**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 6](#_Toc448656522)

[1.1 基础概念 6](#_Toc448656523)

[1.1 1 内核对象（Kernel Objects） 6](#_Toc448656524)

[1.1.2 句柄（Capabilities） 6](#_Toc448656525)

[1.1.3 虚拟地址空间（Virtual Address Spaces） 7](#_Toc448656526)

[1.1.4 线程（Threads） 8](#_Toc448656527)

[1.1.5 进程间通信（Inter-Process Communication，IPC） 8](#_Toc448656528)

[1.1.6 设备驱动与中断（Device Drivers and IRQs） 8](#_Toc448656529)

[1.1.7 抢占（Preemption） 8](#_Toc448656530)

[1.2 零碎信息 9](#_Toc448656531)

[1.2. 1 L4操作系统综述 9](#_Toc448656532)

[1.2. 2 微内核操作系统分代 11](#_Toc448656533)

[2 内核服务与内核对象 13](#_Toc448656534)

[2.1 基于句柄的访问控制 13](#_Toc448656535)

[2.2 系统调用 14](#_Toc448656536)

[2.3 内核对象 15](#_Toc448656537)

[2.4 内核内存申请 16](#_Toc448656538)

[2.4.1 内存重用 16](#_Toc448656539)

[3 句柄空间 18](#_Toc448656540)

[3.1 句柄与CSpace管理 19](#_Toc448656541)

[3.1.1 CSpace创建 19](#_Toc448656542)

[3.1.2 CNode方法（CNode Methods） 19](#_Toc448656543)

[3.1.3 句柄权限（Capability Rights） 20](#_Toc448656544)

[3.1.4 句柄继承树（Capability Derivation Tree） 21](#_Toc448656545)

[3.2 删除、撤销、回收（Deletion, Revocation, and Recycling） 22](#_Toc448656546)

[3.3 CSpace编址（CSpace Addressing） 23](#_Toc448656547)

[3.3.1 句柄地址查找Capability Address Lookup） 25](#_Toc448656548)

[3.3.2 句柄编址（Addressing Capabilities） 27](#_Toc448656549)

[3.4 查找失败描述（Lookup Failure Description） 28](#_Toc448656550)

[3.4.1 Invalid Root 28](#_Toc448656551)

[3.4.2 Missing Capability 28](#_Toc448656552)

[3.4.3 Depth Mismatch 29](#_Toc448656553)

[3.4.4 Guard Mismatch 29](#_Toc448656554)

[3.5 句柄调用Capability Invocation 29](#_Toc448656555)

[4 进程间通信IPC 31](#_Toc448656556)

[4.1 消息寄存器Message Register 31](#_Toc448656557)

[4.2 同步端点Synchronous Endpoint 36](#_Toc448656558)

[4.2.1 端点标记Endpoint Badge 36](#_Toc448656559)

[4.2.2 句柄传输Capability Transfer 37](#_Toc448656560)

[4.2.3 错误Errors 37](#_Toc448656561)

[4.3 异步端点Asynchronous Endpoint 38](#_Toc448656562)

[4.3.1 异步端点标记Asynchronous Endpoint Badges 38](#_Toc448656563)

[4.4 同步端点与IPC（Synchronous Endpoints and IPC） 39](#_Toc448656564)

[4.4 系统功能调用（Sysyem Calls） 40](#_Toc448656565)

[5 通知 42](#_Toc448656566)

[5.1 通知对象Notiﬁcation Objects 42](#_Toc448656567)

[5.2 Signalling, Polling and Waiting 42](#_Toc448656568)

[5.3 绑定通知Binding Notiﬁcations 43](#_Toc448656569)

[6 线程及其执行 44](#_Toc448656570)

[6.1 Threads（线程） 44](#_Toc448656571)

[6.1.1 线程创建Thread Creation 45](#_Toc448656572)

[6.1.2 线程停止Thread Deactivation 46](#_Toc448656573)

[6.1.3 调度Scheduling 46](#_Toc448656574)

[6.1.4 异常Exceptions 47](#_Toc448656575)

[6.1.5寄存器读写方法中的消息结构 47](#_Toc448656576)

[6.2 故障 Faults 48](#_Toc448656577)

[6.2.1 句柄故障Capability Faults 49](#_Toc448656578)

[6.2.2 不明系统调用Unknown Syscall 49](#_Toc448656579)

[6.2.3 用户异常User Exception 51](#_Toc448656580)

[6.2.4 缺页VM Fault 52](#_Toc448656581)

[6.3 域（Domains） 52](#_Toc448656582)

[6.4 线程控制块TCB 53](#_Toc448656583)

[7 地址空间与虚拟内存 55](#_Toc448656584)

[7.1 概述（Overview） 57](#_Toc448656585)

[7.2 内核对象（Objects） 57](#_Toc448656586)

[7.3 映射属性（Mapping Attributes） 60](#_Toc448656587)

[7.4 共享内存（Sharing Memory） 60](#_Toc448656588)

[7.5 页出错（Page Faults） 62](#_Toc448656589)

[8 硬件IO 63](#_Toc448656590)

[8.1 中断指派（Interrupt Delivery） 63](#_Toc448656591)

[8.2 IA-3特定I/O 64](#_Toc448656592)

[8.2.1 I/O端口（I/O Ports） 64](#_Toc448656593)

[8.2.2 I/O空间（I/O Space） 65](#_Toc448656594)

[9 系统启动过程 67](#_Toc448656595)

[9.1 初始线程环境（Initial Thread's Environment） 67](#_Toc448656596)

[9.2 起动信息帧（BootInfo Frame） 69](#_Toc448656597)

[9.3 引导用令行（Boot Command-line Arguments） 70](#_Toc448656598)

[9.4 多核起动（Multikernel Bootstrapping） 71](#_Toc448656599)

[10 编程接口API 73](#_Toc448656600)

[10.1 出错代码 73](#_Toc448656601)

[10.1.1 Invalid Argument 73](#_Toc448656602)

[10.1.2 Invalid Capability 73](#_Toc448656603)

[10.1.3 Illegal Operation 73](#_Toc448656604)

[10.1.4 Range Error 74](#_Toc448656605)

[10.1.5 Alignment Error 74](#_Toc448656606)

[10.1.6 Failed Lookup 74](#_Toc448656607)

[10.1.7 Delete First 74](#_Toc448656608)

[10.1.8 Revoke First 75](#_Toc448656609)

[10.1.9 Not Enough Memory 75](#_Toc448656610)

[10.2 基础系统API 75](#_Toc448656611)

[10.2.1 Send 76](#_Toc448656612)

[10.2.2 Recv 77](#_Toc448656613)

[10.2.3 Call 77](#_Toc448656614)

[10.2.4 Reply 77](#_Toc448656615)

[10.2.5 Polling Send 78](#_Toc448656616)

[10.2.6 Reply Recv 78](#_Toc448656617)

[10.2.7 NBRecv 79](#_Toc448656618)

[10.2.8 Yield 79](#_Toc448656619)

[10.2.9 Signal 79](#_Toc448656620)

[10.2.10 Wait 80](#_Toc448656621)

[10.2.11 Poll 81](#_Toc448656622)

[9.3 体系结构无关的对象方法 81](#_Toc448656623)

[10.3.1 CNode - Copy 84](#_Toc448656624)

[10.3.2 CNode - Delete 85](#_Toc448656625)

[10.3.3 CNode - Mint 85](#_Toc448656626)

[10.3.4 CNode - Move 86](#_Toc448656627)

[10.3.5 CNode - Mutate 87](#_Toc448656628)

[10.3.6 CNode - Recycle 87](#_Toc448656629)

[10.3.7 CNode - Revoke 88](#_Toc448656630)

[10.3.8 CNode - Rotate 88](#_Toc448656631)

[10.3.9 CNode - Save Caller 89](#_Toc448656632)

[10.3.10 Debug - Halt 90](#_Toc448656633)

[10.3.11 Debug - Put Character 90](#_Toc448656634)

[10.3.12 Debug - Snapshot 91](#_Toc448656635)

[10.3.13 Debug - CapIdentify 91](#_Toc448656636)

[10.3.14 Debug - Run 91](#_Toc448656637)

[10.3.15 Benchmark - ResetLog 92](#_Toc448656638)

[10.3.16 Benchmark - DumpLog 92](#_Toc448656639)

[10.3.17 Benchmark - LogSize 92](#_Toc448656640)

[10.3.18 DomainSet - Set 93](#_Toc448656641)

[10.3.19 IRQ Control - Get 94](#_Toc448656642)

[10.3.20 IRQ Handler - Acknowledge 94](#_Toc448656643)

[10.3.21 IRQ Handler - Clear 95](#_Toc448656644)

[10.3.22 IRQ Handler - Set Endpoint 95](#_Toc448656645)

[10.3.23 TCB - Configure 96](#_Toc448656646)

[10.3.24 TCB - Copy Registers 97](#_Toc448656647)

[10.3.25 TCB - Read Registers 97](#_Toc448656648)

[10.3.26 TCB - Resume 98](#_Toc448656649)

[10.3.27 TCB - Set IPC\_Buffer 99](#_Toc448656650)

[10.3.28 TCB - Set Priority 99](#_Toc448656651)

[10.3.29 TCB - Set Space 100](#_Toc448656652)

[10.3.30 TCB - Suspend 100](#_Toc448656653)

[10.3.31 TCB - Write Registers 101](#_Toc448656654)

[10.3.32 Untyped - Retype 101](#_Toc448656655)

[10.3.33 对象大小汇总 103](#_Toc448656656)

[10.4 IA-32特定对象方法（IA-32-Specific Object Methods） 104](#_Toc448656657)

[10.4.1 IA32 ASID Control - Make Pool 104](#_Toc448656658)

[10.4.2 IA32 ASID Pool - Assign 104](#_Toc448656659)

[10.4.3 IA32 IO Port - In 8 105](#_Toc448656660)

[10.4.4 IA32 IO Port - In 16 106](#_Toc448656661)

[10.4.5 IA32 IO Port - In 32 106](#_Toc448656662)

[10.4.6 IA32 IO Port - Out 8 107](#_Toc448656663)

[10.4.7 IA32 IO Port - Out 16 107](#_Toc448656664)

[10.4.8 IA32 IO Port - Out 32 108](#_Toc448656665)

[10.4.9 IA32 IO Page Table - Map 108](#_Toc448656666)

[10.4.10 IA32 Page - Map IO 109](#_Toc448656667)

[10.4.11 IA32 Page - Map 110](#_Toc448656668)

[10.4.12 IA32 Page - Remap 110](#_Toc448656669)

[10.4.13 IA32 Page - Unmap 111](#_Toc448656670)

[10.4.14 IA32 Page - Get Address 112](#_Toc448656671)

[10.4.15 IA32 Page Table - Map 112](#_Toc448656672)

[10.4.16 IA32 Page Table - Unmap 113](#_Toc448656673)

[10.5 ARM-Speci c Object Methods 113](#_Toc448656674)

[10.5.1 ARM ASID Control - Make Pool 113](#_Toc448656675)

[10.5.2 ARM ASID Pool - Assign 114](#_Toc448656676)

[10.5.3 ARM Page - Flush Caches 115](#_Toc448656677)

[10.5.4 ARM Page - Map 115](#_Toc448656678)

[10.5.5 ARM Page - Remap 116](#_Toc448656679)

[10.5.6 ARM Page - Unmap 117](#_Toc448656680)

[10.5.7 ARM Page - Get Address 117](#_Toc448656681)

[10.5.8 ARM Page Table - Map 118](#_Toc448656682)

[10.5.9 ARM Page Table - Unmap 118](#_Toc448656683)

[10 muslc c库 120](#_Toc448656684)

[11 内存相关的库 121](#_Toc448656685)

[11.1 libsel4allocman 121](#_Toc448656686)

[11.1.1 allocman数据结构 122](#_Toc448656687)

[11.1.2 allocman函数 124](#_Toc448656688)

[11.1.3 bootstrap数据结构 131](#_Toc448656689)

[11.1.4 bootstrap函数 135](#_Toc448656690)

[11.2 libsel4vka 142](#_Toc448656691)

[11.2.1 申请一个Slot 144](#_Toc448656692)

[11.2.2 申请原始内存 145](#_Toc448656693)

[11.3 libsel4vspace 146](#_Toc448656694)

[12 libsel4utils 147](#_Toc448656695)

[12.1 进程Process 148](#_Toc448656696)

[12.2 线程Thread 150](#_Toc448656697)

[12.2.1 sel4utils\_configure\_thread 150](#_Toc448656698)

[12.2.2 sel4utils\_start\_thread 151](#_Toc448656699)

[12.2.3 sel4utils\_clean\_up\_thread 152](#_Toc448656700)

[12.2.4 sel4utils\_start\_fault\_handler 152](#_Toc448656701)

[12.2.5 sel4utils\_print\_fault\_message 153](#_Toc448656702)

[12.2.6 sel4utils\_get\_tcb 153](#_Toc448656703)

[12.2.7 sel4utils\_suspend\_thread 153](#_Toc448656704)

[13 libsel4simple 154](#_Toc448656705)

[13.1 simple\_get\_frame\_cap\_fn 154](#_Toc448656706)

[13.2 simple\_get\_frame\_mapping\_fn 155](#_Toc448656707)

[13.3 simple\_get\_frame\_info\_fn 156](#_Toc448656708)

[13.4 simple\_get\_IRQ\_control\_fn 156](#_Toc448656709)

[13.5 simple\_get\_IOPort\_cap\_fn 157](#_Toc448656710)

[13.6 simple\_ASIDPool\_assign\_fn 157](#_Toc448656711)

[13.7 simple\_get\_cap\_count\_fn 157](#_Toc448656712)

[13.8 simple\_get\_nth\_cap\_fn 158](#_Toc448656713)

[13.9 simple\_get\_init\_cap\_fn 158](#_Toc448656714)

[13.10 simple\_get\_cnode\_size\_fn 158](#_Toc448656715)

[13.11 simple\_get\_untyped\_count\_fn 158](#_Toc448656716)

[13.12 simple\_get\_nth\_untyped\_fn 159](#_Toc448656717)

[13.13 simple\_get\_userimage\_count\_fn 159](#_Toc448656718)

[13.14 simple\_get\_nth\_userimage\_fn 160](#_Toc448656719)

[13.15 simple\_get\_iospace\_fn 160](#_Toc448656720)

[14 libplatsupport 161](#_Toc448656721)

[14.1 基础知识 161](#_Toc448656722)

[15 简易seL4编程 163](#_Toc448656723)

[索引 164](#_Toc448656724)

[参考文献 165](#_Toc448656725)

# 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，没有提供像传统的操作系统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，就像是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，像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Ox41FFFFFF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。

用户态的帧（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，每个地址空间是一个指向最上层页表的句柄（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，一个像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帧（device frame），这些设备帧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，D0-178B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](http://os.inf.tu-dresden.de/l4env/doc/html/l4sys-l4v2/index.html)。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**](http://www.inf.tu-dresden.de/index.php?node_id=1582&ln=en)**。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**](http://os.inf.tu-dresden.de/pubs/sosp97/),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](http://os.inf.tu-dresden.de/drops/overview.html)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](http://i30www.ira.uka.de/index.php?lid=en)和[新南威尔士州立大学操作系统研究小组](http://ertos.nicta.com.au/)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](http://www.l4ka.org/publications/2004/LeVasseur04UnmodifiedDriverReuse.pdf) ,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](http://www.ok-labs.com/)公司维护，但是研究工作基本上都是在[ERTOS](http://ertos.nicta.com.au/)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](http://www.cs.york.ac.uk/ftpdir/reports/YCST-2004-05.pdf)》 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，你不得不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；
* 通知（Notifications）提供一个像二进制信号量一样的非阻塞信号机制。
* 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一个端点，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()调用和seL4\_Recv（）的结合，发出请求的线程将被阻塞直到消息被接收并且收到回复信息。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\_Call首先执行seL4\_Send()调用，之后的操作和内核是否是MCS配置有关。在非MCS配置下，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。该应答句柄是一次性的，被调用后就立即被内核禁用。在MCS配置下，内核会更新由接收方提供的回复目标，回复目标是一个用于追踪回复信息的容器。在两种配置下，calling的线程都会回复目标被调用后才解除阻塞。

* **seL4\_Recv()/seL4\_Wait()** 阻塞式接收

用于线程阻塞式通过端点（endpoint）或通知（notification）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或通知，如果在其它类型的句柄上调用seL\_Recv()，将引起一个异常。

（在MCS配置下，Wait是Recv的变式，区别在于Wait不需要被提供回复的对象，在非MCS配置下，Wait和Recv相同）

* **seL4\_NBRecv()** 非阻塞式接收

用于线程查看已经等待在通知上的信号或者已经等待在端点上的信息。这个系统调用的对象只能是端点或通知，如果在其它类型的句柄上调用seL\_Recv()，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\_ReplyRecv()** 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。

* **seL4\_NBWait()** 非阻塞式接收(只在MCS配置下存在)

用于线程查看已经等待在通知上的信号或者已经等待在端点上的信息。没有请求中的信号或信息则直接返回。

* **seL4\_NBSendRecv()/NBSendWait()** 非阻塞式发送并转入接收(只在MCS配置下存在)

NBSend（）和Recv（）/Wait（）的结合。

## 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，像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（数量为2的整数幂），槽中可以有一个句柄，也可以为空。

* 线程控制块 Thread Control Blocks （TCBs）

代表seL4中执行的线程。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。

* 调度上下文 Scheduling contexts（只在MCS配置下）

代表seL4中的CPU时间。用户可以从原始对象中创建调度上下文，但调度上下文创建时为初始为空，不代表任何时间。最初每个node都有一个叫SchedControl的句柄，该句柄允许调度上下文（与优先级控制线程对CPU时间的访问相结合）随着参数增加信息。

* 端点 Endpoints

IPC是同步的：尝试向端点发送信息或从端点接收信息的线程会阻塞，直到信息可以被送达。这意味着仅当发送方和接收方在端点处会面，内核可以用一个复制来传递信息。

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）。另外，端点句柄具有授予的权力（允许将句柄作为传递信息的一部分）

* 回复对象 Reply Objects（只在MCS配置下）

追踪调度上下文，为一次性回复句柄提供容器。

* 通知对象 Noitication Objects

提供简单的信号机制。通知是一个字位宽的标志数组，每个标志都是一个二进制信号量。在单次操作中提供标志子集的信号机制，查看任何标志位，并在任何标志位收到信号前保持阻塞。 通知句柄可以是signal-only或是wait-only的。

* 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。这些对象大部分与硬件直接相关，也就是所谓的架构依赖。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 specific interrupt source associated with a specific 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。应用想要创建新的对象时，必须对内存有显式的权限（通过Untyped Memory句柄），所有对象在创建后都消耗一个固定大小的内存。这个机制可以精确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（除硬件规定的限制），所以可以避免耗尽资源所导致的终止服务

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。之后创建最初的用户线程（有合适的地址和句柄空间）。余下的内存以两种形式留给初始化线程，一种时指向原始内存（Untyped Memory）的句柄，还有就是额外的指向“引导初始线程所必需的内核对象”的句柄。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。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() receives 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.

原始内存对象代表着两种不同的内存：一般用途的内存以及设备内存。一般用途的内存可以被转换成任意目标类别，并可以被任何由内核提供的在原始内存上的操作所使用。设备内存包括由硬件平台所决定的为设备预留的区域。设备对象在以下方面被内核限制：

设备原始对象只能被转换为帧或者其他原始对象，例如开发者不能从设备内存中创造端点。

从设备原始对象转换而来的帧不能被设置为IPC buffer，或被在ASID池的创造中被使用。

### 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。这样，在这个调用之后，就没有指向原始区域的任何有效引用了。此后区域就可以安全的被转换或重用。

句柄只是一个象c++ io库打开文件函数返回的那个整数。

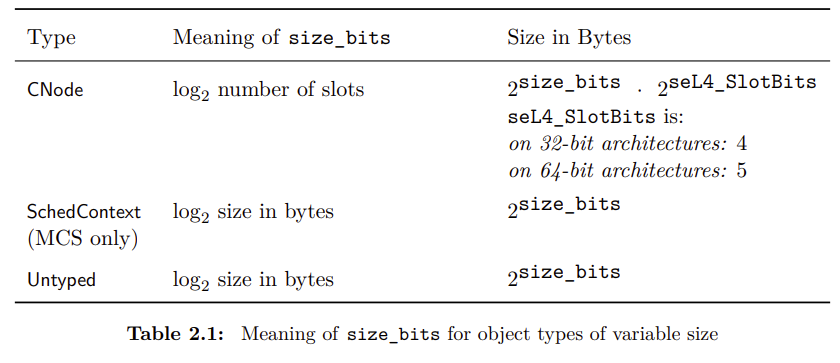
int open(const char \* pathname, int flags);

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。

### 2.4.2 对象大小的总结

当进行类型转换时，知道对象需要占据多少内存是很有用的。对象的大小在libsel4中定义。需指出，CNodes, SchedContexts (只在MCS配置下)以及Untyped Objects有着可变的大小。当把原始内存转换成CNodes或者SchedContexts，或是Untyped Object分解成更小的Untyped Objects时，seL4\_Untyped\_Retype（）的size\_bits参数用于指定目标对象的大小。对于所有其他类别，它们的大小是固定的，所以seL4\_Untyped\_Retype（）的size\_bits参数被忽略。



对seL4\_Untyped\_Retype（）的单次调用可以将Untyped Object转换成多种对象。创建的对象数量由参数num\_objects指定。所有创建的对象必须是同一类，由参数type指定。对于可变大小对象的创建，单次创建的每个对象要求大小相同。如果创建对象所需的内存大于原始对象剩余的未分配内存，将会报错。

# 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。线程使用装有句柄的槽的地址来在它们的Cspace中提及句柄。Cspace中的地址是指向目标槽的一系列CNode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，包括在用户空间的Cspace。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。

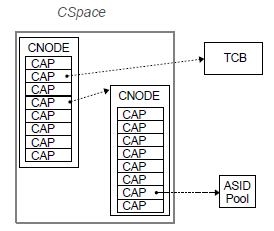


图3.1 CSpace与CNode关系示意图

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16/32（取决于是32还是64位架构）字节，每个槽刚好存储一个句柄。跟其它内核对象一样，CNode也必须通过调用[seL4\_Untyped\_Retype()](#page76)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，以及在存在的CNodes中是否留有空余的句柄槽以便[seL4\_Untyped\_Retype()](#page76)的成功调用。

### 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()**](#page60)

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()](#page60)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 specified 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 specified capability slots. It is essentially two  [seL4 CNode Move()](#page61) invocations: one from the second specified slot to the first, and one from the third to the second. The first and third specified 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 specified 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保存到指定槽中。（非MCS配置下）

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\_CancelBadgedSends()**

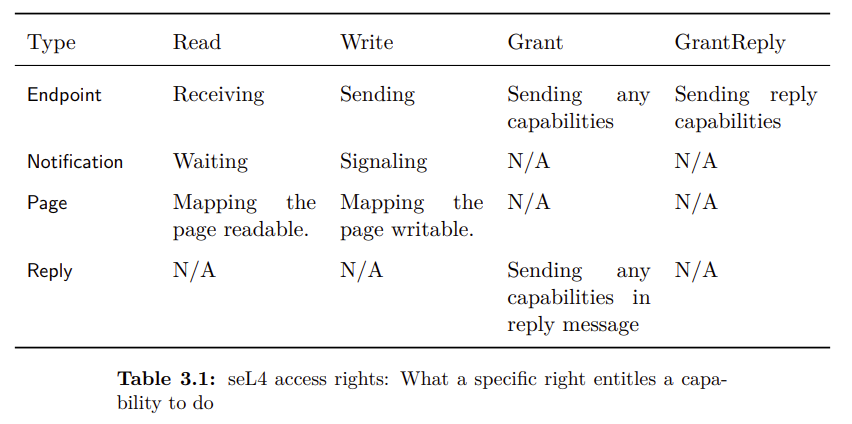
取消任何与特定句柄使用相同标记或相同对象的发送。

### 3.1.3刚完成类型转换对象的句柄（Capabilities to Newly-Retyped Objects）

当使用seL4\_untyped\_retype()将原始内存转换为对象时。新转换得到的对象的句柄被放置在由 root, node\_index和node\_depth等参数指定的一个CNode中的连续的槽中。node\_offset参数指定第一个句柄被放置在CNode中的下标。参数num\_object指定创建的句柄数。所有的槽必须都是空的，不然会报错。所有句柄都要放到同一个CNode中。

### 3.1.4句柄权限（Capability Rights）

就像之前提到的，一些句柄权限带有访问权。 目前，访问权与Endpoints ,Notifications,Pages和Replying的句柄有关。与句柄相关的访问权决定了可以调用的方法。seL4支持四种访问权，读Read、写Write、授予Grant，授予回复Grantreply。读Read、写Write、授予Grant两两正交。Grantreply是弱化版的Grant，例如，如果你已经由了Grant，那么有没有Grantreply就无所谓了。每种访问权限的意思根据不同的对象类别而有所不同，具体在Table3.1中描述。



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。

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。

### 3.1.5 句柄继承树（Capability Derivation Tree）

像在2.4.1中提到的那样，seL4把句柄间的继承关系记录在句柄继承树中。

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  [(Sec](#page23)tion 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，均是第一层的子句柄。第三层左侧是第二层untyped句柄的复制。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。

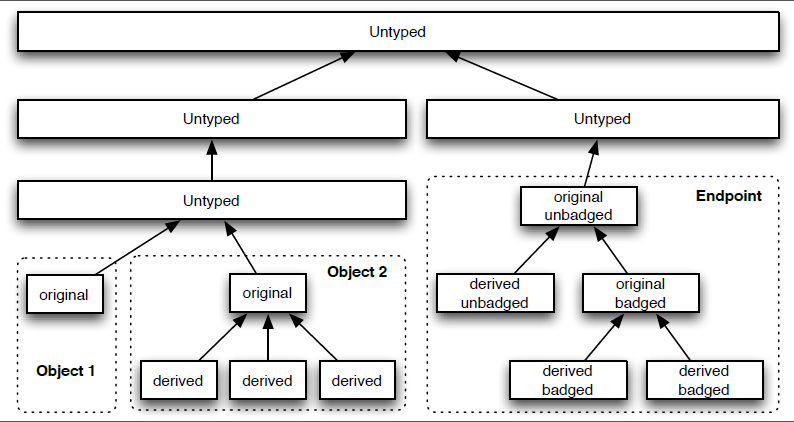


Figure 3.1: 句柄继承树示例.

普通的原初句柄可以有一层的继承句柄，对这些句柄的进一步复制将产生兄弟句柄，就像在第五层中展示的那样。端点和通知的句柄是这个体系的例外，它们通过标记机制能够支持额外的一层深度。原初的端点或通知句柄是未标记的。使用铸造方法可以创造有特定标记的句柄。这个新的，带有标记的句柄被当作原句柄来处理，并且和其他句柄一样支持一级的子句柄。

## 3.2 删除、撤销（Deletion, Revocation）

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 specified 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.

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对象的有向图结构，每个CNode是一个由槽slot组成的表，每个槽中为空或者存一个句柄或指向其它CNode的句柄。回忆2.3中提到过每个CNode中包含的槽的数目是2的整数幂，指数幂被称为基数radix。内核在每个线程的线程控制块TCB中保存了代表CSpace的根CNode。内核在每个线程的TCB中存储了一个句柄，该句柄指向各个线程的Cspace中的根CNode。从概念上来说，一个CNode句柄不只存了对CNode的引用，还携带一个守卫guard值。

**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地址的最高几位，看它与CNode的守卫（guard）值是否匹配。如果不匹配，那么查找失败。

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，像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，内核就从句柄里拿出接下来的radix比特的句柄地址，作为CNode句柄指向的CNode的索引。由这些接下来的radix比特指向的槽s可能装有另一个CNode或者装有其他东西（包括空）。如果s装有了一个CNode句柄c，并且在句柄地址还存在着剩余的比特（接在radix比特后面）未被翻译，那么就继续这个查找过程：从句柄c查起，使用剩余的句柄地址来查。否则，查询成功结束，被讨论的此句柄地址指向句柄槽s。（radix比特里的一些位就是偏移植，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 first consults the CNode capability in the thread's TCB that defines the root of the thread's CSpace. It then compares that CNode's guard value against the most significant bits of the capability address. If the two values are different, lookup fails. Otherwise, the kernel then uses the next most-significant radix bits of the capability address as an index into the CNode to which the CNode capability refers. The slot s identified 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 different 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.

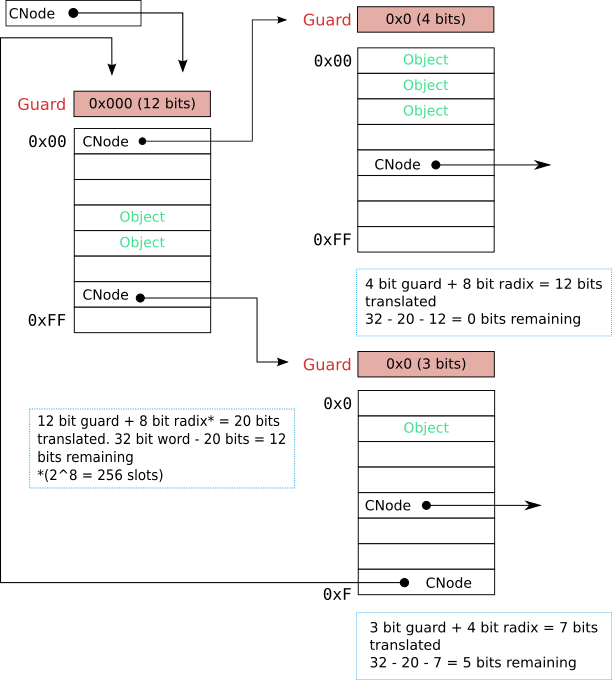


图3.2 CSpace示例图，各层次（level）的对象引用，各种守卫（guard）、索引尺寸（radix sizes）和CNode内部的相互引用

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。句柄的编址与上一节的地址转换算法一致。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）。当一个CPointer与深度限制depth共同使用时，只有最低的depth位会在地址转换中被使用。

某些方法，如[seL4\_Untyped\_Retype(](#page76))，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。

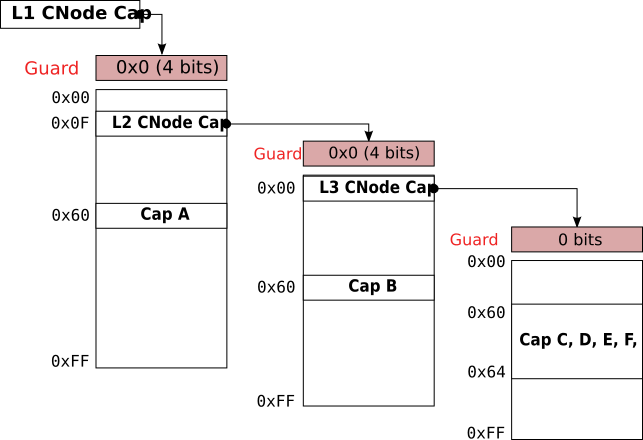


图3.3 任意CSpace拓扑图

**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IPCbuffer的异常处理函数中（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IPCbuffer中不同偏移的地方。

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 + seL4\_CapFault\_BitsLeft | *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 + seL4\_CapFault\_BitsLeft | Bits of CPTR remaining to decode |
| seL4\_CapFault\_DepthMismatch\_BitsFound | 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，radix比特中剩下的比特数小于当前访问的守卫（guard）尺寸，或者比特位数够，但两者不相同。

|  |  |
| --- | --- |
| Data | Meaning |
| Offset + 0 | seL4 GuardMismatch |
| Offset + seL4\_CapFault\_BitsLeft | 剩下还没解析的比特位  Bits of CPTR remaining to decode |
| seL4\_CapFault\_GuardMismatch\_GuardFound | The CNode's guard |
| Offset + seL4\_CapFault\_GuardMismatch\_BitsFound | The CNode's guard size |

# 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”。

像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。

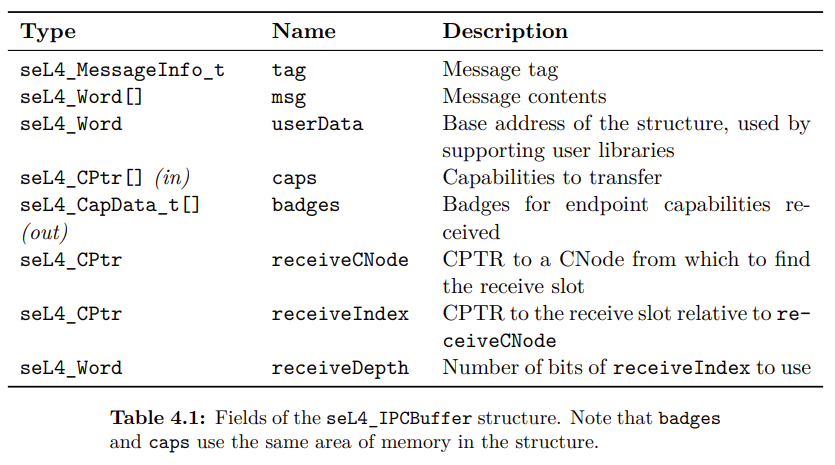
seL4微内核提供了线程间传递消息的机制，这个机制也用来传递用户程序与内核提供的服务之间的通信。信息通过调用句柄来发送到内核对象。发送给端点的信息最终会到其他线程，而发送给其他对象的信息会被内核处理。本章将讨论通用的消息格式和端点，及它们在不同应用间如何通信。

## 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，最初的几个消息寄存器由CPU寄存器实现，剩下的数量由固定的内存区域IPC buffer来支持。这样的设计是为了效率高：非常短的消息无需使用内存。IPC缓冲区归属于调用线程。

每个IPC信息都有一个标签（结构体seL4\_MessageInfo\_t）。标签由四部分组成：lable，消息长度，句柄数量（extraCaps域）以及未解封的句柄（capsUnwrapped）域。消息长度和句柄数量决定消息寄存器或者发送线程想要发送到句柄数量。或决定真正送达的消息寄存器和句柄数量。lable不被内核翻译，并作为消息的第一个负载数据来不被改动的进行传输。例如lable可能被用来确定需要的操作。capsUnwrapped域只在接收方被使用，来指定句柄被接收的方式。在4.2.2章节会解释这一点。

内核默认IPC缓冲区包含一个名为seL4\_IPCBuffer的结构类型，定义在表4.1中。内核使用尽可能多的物理信息寄存器来传递IPC信息。当需要传递的参数数量多余可以使用的物理寄存器数量时，内核会使用IPC缓冲区的msg域来传参。不过在这个数组里还是留有了物理信息寄存器的空间。例如，如果IPC传递或者内核对象调用需要4个信息寄存器，而在此架构中只有两个可以使用的物理信息寄存器，那么参数1和2会通过物理信息寄存器传递，参数3和4会通过msg[2]，msg[3]传递。这使得用户级的对象调用存根在需要的情况下可以把物理寄存器中传递的参数复制到msg数组的空余部分。Msg域存在于内核忽略的seL4\_IPCBuffer结构中。用户级的存根可能想将message tag从CPU寄存器复制到Msg域中，然而提供给内核的用户级存根不会这么做。



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4.2 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4.3 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4.4 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 re-  ceived |
|
| 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，像这样的函数：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 端点Endpoints

端点允许少量的数据及句柄（也就是IPC buffer）在两线程间传递。端点对象可通过使用章节2.2中的系统调用来直接被调用。

IPC端点使用相遇模型，也就是同步和阻塞模型。端点对象可能在线程中排队来发送或者接收。

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。

在没有写权限的情况下尝试去Send或者Call会返回错误。对于Send的情况，错误被直接忽略（内核不允许回复）。因此如果一个Send请求因为没有写权限而失败，我们也无从得知。同样，没有读权限的Recv请求也会失败，见6.2章。否则，通过端点接收到的来自其他线程的错误信息和正常信息将不可区分。

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）。端点标记是由调用者指定的附属在端点句柄上的一个32/64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。

在32位的平台下，标记只有后28位被使用，内核会忽略最高的4位。而在64位的平台下，全部64位被使用。

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

消息中可以有句柄，句柄会被抄送给接收者。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，IPC缓冲区里有三个域：receiveCNode，receiveIndex和receiveDepth。这三个域确定根CNode，句柄地址，以及用多少位来寻找放置句柄的槽。句柄地址在3.3.2章节中有描述。

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。

一个未被解封的句柄通过从发送CNode的槽复制到接收CNode的槽来被传输。发送方保持对发送句柄的访问。

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。

### 4.2.4 调用与回复 Calling and replying

如同在2.2章解释的那样，当使用者在一个端点句柄调用seL4\_Call(),将发生某些特定的行为。第一步，call将会与seL4\_Send()做出完全相同的行为。在相遇及所有的正常IPC流程结束之后，seL4\_Call()不会直接返回调用者，而是将查看Grant或者Grantreply是否在端点句柄被调用。

如果未被调用：那么调用者线程将如同调用了seL4\_TCB\_Suspend()一样暂停，发送部分照常执行。

如果被调用：接收方TCB的一个特定槽中会设置一个回复句柄。回复句柄的授予权将通过复制端点句柄的授予权来设置，该端点句柄由接收方通过seL4\_Recv()调用。之后，调用线程被阻塞，等待回复。

回复句柄直接指向调用者线程，一旦调用被执行，回复句柄就与原端点完全无关。即便远端点已经不存在，回复句柄也会持续存在并指向仍在等待回复的调用者。

在此之后，回复句柄可以在seL4\_Reply()相关的位置被调用，或通过seL4\_CNode\_- SaveCaller()来被保存在一个可寻址的槽中，之后通过seL4\_Send().来调用。特定的槽不能通过任何的CPtr来寻址，因为它不属于任何CSpace。

在端点上回复句柄与一个正常的send以相同的方式被调用。回复句柄隐含有写权限，所以信息总是可以通过。在回复中转发句柄只能在回复句柄有授予权的时候发生，转发的方式与在4.2.2一节中提到的在一个正常的IPC中转发相同。

与普通的端点转发相比主要的不同在于内核确保调用回复句柄的过程不阻塞：如果你有一个回复句柄，那么它指向的线程就一定在等待回复。然而回复句柄是一种非持有的引用，与其他所有句柄都相反。这意味着如果调用者线程不存在了或者以某种方式被修改导致不能完成回复（例如被seL4\_TCB\_Suspend()暂停），内核会立即摧毁回复句柄。

回复句柄一被调用，调用者就像完成了seL4\_Recv()操作一样只接收信息，并开始继续运行。

seL4\_Call()不只是因为效率的原因而存在。它与在send后面紧接着一个recv的区别在于它使得系统可在比传统设置更高效的设置下工作。特别地，它不查看标记就可以确保回复是从收到call的线程发来的。

# 5 通知

通知（Notifications）逻辑上表现为信号量的集合，提供一个象二进制信号量一样的非阻塞信号机制。

1965年，荷兰学者Dijkstra提出的信号量（Semaphores）机制是一种卓有成效的进程同步工具。在长期且广泛的应用中，信号量机制又得到了很大的发展，它从整型信号量经记录型信号量，进而发展为“信号量集”机制。现在，信号量机制已经被广泛地应用于单处理机和多处理机系统以及计算机网络中。1965年，荷兰学者Dijkstra提出的信号量（Semaphores）机制是一种卓有成效的进程同步工具。在长期且广泛的应用中，信号量机制又得到了很大的发展，它从整型信号量经记录型信号量，进而发展为“信号量集”机制。现在，信号量机制已经被广泛地应用于单处理机和多处理机系统以及计算机网络中。

## 5.1 通知对象Notiﬁcation Objects

A Notiﬁcation object contains a single data word, called the notiﬁcation word. Such an object supports two operations: seL4 Signal() and seL4 Wait().

Notiﬁcation capabilities can be badged, using seL4 CNode Mutate() or seL4 CNode Mint(), just like Endpoint capabilities (see Section 4.2.1). As with Endpoint capabilities, badged Notiﬁcation capabilities cannot be unbadged, rebadged or used to create child capabilities with diﬀerent badges.

## 5.2 Signalling, Polling and Waiting

The seL4 Signal() method updates the notiﬁcation word by bit-wise or-ing it with the badge of the invoked notiﬁcation capability. It also unblocks the ﬁrst thread waiting on the notiﬁcation (if any). As such, seL4 Signal() works like concurrently signalling multiple semaphores (those indicated by the bits set in the badge). If the signal sender capability was unbadged or 0-badged, the operation degrades to just waking up the ﬁrst thread waiting on the notiﬁcation (also see below).

The seL4 Wait() method works similarly to a select-style wait on the set of semaphores: If the notiﬁcation word is zero at the time seL4 Wait() is called, the invoker blocks. Else, the call returns immediately, setting the notiﬁcation word to zero and returning to the invoker the previous notiﬁcation-word value.

The seL4 Poll() is the same as seL4 Wait(), except if no signals are pending (the notiﬁcation word is 0) the call will return immediately without blocking.

If threads are waiting on the Notiﬁcation object at the time seL4 Signal() is invoked, the ﬁrst queued thread receives the notiﬁcation. All other threads keep waiting until the next time the notiﬁcation is signalled.

## 5.3 绑定通知Binding Notiﬁcations

Notiﬁcation objects and TCBs can be bound together in a 1-to-1 relationship through the seL4 TCB BindNotification() invocation. When a Notiﬁcation is bound to a TCB, signals to that notiﬁcation object will be delivered even if the thread is receiving from an IPC endpoint. To distinguish whether the received message was a notiﬁcation or an IPC, developers should check the badge value. By reserving a speciﬁc badge (or range of badges) for capabilities to the bound notiﬁcation — distinct from endpoint badges — the message source can be determined.

Once a notiﬁcation has been bound, the only thread that may perform seL4 Wait() on the notiﬁcation is the bound thread.

# 6 线程及其执行

线程与执行Threads and Execution

## 6.1 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）。在MCS配置下，调用上下文来负责管理执行时间。在非MCS配置下，执行时间由线程抽象来代表。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）。

在MCS配置下，调用上下文被抽象为调用上下文对象（SCO），且线程在非必须或者未收到调度上下文的情况下不会运行。

### 6.1.1 线程控制块 Thread Control Blocks

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。尽管线程并不一定要有一个IPC buffer，但没有IPC buffer的话，将无法完成绝大多数的内核调用，因为它们需要传递句柄。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;

### 6.1.2 线程创建Thread Creation

像其它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。这两种方法都把线程转入就绪状态。

在主内核中，这会导致线程被直接加入到调度器中。在MCS内核中，线程只在有调度上下文的情况下才运行。

在SMP的内核配置下，线程会在它附属的核中被继续。对于主核，这一机制通过seL4\_TCB\_SetAffinity()来设置，而在MCS核中，归属权通过调度上下文对象来获得。

### 6.1.3 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。

### 6.1.4 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，无节拍机制的调度器，具有256级优先级。所有的线程都有一个最大控制优先级（MCP）和一个普通的优先级，后者才是线程中有效的有效的优先级。当一个线程要去修改另一个（包括自己）线程的优先级时，它必须提供使用MCP的线程句柄。

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或MCP设得与自己的MCP一样高，就是说，线程不能造个比自己的MCP还高的子线程。最初的线程在初始时有着系统最高的优先级和MCP(seL4\_MaxPrio)。

设置线程优先级和MCP的方法为： seL4\_TCB\_SetPriority()和seL4\_TCB\_SetMCPPriority()。

在可被调度的线程中，每次会选出优先级最高的就绪线程。

线程优先级seL4\_PrioProps\_t由两部分组成：

Priority 线程被调度时使用的优先级

Maximum controlled priority (MCP) 线程设置自己或其他线程的优先级或MCP时可以设置的最大值。

### 6.1.5 MCS调度 MCS Scheduling

这一章节只适用于MCS配置使能的情况，此时线程必须有可用的线程调度对象，才可以被交予调度器。

### 6.1.6 线程调度上下文 Scheduling Context

对cpu执行时间的访问由调度上下文对象来控制，调度上下文由元组预算budget(b)和时长period (p)配置得到。它们的单位都是微秒，由seL4\_SchedControl\_Configure()设置。元组（b,p）构成线程执行的上限——内核在每p微秒中最多允许线程执行b微秒。然而b/p不代表运行时间占比的下限，一个线程在所有线程中有着最高或者与最高相同的优先级才可以到达b/p的时间占比，内核对此不会进行准入测试。结果就是，并不是所有参数的集合都是可以真正被调度的。如果多个线程都同时有着可用的时间预算，那么它们按先入先出来调度，时间预算用完了会应用时间片轮转算法。

可以被调度器选中的调度上下文（也就是时间预算还未用完的），被称为active激活的。预算的补充对于时间片轮转和偶发性的线程来说时不同的。对于进行时间片轮转的线程来说，每次当前点的调度上下文切换时，预算就会补充一次，直到它被耗光并被再次补充。

b == p的线程被称位时间片轮转线程，b作为时间片。否则内核使用偶发性服务器来进行暂时的强制隔离，使得对于任意可能的p值，b/p的特性都不会被超过。理论上，偶发性服务器使用的暂时隔离——防止线程使用超出它们分配的时间预算的时间，用以下算法来实现：

当线程在当前时间T开始执行，记录时刻Ts。

当线程停止执行（被抢占或阻塞）时，在Ts+p时刻补充T-Ts的消耗量，并把这部分时间从当前正被使用的补充时间中减去。

seL4通过在每个调度上下文中维护一个偶发性补充replenishment（简写位refills）的有向表来实现这个算法。每个replenishment由一个元组（rTime，rAmount）组成，rTime代表可以进行补充的时间点，rAmount代表补充的量。当线程执行时，它不断地从补充表的表头的rAmount汲取budget。如果rTime在未来的时间，属于当前调度上下文的线程会排在线程队列中等待更多budget。

时间片轮转线程和偶发性线程按相同方式被处理，除开补充预算的方式不同。时间片轮转线程只有两个refills，而且都总是准备好被使用。当时间片轮转线程停止执行时，预算从replenishment的头部移到尾部。一旦头部预算被消耗完，线程就按它的优先级从队列中被移除，并被加到队尾。

取决于replenishment的不同数量，偶发性的线程的行为不同，因此replenishment的数量必须是确定的。开发者通过两种方式来配置replenishment表的大小：

在使用seL4\_Untyped\_retype()创建调度上下文时，调度上下文的大小会确定，单个调度上下文中refills的最大数量也随之确定。

每个调度上下文都有一个名为extra\_refills的参数，限制调度上下文的的refills数量。这个值是加载一个基础值（2）的基础上，并且也受到调度上下文大小的限制。

执行时间非常短的线程（例如中断处理线程）被抢占的次数很少，因此应该有更少的refills，而b值大的运行时间长的线程应该有更高的refill值。附属于0个refill的调度上下文的线程将周期性运行，使用它们头部replenishment或者被yield的线程直到属于它们的下个周期前将不会被调度。

考虑到replenishment的数量有限，如果一个点的调度上下文改变，且即将离开的调度上下文没有足够的空间去存储新的replenishment，seL4将使用移除当前replenishment的方式来腾出空间。如果接下来的replenishment还没就绪，这可能导致抢占。拥有更多数量refills的线程消耗的时间将更接近于它们的budget，因为它们可以多次被抢占或切换线程时而不用补充他们的replenishment队列。然而调度的开支会增大，因为replenishment表会受到碎片化的影响。

线程执行时总是从当前的调度上下文中消耗预算。可以使用系统调用seL4\_Yield()来牺牲任何剩余的预算，并在下一个replenishment就绪前阻塞。

### 6.1.7 被动线程Passive Threads

线程可以通过seL4\_SchedContext\_UnbindObject()来解除和调度上下文的关联。这和暂停一个线程是不同的，区别在于阻塞等待于端点或通知队列的线程仍留在队列中并且可以接收信息或信号。然而解除调度上下文的线程在获得调度上下文前将不可被调度。没有调度上下文的线程被称为被动线程，因为它们没有其他线程的帮助就不能执行。

### 6.1.8 调度上下文创建Scheduling Context Creation

和其他的对象一样，调度上下文从seL4\_Untyped\_Retype创建而来。刚创建时，调度上下文是空的，代表0%的cpu执行时间。要给调度上下文增加参数，需要调用合适的SchedControl句柄，它在单一点上提供对cpu时间管理的访问。每个点的句柄调度控制句柄在运行时提供给初始任务。线程运行在调度上下文配置给它的点上。调度上下文的参数可以通过seL4\_SchedControl\_ConfigureFlags()设置和更新，它让预算和时长与一组标志进行按位或运算。

seL4\_SchedContext\_Sporadic：根据偶发性服务器算法而不是连续不断的带宽来限制执行时间。

内核不进行任何可调度性测试，因为任务准入留给用户级政策处理，可以在线或离线，可以静态或动态或者完全没有。

### 6.1.9 调度上下文给予Scheduling Context Donation

除了通过seL4\_SchedContext\_Bind()和seL4\_SchedContext\_UnbindObject()来显式地绑定或移除调度上下文，调度上下文还可以通过IPC在线程间移动。当使用系统调用seL4\_Call()和seL4\_NBSendRecv()来和一个被动线程通信时，调度上下文被隐式地给予。当一个活跃的线程用seL4\_Call()调用了一个端点并与一个被动线程相遇时，活跃线程的调度上下文会交给被动线程。生成的回复句柄可以保证被调用者仅仅是借用这个调度上下文：当回复句柄被发送的回复信息消耗后，调度上下文会返回给调用者。如果回复句柄被撤销，此时被调用者拥有调度上下文，那么调度上下文会归还给调用者。然而，如果是在一个深的调用链中，调用链的中间有一个回复句柄被撤销，此时被调用者还不持有调度上下文，线程将从调度上下文中被移除，调用上下文将维持在当前所在的地方。如果接收方不提供追踪给予的回复对象（例如使用seL4\_Wait()而不是seL4\_Recv()）那么调度上下文的给予将不会发生，但消息会照常送达。被动的接收方将被设置为不活跃，因为它没有调度上下文。

考虑一个例子，线程A调用线程B，线程B调用线程C。如果当C持有调度上下文，而B指向A的回复句柄被撤销，那么调度上下文将保留在C线程处。然而A和C之间仍会保留调用链，这样当C的回复句柄被撤销或是被调用时，调度上下文将会回到A。

seL4\_NBSendRecv()也可以导致调度上下文给予。如果操作的非阻塞发送阶段导致信息被发送至被动线程，调度上下文将被发送至这个被动线程，且进行系统调用的线程在接收阶段在接收端点上变成被动线程。不产生回复句柄，所以不能保证调度上下文会返回。这增加了记录管理的复杂度，但能使数据流像架构而不是像远程过程调用一样。特别指出seL4\_Call()不保证调度上下文的返回：这是一种本质上被信任的操作，因为服务器无法回复并返回调度上下文。

调度上下文也可以通过seL4\_SchedContext\_Bind()/seL4\_SchedContext\_UnbindObject()来与通知对象关联或解耦。如果一个信号发送到一个通知对象，该通知对象上有阻塞等待在上面的被动线程，该被动线程会获得与通知对象相关联的调度上下文。当该线程阻塞在该通知对象时，调度上下文会归还。这一特性可以使得被动服务器也能使用通知连结。

调度上下文可以用seL4\_SchedContext\_Unbind()从任意对象上解除关联（被关联或获得调度上下文给予的通知对象或者TCBs）

被动线程将会运行在调度上下文配置的CPU点上，并会被迁移至IPC。

### 6.1.10 调度算法 Scheduling Algorithm

只有拥有激活的调度上下文的线程才可以被调度。所有可以被调度的线程中，处于就绪状态的最高优先级线程会被选出。

线程如果拥有足够高的MCP且并且有合适的调度上下文句柄，它就可以操控调度器并使用IPC实现用户级的调度器。

调度上下文提供对CPU执行时间的访问并规定了CPU执行时间的上限，但当线程执行时受线程优先级决定。结果上，对CPU的访问是一个线程MCPs，调度上下文和调度控制句柄的函数。内核会确保在任何时间段内线程的运行时间都不会超过它们调度上下文中的预算，然而还是要靠系统设计者来保证整个系统是可调度的。

### 6.1.11 异常Exceptions

每个线程有两个与它关联的处理异常的端点（exception-handler endpoint），一个标准异常处理例程，一个暂停异常处理例程，后者只适用于MCS配置。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。这个线程接下来便可以采取合适的行动。错误IPC信息在章6.2节详细介绍。

标准异常处理端点通过seL4\_TCB\_SetSpace()或seL4\_TCB\_SetSchedParams()设置，而暂停异常处理例程通过seL4\_TCB\_SetTimeoutEndpoint()来设置。通过这些方法，可以将异常处理例程的句柄地址和线程相关联。这个地址之后被用来查找处理例程的端点，并且指向该端点的句柄会被安装在线程的内核CNode中。没有异常处理例程的线程，会使用一个空的句柄，但对于不同的异常处理例程类别，结果不一样。在报出一个异常前，处理例程的句柄会被有效化。内核不会再执行一次查找，但会查看句柄是一个有着正确权限的端点。

异常处理端点必须有Write权限，且必须有Grant和Grantreply权限二者之一。应答异常，将引起线程重新开始。对于特定的异常类别，回复信息的内容也许会被用来设置被重启的线程中寄存器的值。6.2章对此有详细解释。

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);

### 6.1.11.1 标准异常Standard Exceptions

当线程触发错误且该错误必须借助其它线程的的行为才能恢复时，使用标准异常处理例程。例如，如果一个线程因为未映射的虚页表而产生错误，那么在页表映射前，该线程不会有任何进展。如果线程在被设置为空句柄的时候触发了一个需要标准异常处理例程处理的错误，那么内核将暂停该线程使其不再运行。这是因为如果没有其他线程的帮助，标准异常无从恢复。所以没有设置标准异常处理例程的线程应被信任为不出错。

标准异常处理例程可以是被动的，在这种情况下它们运行在错误线程的调度上下文上。

### 6.1.11.2暂停异常（只在MCS配置下） Timeout Exceptions（MCS only）

对于拥有有效的暂停异常处理例程句柄的线程，如果当线程的时间预算用完但仍尝试运行，就产生暂停错误。暂停错误的处理是非强制的：如果线程没有设定暂停异常处理例程，那么就不会产生暂停异常，当时间预算补充后，线程会继续运行。这使得暂时敏感的线程可以在其它线程忽略它们的情况下处理超量预算。

暂停错误按线程登记，这意味着尽管用户也许没有暂停异常处理例程，服务器也可以允许单线程，时间敏感，被动的服务器去使用一个暂停异常处理例程，当服务器结束完成请求后，从恶意的或者不被信任的用完时间预算的用户中恢复系统。暂停错误处理例程可以访问服务器回复对象并向用户回复错误，之后重启服务器来来解决下一个用户请求。

如果回复信息被发送给网状服务器和调度上下文，但没有返回有效的预算，在网状服务器有暂停错误处理例程的情况下，会产生另一个暂停错误。

### 6.1.12寄存器读写方法中的消息结构

Message Layout of the Read-/Write-Registers Methods

通过seL4\_TCB\_ReadRegisters()和seL4\_TCB\_WriteRegisters()可以读写线程的寄存器。对于某些寄存器，内核会默认去除特定范围的位，并强迫它们去包含特定的值，避免它们被恶意设定为特定的值来危害系统，或者确保架构规格中规定的值被正确设定。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] |

## 6.2 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：

seL4\_CapFault,

seL4\_VMFault,

seL4\_UnknownSyscall,

seL4\_UserException,

seL4\_Fault\_DebugException,

seL4\_Fault\_TimeoutFault,

seL4\_Fault\_NullFault（表示没有错误，是正常的IPC信息）

错误以一种模仿来自错误线程调用的方式来发送。这意味着错误端点必须要有写权限，且至少有授予和授予回复权限二者之一，才可以发送错误信息。否则，就又出了一个错误（一般情况下线程仅暂停）。

### 6.2.1 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6.1。

表6.1 IPC消息内容

|  |  |
| --- | --- |
| 意思  Meaning | IPC缓存区位置  IPC buffer Location |
| Address at which to restart execution | seL4\_CapFault\_IP |
| Capability address | seL4\_CapFault\_Addr |
| In receive phase (1 if the fault happened | seL4\_CapFault\_InRecvPhase |
| during a wait system call, 0 otherwise) |  |
| Lookup failure description. As described | seL4\_CapFault\_LookupFailureType |
| in [Section 3.](#page28)4 |  |

### 6.2.2 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] |

### 6.2.3 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。

应答错误IPC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] |

### 6.2.4 Debug异常：断点和观察点 Debug Exception: Breakpoints and Watchpoints

Debug异常用向线程发送追踪或debug相关的事件。例如断点，观察点，追踪事件和指示表现观察事件。在内核配置包括它们的时候（当设置CONFIG\_HARDWARE\_DEBUG\_API时），用户空间线程支持它们。硬件debugging拓展API在内核被移植到一下平台时被支持：

• PC99: IA-32 and x86\_64

• Sabrelite (i.MX6)

• Jetson TegraK1

• HiSilicon Hikey

• Raspberry Pi 3

• Odroid-X (Exynos4)

• Xilinx zynq7000

可用的硬件debug资源的信息用以下的常量来呈现：

seL4\_NumHWBreakpoints：定义了硬件平台上所有可用类别的可用硬件断点寄存器的总数。以ARM Cortex A7为例，有6个专有的指令断点寄存器，4个专有的数据观察点寄存器，总共10个监察寄存器。因此在这个平台上seL4\_NumHWBreakpoints是10。指令断点寄存器总是会被分配较低的API-IDs，而数据观察点会分配在它们后面。

此外，seL4\_NumExclusiveBreakpoints，seL4\_NumExclusiveWatchpoints和 seL4\_NumDualFunctionMonitors在各目标平台上都用来定义某类别可用的硬件断点和观察点的数量。

seL4\_NumExclusiveBreakpoints：定义了能够只在指令执行出错时报错的硬件寄存器数量。目前只在ARM平台上设置。第一个专有断点的API-ID记录在seL4\_FirstBreakpoint。如果没有指令中断专有寄存器，seL4\_NumExclusiveBreakpoints将被设置为0，seL4\_FirstBreakpoint被设置为-1。

seL4\_NumExclusiveWatchpoints：定义了能够只在数据访问出错时报错的硬件寄存器数量。目前只在ARM平台上设置。第一个专有断点的API-ID记录在seL4\_FirstWatchpoint。如果没有指令中断专有寄存器，seL4\_NumExclusiveWatchpoints将被设置为0，seL4\_FirstWatchpoint被设置为-1。

seL4\_NumDualFunctionMonitors：定义了在上述两种错误产生时都能报错的硬件寄存器数量。目前只在x86平台上设置。第一个专有断点的API-ID记录在seL4\_FirstDualFunctionMonitor。如果没有指令中断专有寄存器，seL4\_NumDualFunctionMonitors将被设置为0，seL4\_FirstDualFunctionMonitor被设置为-1。

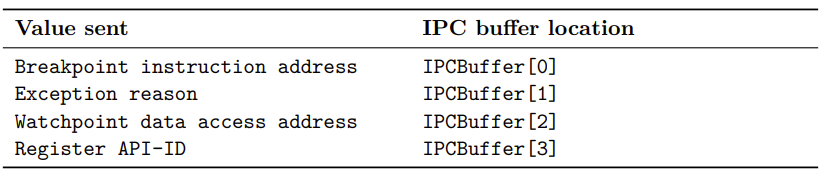


表6.2：Debug错误信息布局：单步错误内核不在错误信息中返回API-IDs。

### 6.2.5 Debug异常：Single-stepping

在内核设置CONFIG\_HARDWARE\_DEBUG\_API时，内核提供用户空间线程的硬件单步的支持。为达到这一目的，内核提供了seL4\_TCB\_ConfigureSingleStepping.这一调用。

调用者应该选择一个与指令断点相关的API-ID，当设定单步功能（也就是API-ID从0到seL4\_NumExclusiveBreakpoints - 1）的时候去使用。然而，并不是所有的硬件平台都需要一个真正的硬件断点寄存器来提供单步功能。如果调用者的硬件平台需要硬件断点寄存器的使用，它会使用在bp\_num中给它的断点寄存器，并在bp\_was\_consumed中返回true。如果硬件平台不需要断点来提供单步，seL4会在bp\_was\_consumed中返回false，并保持bp\_num不变。

如果bp\_was\_consumed为真，直到调用者通过随后对seL4\_TCB\_ConfigureSingleStepping的调用或是通过n\_instr为0的错误回复来禁用单步和释放该寄存器前，调用者都不应该尝试去为Breakpoint或Watchpoint使用来重新配置bp\_num。置num\_instructions为0来禁用单步。

在需要一个真正的配置了单步功能的硬件寄存器的架构上，seL4会限制可以被配置为单步的寄存器数量，任何时刻都只能有一个。任何被配置为单步的寄存器（如果有），会在单步debug错误回复中成为隐式的bp\_num参数。

内核的单步，还支持在发送单步错误信息前跳过特定数量的指令。单步时Num\_instructions应该被设置为1或者任何非0的整数，来在回复单步前跳过这个数目的指令。

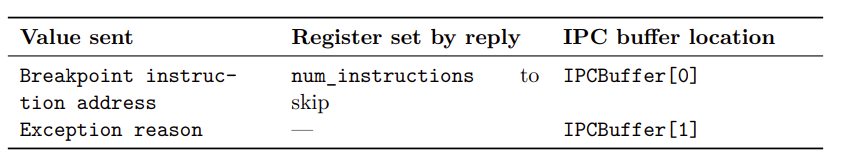


Table 6.3:单步错误信息布局

### 6.2.6 暂停错误（只在MCS配置下） Timeout Fault (MSC only)

### 6.2.7 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] |

## 6.3 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。

## 6.4 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。

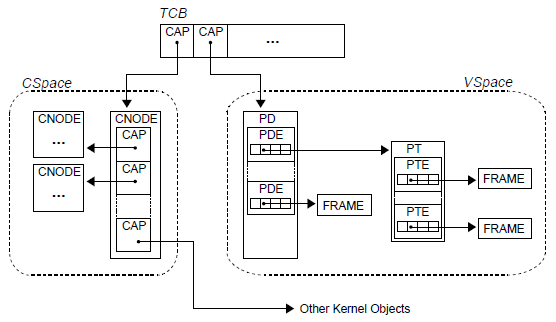


图5.1 与一个应用相关的内部数据结构图

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>

# 7 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。

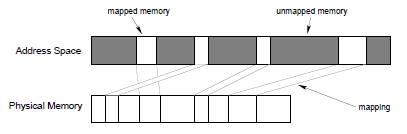


图6.1 虚拟内存与物理内存的映射关系

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。

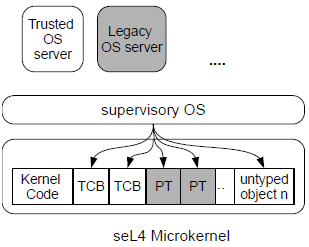


图6. 系统引导时的seL4内存布局

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。

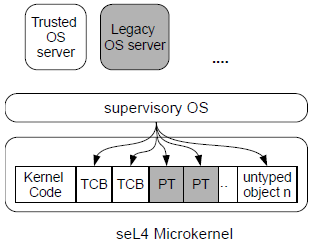


图6.内核服务被创建后的seL4内存布局

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。

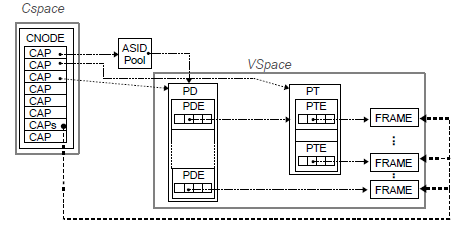


图6.1 seL4中的内存管理机制示意图

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。

## 7.1 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.

## 7.2 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()](#page87) or  [seL4 IA32 Page Map()](#page82) 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

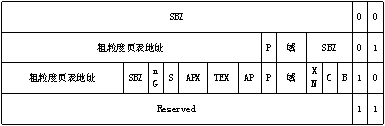


图6.2 ARMv6体系结构示意图

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，

## 7.3 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;

## 7.4 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

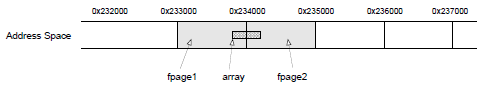


图6. 共享的页范围比较合适

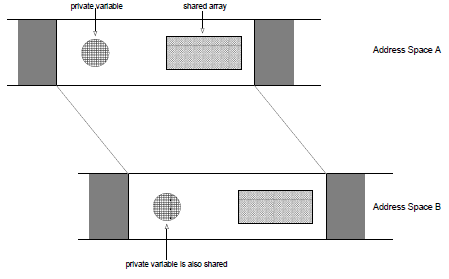


图6. 把不该共享数据结构也共享出去了，图中圆的数据结构是私有数据结构

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()。

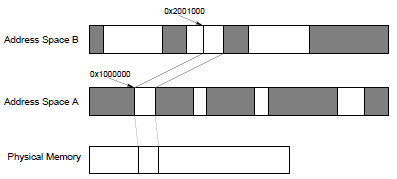


图6 两个虚拟地址空间共享一段物理内存

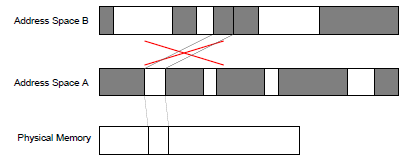


图6. 一段共享内存区间没有被映射（map）

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。

## 7.5 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。

# 8 硬件IO

硬件IO，Hardware I/O。

## 8.1 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。

## 8.2 IA-3特定I/O

### 8.2.1 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;

### 8.2.2 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]](#page91).

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),](#page37) 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(](#page60)) 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],](#page92) 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].](#page91) There is presently no way to query seL4 for how many Domain IDs are supported by the IOMMU and the  [seL4 CNode Mint()](#page60) 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](#page41) 6), namely seL4 IA32 Page Map().

Unmapping is accomplished with the usual unmap (see  [Chapter](#page41) 6) API call,  [seL4](#page83) - IA32 Page Unmap().

More information about seL4's IOMMU abstractions can be found in  [[Pal09]](#page92).

# 9 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

## 9.1 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](#page45)) |
| 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 \*/

};

## 9.2 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](#page51)) | | |
| seL4\_Word | | | numNodes | number of nodes (see  [Sec](#page51)- [tion 8.4](#page51)) | | |
| 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.](#page51)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 (2n bytes) | | |
| uint8 |  | t | initThreadCNodeSizeBits | CNode size (2n slots) | | |
| seL4\_Word | | | numDeviceRegions | number of device memory regions | | |
| seL4\_DeviceRegion[] | | | deviceRegions | device memory regions (see [Table 8.3](#page50)) | | |

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 (2n 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。

## 9.3 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](#page51)) | 1 |
| 最大共享帧数  num\_sh\_frames | Number of frames shared be- tween seL4 nodes (see  [Sec](#page51)- [tion 8.4](#page51)) | 0 |

## 9.4 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。

# 10 编程接口API

## 10.1 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.

### 10.1.1 Invalid Argument

A non-capability argument is invalid.

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Meaning | | |
|  |  |  |  |
| Label | seL4 |  | InvalidArgument |
| IPCBuffer[0] | Invalid argument number | | |
|  |  |  |  |

### 10.1.2 Invalid Capability

A capability argument is invalid.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field | Meaning | | |  |
|  |  |  |  |  |
| Label | seL4 |  | InvalidCapability |  |
|  |  |
| IPCBuffer[0] | Invalid capability argument number | | |  |
|  |  |  |  |  |

### 10.1.3 Illegal Operation

The requested operation is not permitted.

|  |  |
| --- | --- |
| Field | Meaning |
| Label | seL4 IllegalOperation |

### 10.1.4 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 | | |
|  |  |  |  |

### 10.1.5 Alignment Error

A supplied argument does not meet the alignment requirements.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field | Meaning | | |  |
|  |  |  |  |  |
| Label | seL4 |  | AlignmentError |  |
|  |  |
|  |  |  |  |  |

### 10.1.6 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.](#page28)4 | | |
|  |  |  |  |

### 10.1.7 Delete First

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

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Meaning | | |
|  |  |  |  |
| Label | seL4 |  | DeleteFirst |
|  |  |  |  |

### 10.1.8 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 |  |
|  |  |
|  |  |  |  |  |

### 10.1.9 Not Enough Memory

The Untyped Memory object does not have enough unallocated space to complete the  [seL4 Untyped Retype()](#page76) request.

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Meaning | | |
|  |  |  |  |
| Label | seL4 |  | NotEnoughMemory |
| IPCBuffer[0] | Amount of memory available in bytes | | |

## 10.2 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

### 10.2.1 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。

### 10.2.2 Recv

**原型：**

static inline seL4\_MessageInfo\_t seL4\_Recv

static inline seL4\_MessageInfo\_t

seL4\_Recv(seL4\_CPtr src, seL4 Word\* sender)

**功能：**

等待从一个句柄接收消息，如果没接到消息，则阻塞。

**参数说明：**

|  |  |  |
| --- | --- | --- |
| Type | Name | Description |
| seL4\_CPtr | src | 要调用的句柄 |
| seL4 Word\* | sender | The address to write sender information to. The sender information is the badge of the endpoint capability that was invoked by the sender, or the notiﬁcation word of the notiﬁcation object that was signalled. This parameter is ignored if NULL. |

**返回值：**

一个seL4\_MessageInfo\_t结构。

### 10.2.3 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);

}

### 10.2.4 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。

### 10.2.5 Polling 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。None-Blocking Send

**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。

### 10.2.6 Reply Recv

**原型：**

static inline seL4\_MessageInfo\_t seL4\_ReplyRecv

static inline seL4\_MessageInfo\_t

seL4\_ReplyRecv(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\_ReplyRecv(ep, tag, &sender\_badge);

}

return 0;

}

### 10.2.7 NBRecv

### 10.2.8 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

### 10.2.9 Signal

**原型：**

static inline void seL4\_Signal

static inline void

seL4\_Signal(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\_Signal()的源码能看出，seL4\_Signal()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\_Signal(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");

}

### 10.2.10 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。

### 10.2.11 Poll

## 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>

### 10.3.1 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。

### 10.3.2 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。

### 10.3.3 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。

### 10.3.4 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。

### 10.3.5 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。

### 10.3.6 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。

### 10.3.7 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。

### 10.3.8 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。

### 10.3.9 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。

### 10.3.10 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。

### 10.3.11 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 |

### 10.3.12 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。

### 10.3.13 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;

}

### 10.3.14 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"

);

}

### 10.3.15 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

### 10.3.16 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;

}

### 10.3.17 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;

}

### 10.3.18 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。

### 10.3.19 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);

}

### 10.3.20 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。

### 10.3.21 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。

### 10.3.22 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;

}

### 10.3.23 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。

### 10.3.24 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。

### 10.3.25 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;

### 10.3.26 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](#page53) for a description of the message register and tag contents upon error.

Description: See  [Section 5.](#page35)1

### 10.3.27 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。

### 10.3.28 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](#page53) for a description of the message register and tag contents upon error.

Description: See  [Section 5.1.](#page36)3

### 10.3.29 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 ca- |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | pability 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](#page53) for a description of the message register and tag contents upon error.

Description: See  [Section 5.](#page35)1

### 10.3.30 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](#page53) for a description of the message register and tag contents upon error.

Description: See  [Section 5.1.](#page35)2

### 10.3.31 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。

### 10.3.32 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 | 2n bytes (where n 4) |
| n-slot CNode | 16n bytes (where n 2) |
| Synchronous Endpoint | 16 bytes |
| Asynchronous Endpoint | 16 bytes |
| IRQ Control | -- |
| IRQ Handler | -- |

### 10.3.33 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 |

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

### 10.4.1 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](#page53) for a description of the message register and tag contents upon error.

Description: See  [Chapter](#page41) 6

### 10.4.2 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 as- |  |
|  |  |  |  |
|  |  |  |  |  |  |  |  | signed to. Must not be full. Each |  |
|  |  |  |  |  |  |  |  | ASID pool can contain 1024 entries. |  |
| seL4 |  | IA32 |  | PageDirectory | vroot | | | The page directory that is being as- |  |
|  |  |  |  |  |  |  |  | signed to an ASID pool. Must not al- |  |
|  |  |  |  |  |  |  |  | ready 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](#page53) for a description of the message register and tag contents upon error.

Description: See  [Chapter](#page41) 6

### 10.4.3 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.](#page46)1

Description: See  [Section 7.2.](#page46)1

### 10.4.4 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.](#page46)1

Description: See  [Section 7.2.](#page46)1

### 10.4.5 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.](#page46)1

Description: See  [Section 7.2.](#page46)1

### 10.4.6 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](#page53) for a description of the message register and tag contents upon error.

Description: See  [Section 7.2.](#page46)1

### 10.4.7 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](#page53) for a description of the message register and tag contents upon error.

Description: See  [Section 7.2.](#page46)1

### 10.4.8 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 |  |  | 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](#page53) for a description of the message register and tag contents upon error.

Description: See  [Section 7.2.](#page46)1

### 10.4.9 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 |  |  | 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](#page53) for a description of the message register and tag contents upon error.

Description: See  [Section 7.2.](#page46)2

### 10.4.10 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](#page22). |  |
| 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](#page53) for a description of the message register and tag contents upon error.

Description: See  [Section 7.2.](#page46)2

### 10.4.11 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 val- |  |
|  |  |  |  |  |  |  |  | ues for this type are given in  [Sec](#page22)- |  |
|  |  |  |  |  |  |  |  | [tion 3.1.3](#page22). |  |
| seL4 |  | IA32 |  | VMAttributes | attr | | | VM Attributes for the mapping. Pos- |  |
|  |  |  |  |  |  |  |  | sible values for this type are given in |  |
|  |  |  |  |  |  |  |  | [Chapter 6](#page41). |  |
|  |  |  |  |  |  |  |  |  |  |

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See  [Section 9.1](#page53) for a description of the message register and tag contents upon error.

Description: See  [Chapter](#page41) 6

### 10.4.12 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 val- |  |
|  |  |  |  |  |  |  |  | ues for this type are given in  [Sec](#page22)- |  |
|  |  |  |  |  |  |  |  | [tion 3.1.3](#page22). |  |
| seL4 |  | IA32 |  | VMAttributes | attr | | | VM Attributes for the mapping. Pos- |  |
|  |  |  |  |  |  |  |  | sible values for this type are given in |  |
|  |  |  |  |  |  |  |  | [Chapter 6](#page41). |  |
|  |  |  |  |  |  |  |  |  |  |

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See  [Section 9.1](#page53) for a description of the message register and tag contents upon error.

Description: See  [Chapter](#page41) 6

### 10.4.13 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](#page53) for a description of the message register and tag contents upon error.

Description: See  [Chapter](#page41) 6

### 10.4.14 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](#page41) 6

### 10.4.15 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. Pos- |  |
|  |  |  |  |  |  |  |  | sible values for this type are given in |  |
|  |  |  |  |  |  |  |  | [Chapter 6](#page41). |  |
|  |  |  |  |  |  |  |  |  |  |

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See  [Section 9.1](#page53) for a description of the message register and tag contents upon error.

Description: See  [Chapter](#page41) 6

### 10.4.16 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 |  |  | 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](#page53) for a description of the message register and tag contents upon error.

Description: See  [Chapter](#page41) 6

## 10.5 ARM-Speci c Object Methods

### 10.5.1 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 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](#page53) for a description of the message register and tag contents upon error.

Description: See Chapter 6

### 10.5.2 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 as- |  |
|  |  |  |
|  |  |  |  |  |  |  |  | signed to an ASID pool. Must not al- |  |
|  |  |  |  |  |  |  |  | ready 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](#page53) for a description of the message register and tag contents upon error.

Description: See Chapter 6

### 10.5.3 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 ushed. |  |
|  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See  [Section 9.1](#page53) for a description of the message register and tag contents upon error.

Description: See  [Chapter](#page41) 6

### 10.5.4 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](#page22). |  |
| seL4 |  | ARM |  | VMAttributes | attr | | | VM Attributes for the mapping. Pos- |  |
|  |  |  |  |  |  |  |  | sible values for this type are given in |  |
|  |  |  |  |  |  |  |  | [Chapter 6](#page41). |  |
|  |  |  |  |  |  |  |  |  |  |

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See  [Section 9.1](#page53) for a description of the message register and tag contents upon error.

Description: See  [Chapter](#page41) 6

### 10.5.5 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](#page22). |  |
| seL4 |  | ARM |  | VMAttributes | attr | | | VM Attributes for the mapping. Pos- |  |
|  |  |  |
|  |  |  |  |  |  |  |  | sible values for this type are given in |  |
|  |  |  |  |  |  |  |  | [Chapter 6](#page41). |  |
|  |  |  |  |  |  |  |  |  |  |

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See  [Section 9.1](#page53) for a description of the message register and tag contents upon error.

Description: See  [Chapter](#page41) 6

### 10.5.6 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](#page53) for a description of the message register and tag contents upon error.

Description: See  [Chapter](#page41) 6

### 10.5.7 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](#page41) 6

### 10.5.8 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. Pos- |  |
|  |  |  |  |  |  |  |  | sible values for this type are given in |  |
|  |  |  |  |  |  |  |  | [Chapter 6](#page41). |  |
|  |  |  |  |  |  |  |  |  |  |

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See  [Section 9.1](#page53) for a description of the message register and tag contents upon error.

Description: See  [Chapter](#page41) 6

### 10.5.9 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 un- |  |
|  |  |  |  |
|  |  |  |  |  |  |  |  | mapped. |  |
|  |  |  |  |  |  |  |  |  |  |

Return value: A return value of 0 indicates success. A non-zero value indicates that an error occurred. See  [Section 9.1](#page53) for a description of the message register and tag contents upon error.

Description: See  [Chapter](#page41) 6

# 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 ↓

\* 0xdfffffff ↑

\* | 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 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

# 15 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

## 15.1 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.

# 16 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

**ASID**, 56

capability, 6

PDE, 55

PTE, 55

seL4\_GetBadge, 35

seL4\_GetMR, 35

seL4\_GetTag, 35

seL4\_GetUserData, 35

seL4\_MessageInfo\_t, 34

seL4\_SetCap, 35

seL4\_SetCapReceivePath, 35

seL4\_SetMR, 35

seL4\_SetTag, 35

seL4\_SetUserData, 35

sel4utils\_bootstrap\_vspace\_with\_bootinfo, 132

vka, 140

**vka\_alloc\_object**, 141

**vka\_alloc\_object\_leaky**, 141

**vka\_free\_object**, 141

**vka\_object\_t**, 140

watermark, 16

# 参考文献

1. seL4 Reference Manual API version 1.3
2. seL4 Reference Manual Version 3.0.0
3. The Clustered Multikernel: An Approach to Formal Verification of Multiprocessor Operating-System Kernels
4. seL4主设计师的BLOG， microkerneldude Random rants and pontifications by Gernot Heiser：<https://microkerneldude.wordpress.com/>