- utspace_paddr

```
typedef uintptr_t (*vka_utspace_paddr_fn)(void *data, uint32_t target, seL4_Word type, seL4_Word size_bits);
```

功能：

一个对象的物理内存。

参数：

[输入] data 申请器标识 cookie

[输入] target 指针，utspace_alloc()返回的那个

[输入] type 申请的 seL4 对象类型，与传给 Untyped_Retype 的一致

[输入] size_bits 申请的 seL4 对象大小，与传给 Untyped_Retype 的一致

返回值：

成功则返回物理地址 paddr，失败返回 NULL

11.3 libsel4vspace

虚拟内存管理。

导出头文件：vspace.h

- vspace_new_stack

```
void *vspace_new_stack(vspace_t *vspace);
```

功能：

创建一个栈

参数：

[输入] vspace

返回值：

成功则返回栈顶指针，失败返回 NULL

- vspace_free_stack

```
void vspace_free_stack(vspace_t *vspace, void *stack_top);
```

功能：

释放一个栈，只释放虚拟资源，不释放物理资源

参数：

[输入] vspace

[输入] stack_top 栈顶指针

返回值：

（无）

12 libsel4utils

seL4 的 API，逻辑太低，其中甚至看不到操作系统的影子。操作系统是什么？操作系统分两层：下层，对设备进行抽象、管理；上层，负责提供程序模型的支持。

为了把 seL4 包装得稍象个操作系统，需要更强功能的 API，于是有了 libsel4utils。

libsel4utils 提供了一个象操作系统（OS-like）的一些程序功能，方便在 seL4 上开发操作系统，或者写一些 seL4 实验程序。虽然这些程序尽量少地做模型，但还是有很多约定的。

如果你设计的操作系统模型与传统的 Linux 这样的操作系统有巨大差异,这些程序也就只能供你能考一下。

libsel4utils 提供了下列库：

- threads
- processes
- elf loading
- virtual memory management
- stack switching
- debugging tools

这里没有约定使用哪个内存申请与管理器,因为只要是实现了 seL4 vka 的都可以使用，但建议使用 libsel4allocman。

依赖：

本库依赖于： libsel4vka、libsel4vspace、libutils、libelf、libcpio、libsel4。

头文件：

include/sel4utils

- client_server_vspace.h -- a virtual address space that proxies calls between two different vspaces
- elf.h -- elf loading.
- mapping.h -- page mapping.
- process.h -- process creation, deletion.
- profile.h -- profiling.
- sel4_debug.h -- for printing seL4 error codes.
- stack.h -- switch to a newly allocated stack.
- thread.h -- threads (kernel threads) creation, deletion.
- util.h -- includes utilities from libutils.
- vspace.h -- virtual memory management (implements vspace interface)
- vspace_internal.h -- virtual memory management internals, for hacking the above.

arch_include/sel4utils

- util.h -- utils to assist in writing arch independent code.

配置信息：

- SEL4UTILS_STACK_SIZE -- the default stack size to use for processes and threads.
- SEL4UTILS_CSPACE_SIZE_BITS -- the default cspace size for new processes (threads use the current cspace).

12.1 进程 Process

从下面的源码中看一下对进程（Process）的定义。

```
typedef struct object_node object_node_t;

struct object_node {
    vka_object_t object;
    object_node_t *next;
};

typedef struct {
    vka_object_t pd;
    vspace_t vspace;
    sel4utils_alloc_data_t data;
    vka_object_t cspace;
    uint32_t cspace_size;
    uint32_t cspace_next_free;
    sel4utils_thread_t thread;
    vka_object_t fault_endpoint;
    void *entry_point;
    uintptr_t sysinfo;
    object_node_t *allocated_object_list_head;
    /* if the elf wasn't loaded into the address space, this describes the regions.
     * this permits lazy loading / copy on write / page sharing / whatever crazy
thing
     * you want to implement */
    int num_elf_regions;
    sel4utils_elf_region_t *elf_regions;
} sel4utils_process_t;

/* sel4utils processes start with some caps in their cspace.
 * These are the caps
```

```
 */
enum sel4utils_cspace_layout {
    /*
     * The root cnode (with appropriate guard)
     */
    SEL4UTILS_CNODE_SLOT = 1,
    /* The slot on the cspace that fault_endpoint is put if
     * sel4utils_configure_process is used.
     */
    SEL4UTILS_ENDPOINT_SLOT = 2,
};

typedef struct {
    /* should we handle elf logic at all? */
    bool is_elf;
    /* if so what is the image name? */
    char *image_name;
    /* Do you want the elf image preloaded? */
    bool do_elf_load;

    /* otherwise what is the entry point and sysinfo? */
    void *entry_point;
    uintptr_t sysinfo;

    /* should we create a default single level cspace? */
    bool create_cspace;
    /* if so how big ? */
    int one_level_cspace_size_bits;

    /* otherwise what is the root cnode ?*/
    /* Note if you use a custom cspace then
     * sel4utils_copy_cap_to_process etc will not work */
    vka_object_t cnode;

    /* do you want us to create a vspace for you? */
    bool create_vspace;
    /* if not what is the page dir, and what is the vspace */
    vspace_t *vspace;
    vka_object_t page_dir;

    /* if so, is there a regions you want left clear?*/
    sel4utils_elf_region_t *reservations;
    int num_reservations;
```

```
    /* do you want a fault endpoint created? */
    bool create_fault_endpoint;
    /* otherwise what is it */
    vka_object_t fault_endpoint;

    int priority;
#ifndef CONFIG_KERNEL_STABLE
    seL4_CPtr asid_pool;
#endif
} sel4utils_process_config_t;
```

一个虚拟地址空间，一个 ELF 文件，把它们对应起来，成了进程。

12.2 线程 Thread

从下面的源码中看一下对线程（Thread）的定义。

thread.h

```
typedef struct sel4utils_thread {
    vka_object_t tcb;
    void *stack_top;
    seL4_CPtr ipc_buffer;
    seL4_Word ipc_buffer_addr;
} sel4utils_thread_t;
```

12.2.1 sel4utils_configure_thread

功能：

配置一个线程，为其申请需要的资源。

参数：

- vka initialised vka to allocate objects with
- parent vspace structure of the thread calling this function, used for temporary mappings
- alloc initialised vspace structure to allocate virtual memory with
- fault_endpoint endpoint to set as the threads fault endpoint. Can be 0.
- priority seL4 priority for the thread to be scheduled with.
- cspace the root of the cspace to start the thread in
- cspace_root_data data for cspace access

- 如果函数执行成功，则参数 res 中数据将被初始化，其数据结构为：
  sel4utils_thread_t

返回值：

return 0 on success, -1 on failure. Use CONFIG_DEBUG to see error messages.

原型：

```
int sel4utils_configure_thread(vka_t *vka, vspace_t *parent, vspace_t *alloc, seL4_CPtr fault_endpoint, uint8_t priority, seL4_CNode cspace, seL4_CapData_t cspace_root_data, sel4utils_thread_t *res);
```

12.2.2 sel4utils_start_thread

功能：

起动一个线程，为其申请需要的资源。

第三个参数将要成为线程的 IPC 缓存（IPC Buffer），ARM 上存于寄存器 r2，IA-32 上存于栈上。

参数：

- thread　　thread data structure that has been initialised with sel4utils_configure_thread
- entry_point the address that the thread will start at，线程将要执行的第一条指令的地址

  注意：IA-32 平台，entry_point 是通过栈传给这个函数体的，这就需要入口点为一个函数，如下面这个_start 符号可以是入口点，跳到_start 这个符号代表的地址是可以工作的。

```
void _start(int argc, char **argv) {
    int ret = main(argc, argv);
    exit(ret);
}
```

  跳到下面这样的 start 符号是不工作的。

```
_start:
    call main
```

  因为 call 指令压入了额外的参数（返回值），如果一定要一个汇编的桩（stub），应该先让栈弹出一个整数。可以这样写：

```
_start:
    popl %eax
    call main
```

  这种说法对 ARM 平台无效，因为 ARM 平台是通过寄存器传这个参数的。
- arg0　　a pointer to the arguments for this thread. User decides the protocol.
- arg1　　another pointer. User decides the protocol. Note that there are two args here to easily support C standard: int main(int argc, char **argv).
- resume　　1 to start the thread immediately, 0 otherwise.

返回值：

return 0 on success, -1 on failure.

原型：

```
int sel4utils_start_thread(sel4utils_thread_t *thread, void *entry_point, void *arg0, void *arg1, int resume);
```

12.2.3 sel4utils_clean_up_thread

功能：

释放这个线程所持有的所有资源 Release any resources used by this thread. The thread data structure will not be usable until sel4utils_thread_configure is called again.

参数：

- vka the vka interface that this thread was initialised with
- alloc the allocation interface that this thread was initialised with
- thread the thread structure that was returned when the thread started

原型：

```
void sel4utils_clean_up_thread(vka_t *vka, vspace_t *alloc, sel4utils_thread_t *thread);
```

12.2.4 sel4utils_start_fault_handler

功能：

开始一个带异常处理的线程。Start a fault handling thread that will print the name of the thread that faulted as well as debugging information.

参数：

- fault_endpoint the fault_endpoint to wait on 异常处理端点
- vka allocator
- vspace vspace (this library must be mapped into that vspace).
- prio the priority to run the thread at (recommend highest possible)
- cspace the cspace that the fault_endpoint is in
- data the cspace_data for that cspace (with correct guard)
- name the name of the thread to print if it faults
- thread the thread data structure to populate

返回值：

return 0 on success.

原型：

```
int sel4utils_start_fault_handler(seL4_CPtr fault_endpoint, vka_t *vka, vspace_t
*vspace, uint8_t prio, seL4_CPtr cspace, seL4_CapData_t data, char *name,
sel4utils_thread_t *res);
```

12.2.5 sel4utils_print_fault_message

功能：

打印一个线程错误信息

参数：

- tag the message info tag delivered by the fault.
- name thread name

原型：

```
void sel4utils_print_fault_message(seL4_MessageInfo_t tag, char *name);
```

12.2.6 sel4utils_get_tcb

功能：

取线程控制块 TCB

原型：

```
static inline seL4_TCB
sel4utils_get_tcb(sel4utils_thread_t *thread)
{
    return thread->tcb.cptr;
}
```

12.2.7 sel4utils_suspend_thread

功能：

挂起线程。

原型：

```
static inline int
sel4utils_suspend_thread(sel4utils_thread_t *thread)
{
    return seL4_TCB_Suspend(thread->tcb.cptr);
}
```

13 libsel4simple

一个简单的 seL4 编程接口抽象，与用户层是如何装入（load）的无关。

看 simple.h，了解一下这个简单抽象层都有什么功能。

```
typedef struct simple_t {
    void *data;
    simple_get_frame_cap_fn frame_cap;
    simple_get_frame_mapping_fn frame_mapping;
    simple_get_frame_info_fn frame_info;
    simple_get_IRQ_control_fn irq;
    simple_ASIDPool_assign_fn ASID_assign;
    simple_get_IOPort_cap_fn IOPort_cap;
    simple_get_cap_count_fn cap_count;
    simple_get_nth_cap_fn nth_cap;
    simple_get_init_cap_fn init_cap;
    simple_get_cnode_size_fn cnode_size;
    simple_get_untyped_count_fn untyped_count;
    simple_get_nth_untyped_fn nth_untyped;
    simple_get_userimage_count_fn userimage_count;
    simple_get_nth_userimage_fn nth_userimage;
#ifdef CONFIG_IOMMU
    simple_get_iospace_fn iospace;
#endif
    simple_print_fn print;
} simple_t;
```

这个数据结构由这个函数负责填充：

```
void simple_default_init_bootinfo(simple_t *simple, seL4_BootInfo *bi);
```

13.1 simple_get_frame_cap_fn

功能：

取得物理帧的句柄到指定的位置。

物理帧就是硬件上的控制内存区域，它是在硬件启动时，如 PCI 设备检查时得到，了解哪些内存区域对应哪个设备，才能实现对该设备的控制。

参数：

- data cookie for the underlying implementation，seL4_BootInfo *，系统启动信息
- paddr， 页对齐的物理地址
- size_bits，区域大小，比特位数
- path，放句柄的地方

原型：

typedef seL4_Error (*simple_get_frame_cap_fn)(void *data, void *paddr, int size_bits, cspacepath_t *path);

```
typedef enum {
    seL4_NoError = 0,
    seL4_InvalidArgument,
    seL4_InvalidCapability,
    seL4_IllegalOperation,
    seL4_RangeError,
    seL4_AlignmentError,
    seL4_FailedLookup,
    seL4_TruncatedMessage,
    seL4_DeleteFirst,
    seL4_RevokeFirst,
    seL4_NotEnoughMemory,
} seL4_Error;
```

13.2 simple_get_frame_mapping_fn

功能：

Request mapped address to a region of physical memory.
Note: This function will only return the mapped virtual address that it knows about. It does not do any mapping its self nor can it guess where mapping functions are going to map.

参数：

- data cookie for the underlying implementation
- page aligned physical address
- size of the region in bits
- Returns the vritual address to which this physical address is mapped or NULL if frame is unmapped

原型：

typedef void *(*simple_get_frame_mapping_fn)(void *data, void *paddr, int size_bits);

13.3 simple_get_frame_info_fn

功能：

取得物理帧的数据。

This function will only return the mapped virtual address that it knows about. It does not do any mapping its self nor can it guess where mapping functions are going to map.

参数：

- data cookie for the underlying implementation
- page aligned physical address for the frame
- size of the region in bits
- cap to the frame gets set. Will return the untyped cap unless the underlying implementation has access to the frame cap. Check with implementation but it should be a frame cap if and only if a vaddr is returned.
- (potentially) the offset within the untyped cap that was returned
- Returns the vritual address to which this physical address is mapped or NULL if frame is unmapped

原型：

typedef void *(*simple_get_frame_info_fn)(void *data, void *paddr, int size_bits, seL4_CPtr *cap, seL4_Word *ut_offset);

13.4 simple_get_IRQ_control_fn

功能：

同：seL4_IRQControl_Get(seL4_CapIRQControl, irq, root, index, depth);

取指定中断的 IRQControl 句柄。

参数：

- data，（无用参数）
- irq，中断
- cnode， the CNode in which to put this cap
- index，the index within the CNode to put cap
- depth，Depth of index

原型：

typedef seL4_Error (*simple_get_IRQ_control_fn)(void *data, int irq, seL4_CNode cnode, seL4_Word index, uint8_t depth);

13.5 simple_get_IOPort_cap_fn

功能：

- Request a cap to the IOPorts on IA32
- data cookie for the underlying implementation
- start port number that a cap is needed to
- end port number that a cap is needed to

原型：

```
typedef seL4_CPtr (*simple_get_IOPort_cap_fn)(void *data, uint16_t start_port,
uint16_t end_port);
```

13.6 simple_ASIDPool_assign_fn

功能：

把 vpsace 设到当前线程的 ASID 池（ASID pool）

参数：

- data cookie for the underlying implementation
- vspace to assign

原型：

```
typedef seL4_Error (*simple_ASIDPool_assign_fn)(void *data, seL4_CPtr vspace);
```

13.7 simple_get_cap_count_fn

功能：

取得可以访问的设备地址区间的句柄数量

参数：

- data cookie for the underlying implementation，seL4_BootInfo *，系统启动信息

原型：

```
typedef int (*simple_get_cap_count_fn)(void *data);
```

13.8 simple_get_nth_cap_fn

功能：

取得第 N 个初始地址区间（描述在 bootinfo.h）的句柄。

参数：

- data cookie for the underlying implementation
- the nth starting at 0

原型：

```
typedef seL4_CPtr (*simple_get_nth_cap_fn)(void *data, int n);
```

13.9 simple_get_init_cap_fn

功能：

取得第 N 个初始地址区间（描述在 bootinfo.h）。

通常情况下就是直接返回 cap。

参数：

- @param data for the underlying implementation
- @param the value of the enum matching in bootinfo.h

原型：

```
typedef seL4_CPtr (*simple_get_init_cap_fn)(void *data, seL4_CPtr cap);
```

13.10 simple_get_cnode_size_fn

功能：

取 CNode 大小

参数：

- data cookie for the underlying implementation，seL4_BootInfo *，系统启动信息

原型：

```
typedef uint8_t  (*simple_get_cnode_size_fn)(void *data);
```

13.11 simple_get_untyped_count_fn

功能：

取得原始内存句柄的数量

参数：

- data cookie for the underlying implementation, seL4_BootInfo *, 系统启动信息

原型：

```
typedef int (*simple_get_untyped_count_fn)(void *data);
```

13.12 simple_get_nth_untyped_fn

功能：

取得第 N 块原始内存的信息

参数：

- data cookie for the underlying implementation, seL4_BootInfo *, 系统启动信息
- the nth starting at 0
- the size of the untyped for the returned cap
- the physical address of the returned cap

原型：

```
typedef seL4_CPtr (*simple_get_nth_untyped_fn)(void *data, int n, uint32_t
*size_bits, uint32_t *paddr);
```

13.13 simple_get_userimage_count_fn

功能：

取得 userimage 的数量

seL4 起动（Boot）时，启动信息中有这样的信息：

```
typedef struct {
    seL4_Word start; /* first CNode slot position OF region */
    seL4_Word end;   /* first CNode slot position AFTER region */
} seL4_SlotRegion;
```

即每个应用程序映像的标识区间。

参数：

- data cookie for the underlying implementation, seL4_BootInfo *, 系统启动信息

原型：

```
typedef int (*simple_get_userimage_count_fn)(void *data);
```

13.14 simple_get_nth_userimage_fn

功能：

取得第 N 个 userimage

seL4 起动（Boot）时，启动信息中有这样的信息：

```
typedef struct {
    seL4_Word start; /* first CNode slot position OF region */
    seL4_Word end;   /* first CNode slot position AFTER region */
} seL4_SlotRegion;
```

即每个应用程序映像的标识区间。

参数：

- data cookie for the underlying implementation，seL4_BootInfo *，系统启动信息
- the nth starting at 0

原型：

```
typedef seL4_CPtr (*simple_get_nth_userimage_fn)(void *data, int n);
```

13.15 simple_get_iospace_fn

功能：

（本函数只有在 IOMMU 有效时才被定义，#ifdef CONFIG_IOMMU）

Get the IO space capability for the specified pci device and domain ID

参数：

- data cookie for the underlying implementation，seL4_BootInfo *，系统启动信息
- domainID domain ID to request
- deviceID PCI device ID
- path Path to where to put this cap

原型：

```
typedef seL4_Error (*simple_get_iospace_fn)(void *data, uint16_t domainID,
uint16_t deviceID, cspacepath_t *path);
```

14 libplatsupport

操作系统无关的驱动程序库，如： timers、serial、clocks 等。

```
$ tree arch_include/
arch_include/
├── arm
│   └── platsupport
│       ├── clock.h
│       ├── gpio.h
│       ├── i2c.h
│       ├── irq_combiner.h
│       ├── mux.h
│       └── spi.h
└── ia32
    └── platsupport
        └── arch
            └── tsc.h

$ tree include/
include/
└── platsupport
    ├── chardev.h
    ├── io.h
    ├── serial.h
    └── timer.h
```

14.1 基础知识

- SPI 是串行外设接口（Serial Peripheral Interface）的缩写。SPI，是一种高速的，全双工，同步的通信总线，并且在芯片的管脚上只占用四根线。
  http://baike.baidu.com/view/245026.htm

- 数据选择器 MUX。在电子技术（特别是数字电路）中，数据选择器（英语：multiplexer，简称：MUX），或称多路复用器，是一种可以从多个输入信号中选择一个信号进行输出的器件。
  http://zh.wikipedia.org/zh/%E6%95%B0%E6%8D%AE%E9%80%89%E6%8B%A9%E5%99%A8

- IRQ combiner

- I2C（Inter－Integrated Circuit）总线是由 PHILIPS 公司开发的两线式串行总线，用于连接微控制器及其外围设备。是微电子通信控制领域广泛采用的一种总线标准。它是同步通信的一种特殊形式，具有接口线少，控制方式简单，器件封装形式小，通信速率较高等优点。

- General Purpose Input Output （通用输入/输出）简称为 GPIO，或总线扩展器，利用工业标准 I2C、SMBus 或 SPI 接口简化了 I/O 口的扩展。当微控制器或芯片组没有足够的 I/O 端口，或当系统需要采用远端串行通信或控制时，GPIO 产品能够提供额外的控制和监视功能。

- The Time Stamp Counter (TSC) is a 64-bit register present on all x86 processors since the Pentium. It counts the number of cycles since reset. The instruction RDTSC returns the TSC in EDX:EAX. In x86-64 mode, RDTSC also clears the higher 32 bits of RAX and RDX. Its opcode is 0F 31.[1] Pentium competitors such as the Cyrix 6x86 did not always have a TSC and may consider RDTSC an illegal instruction. Cyrix included a Time Stamp Counter in their MII.

15 简易 seL4 编程

本章通过一个简易的 seL4 程序，试图建立一个示例程序。

vspace_reserve_range
vka_alloc_object_leaky
vka_alloc_page_table_leaky
seL4_ARM_Page_Map
seL4_ARM_PageTable_Map

索引

参考文献

1. seL4 Reference Manual API version 1.3
2. seL4 Reference Manual
3. The Clustered Multikernel: An Approach to Formal Verification of Multiprocessor Operating-System Kernels