RVM通用实时虚拟机监视器
技术参考手册

版本R2T1

突越·中生·菊石（七阶）

Mutatus·Mesozoa·Ammonite(R.VII)

M7M2(Ammonite) R2T1

通用实时虚拟机监视器（二版一型）

技术手册

系统特性

高性能实时响应

·在微控制器上提供优秀的实时虚拟化性能

·在微处理器上提供优秀的平均性能

全面的体系架构支持

·可简易地在多种架构之间移植

·支持准虚拟化和全虚拟化*

高安全性

·虚拟机之间硬件隔离

·支持故障重启和虚拟机看门狗功能

·支持 TrustZone、SGX 等多种处理器内建安全机制*

高灵活性

·可以运行时动态更新各个虚拟机

·可以进行虚拟机快照和恢复

全面的生态环境

·支持虚拟化绝大多数操作系统

·原系统应用程序无需修改即可使用

高易用性

·提供图形化虚拟机配置界面*

*正在开发中

目录

表目录

图目录

版本历史

版本	日期（年-月-日）	说明
R1T1	2018-03-09	初始发布
R2T1	2018-09-18	增加了移植说明

第 1 章 概述

1.1 简介

在现代微控制器（MCU）应用中，人们对灵活性、安全性和可靠性的要求日益提高。越来越多的微控制器应用要求多个互不信任的来源的应用程序一起运行，也可能同时要求实时可靠部分[1]、性能优先部分[2]和某些对信息安全有要求的应用[3]一起运行，而且互相之间不能干扰。在某些应用程序宕机时，要求它们能够单独重启，并且在此过程中还要求保证那些直接控制物理机电输出的应用程序不受干扰。

由于传统实时操作系统（Real-Time Operating System，RTOS）的内核和应用程序通常静态链接在一起，而且互相之间没有内存保护隔离，因此达不到应用间信息隔离的要求；一旦其中一个应用程序崩溃而破坏内核数据，其他所有应用程序必然同时崩溃。一部分 RTOS 号称提供了内存保护，但是其具体实现机理表明其内存保护仅仅能够提供一定程度的应用间可靠性而无法提供应用间安全性，不适合现代微控制器应用的信息安全要求。此外，传统 RTOS 的内核服务[4]是在高优先级应用和低优先级应用之间共用的，一旦低优先级应用调用大量的内核服务，势必影响高优先级应用内核服务的响应，从而造成时间干扰问题[1]。

RVM（Real-time Virtual machine Monitor）就是针对上述问题而提出的、面向高性能 MCU 的系统级虚拟化环境。它是运行在 RME 操作系统上的一个宿主型虚拟机监视器，使得在微控制器平台上运行多个虚拟机成为可能。它能够满足实时和非实时应用一起运行的需求，在实时应用和非实时应用之间不会产生时间干扰。它也能够使多来源应用程序在同一芯片上运行而不发生信息安全问题。

由于 RVM 强制隔离了系统中安全的部分和不安全的部分，这两部分就可以用不同的标准分开开发和认证。这方便了调试，也节省了开发和认证成本，尤其是当使用到那些较为复杂、认证程度较低和实时性较差的的软件包如用户图形界面（Graphical User Interface，GUI）和高级语言虚拟机（Python、Java 虚拟机）等的时候。此外，在老平台上已得到认证的微控制器应用程序可以作为一个虚拟机直接运行在 RVM 上，无需重新认证；这进一步节省了成本。

本手册从用户的角度提供了 RVM 的功能描述。关于各个架构的具体使用，请参看各个架构相应的手册。在本手册中，我们先简要回顾关于虚拟化的若干概念，然后分章节介绍 RVM 的特性和 API。

1.1.1 微控制器应用的特点

在实际应用中，MCU 系统占据了实时系统中的相当部分。它常常有如下几个特点：

[1] 如电机控制等

[2] 如网络协议栈、高级语言虚拟机等

[3] 如加解密算法等

[4] 如内核定时器等

1.1.1.1 深度嵌入

微控制器的应用场合往往和设备本身融为一体，用户通常无法直接感知到此类系统的存在。微控制器的应用程序[1]一旦写入，往往在整个生命周期中不再更改，或者很少更改。即便有升级的需求，往往也由厂家进行升级。

1.1.1.2 高度可靠

微控制器应用程序对于可靠性有近乎偏执的追求，通常而言任何稍大的系统故障都会引起相对严重的后果。从小型电压力锅到大型起重机，这些系统都不允许发生严重错误，否则会造成重大财产损失或人身伤亡。此外，对于应用的实时性也有非常高的要求，它们通常要么要求整个系统都是硬实时系统，或者系统中直接控制物理系统的那部分为硬实时系统。

1.1.1.3 高效的资源利用

由于微控制器本身资源不多，因此高效地利用它们就非常重要了。微控制器所使用的软件无论是在编写上和编译上总体都为空间复杂度优化，只有小部分对性能和功能至关重要的程序使用时间复杂度优化。

1.1.1.4 资源静态分配，代码静态链接

与微处理器不同，由于深度嵌入的原因，微控制器应用中的绝大部分资源，包括内存和设备在内，都是在系统上电时创建并分配的。通常而言，在微控制器内使用过多的动态特性是不明智的，因为动态特性不仅会对系统的实时响应造成压力，另一方面也会增加功能成功执行的不确定性。如果某些至关重要的功能和某个普通功能都依赖于动态内存分配，那么当普通功能耗尽内存时，可能导致重要功能的内存分配失败。基于同样的或类似的理由，在微控制器中绝大多数代码都是静态链接的，一般不使用动态链接。

1.1.1.5 不具备内存管理单元

与微处理器不同，微控制器通常都不具备内存管理单元（Memory Management Unit，MMU），因此无法实现物理地址到虚拟地址的转换。但是，RVM 也支持使用内存保护单元（Memory Protection Unit，MPU），并且大量的中高端微控制器都具备 MPU。与 MMU 不同，MPU 往往仅支持保护一段或几段内存范围，有些还有很复杂的地址对齐限制，因此内存管理功能在微控制器上往往是有很大局限性的。

1.1.1.6 处理器单核居多

如今，绝大多数的微处理器都是多核设计。然而，与微处理器不同，微控制器通常均为单核设计，因此无需考虑很多竞争冒险问题。这可以进一步简化微控制器用户态库的设计。考虑到的确有少部分微

[1] 通常称为“固件”

控制器采取多核设计或者甚至不对称多核设计，在这些处理器上使用微控制器用户态库时，可以考虑将这两个核分开来配置，运行两套独立的操作系统，并且将处理器之间的中断作为普通的设备中断处理，也即两个处理器互相把对方看做自己的外设。

考虑到以上几点，在设计 RVM 时，所有的内核对象都会在系统启动时创建完全，并且在此时，系统中可用内核对象的上限也就决定了。各个虚拟机则被固化于微控制器的片上非易失性存储器中。

1.1.2 软件版权与许可证

综合考虑到微控制器应用、深度嵌入式应用和传统应用对开源系统的不同要求，RVM 微控制器库所采用的许可证为 LGPL v3，但是对一些特殊情况[1]使用特殊的规定。这些特殊规定是就事论事的，对于每一种可能情况的具体条款都会有不同。

1.1.3 主要参考系统

RTOS 通用服务主要参考了如下系统的实现：

RMProkaron（@EDI）

RT-Thread（@睿赛德）

FreeRTOS（@Real-Time Engineering LTD）

Contiki（@The Contiki Community）

虚拟机监视器的设计参考了如下系统的实现：

Nova（@TU Dresden）

XEN（@Cambridge University）

KVM（@The Linux Foundation）

其他各章的参考文献和参考资料在该章列出。

1.2 前言

作为一个虚拟机监视器，RVM 依赖于底层 RME 操作系统提供的内核服务来维持运行。RVM 采用准虚拟化技术，最大限度节省虚拟化开销，典型情况下仅增加约 1%的 CPU 开销。

RVM 完全支持低功耗技术和 Tick-Less 技术，可以实现全系统无节拍工作，最大限度减少电力需求。对于那些对确定性有很强要求的应用，该项功能也可以关闭而使用传统的嘀嗒模式。RVM 也支持可信计算基（Trusted Computing Base，TCB）技术，将系统在安全态和非安全态区分开，允许一些虚拟机运行在普通状态，另一些虚拟机运行在可信计算基状态。这样可以允许轻量级区块链[2]等高敏感操作系统和应用运行在安全环境。

对于整个 RTOS 的虚拟化，可以使用 RVM 的宿主型虚拟机[3]技术，中断响应时间和线程切换时间将会增加到原小型 RTOS 的约 4 倍。对于那些对实时性有极高要求的应用，可以使用 RME 本身支持的容

[1] 比如安防器材、军工系统、航空航天装备、电力系统和医疗器材等

[2] 如 IOTA 等技术[3]

[3] 虚拟机分类见下文所述

器型虚拟机，更可以将处理代码写入中断向量，获取甚至快于小型 RTOS 的响应时间。如有多核处理器还可支持网络功能虚拟化（Network Function Virtualization，NFV）和定制硬件虚拟化，将网络功能或者定制 I/O 功能交给单独的一颗处理器来运行定制的虚拟机，从而大幅度增加网络和设备吞吐。

RME 和 RVM 在全功能配置下共占用最小 64kB 内存[1]，外加固定占用 128kB 程序存储器，适合高性能微控制器使用。此外，如果多个虚拟机同时使用网络协议栈等组件，使用本技术甚至可节省内存，因为网络协议栈等共享组件可以只在系统中创建一个副本，而且多个操作系统的底层切换代码和管理逻辑由于完全被抽象出去，因此在系统中只有一个副本。

在此我们回顾一下虚拟化的种类。在本手册中，我们把虚拟机按照其运行平台分成三类，分别称为原生型（I 型）、宿主型（II 型）和容器型（III 型）。同时，依照虚拟机支持虚拟化方式的不同，又可以分为全虚拟化型（Full Virtualization，FV）和准虚拟化型（Para-Virtualization，PV）。这两种分类方法是互相独立的。

1.2.1 原生型虚拟机监视器

原生型虚拟机监视器作为底层软件直接运行在硬件上，创造出多个物理机的实例。多个虚拟机在这些物理机的实例上运行。通常而言，这类虚拟机监视器的性能较高，但是自身不具备虚拟机管理功能，需要 0 域[2]对其他虚拟机进行管理。此类虚拟机监视器有 KVM、Hyper-V、ESX Server、Xen、XtratuM 等。

1.2.2 宿主型虚拟机监视器

原生型虚拟机监视器作为应用程序运行在操作系统上，创造出多个物理机的实例。多个虚拟机在这些物理机的实例上运行。通常而言，这类虚拟机监视器的性能较低，但是具备较好的虚拟机管理功能和灵活性。虚拟机均无特权，互相之间无法进行管理。此类虚拟机监视器有 VMware Workstation、Virtual Box、QEMU[3]等。RVM 是在 RME 上开发的一个应用程序，因此也属于宿主型虚拟机监视器。

1.2.3 容器型虚拟机监视器

容器型虚拟机监视器是由操作系统本身提供的功能，创造出多个相互隔离的操作系统资源分配空间的独立实例。多个应用程序直接在这些资源分配空间实例上运行。通常而言，这类虚拟机监视器的性能最高，虚拟化的内存和时间开销最低，但是它要求应用程序必须是针对它提供的接口编写的。通常而言，这接口与提供了容器型虚拟机监视器功能的操作系统自身的系统调用一致。它具备最低的管理和运行成本，但灵活性最差。此类虚拟机监视器有 Docker、Pouch、RKT、LXC 等。当 RVM 导致的性能开销被认为不合适时，由于 RME 操作系统自身也原生具备容器型虚拟机监视器的能力，也可以把 RVM 提供的用户态库用来开发原生的容器型虚拟机应用。

---

[1] 此时可支持 4 个虚拟机，若分配 128kB 内存则可支持 32 个虚拟机

[2] 也即第一个启动的、具有管理特权的虚拟机

[3] 不包括其 KVM 版本

1.2.4 全虚拟化型虚拟机监视器

全虚拟化型虚拟机监视器是有能力创建和原裸机完全一致的硬件实例的虚拟机监视器。通常而言，这需要 MMU 的介入来进行地址转换，而且对特权指令要进行二进制翻译、陷阱重定向或专用硬件[1]处理，对于 I/O 也需要特殊硬件功能[2]来重定向。此类虚拟机在 MCU 环境上难于实现，因为微控制器通常都不提供 MMU，也不提供相应的虚拟化硬件进行重定向功能。此类虚拟机监视器有 VMware Workstation、Virtual Box、QEMU、KVM、Hyper-V、ESX Server 等。

1.2.5 准虚拟化型虚拟机监视器

全虚拟化型虚拟机监视器是创建了和原裸机有些许差别的的硬件实例的虚拟机监视器。它使用超调用（Hypercall）处理特权指令和特殊 I/O，并借此避开那些复杂、不必要的全虚拟化必须提供的细节。此类虚拟机在 MCU 环境上能够以较高的性能实现。此类虚拟机监视器有 Xen 等。RVM 也是一个准虚拟化型虚拟机监视器，通过超调用处理系统功能。

1.3 RVM 的组成部分

RVM 由用户态库、中断服务线程和虚拟机监视器三个部分组成。关于这三个部分的介绍如下：

1.3.1 用户态库部分

这一部分是 RME 提供的系统调用的简单封装，它将原来的汇编格式 RME 系统调用接口封装成了 C 语言可以直接调用的形式。这一层被中断服务线程和虚拟机监视器同时依赖，如果需要编写 RME 的原生应用，那么也需要依赖于它。关于该部分的具体介绍以及各个参数的描述请参看 RME 的手册，在这里不加以额外介绍。

1.3.2 用户定义中断服务线程部分

这一部分是 RME 的中断服务线程。它们主要用来接受从中断向量来的信号，并对它们加以初步处理后转送信号出去，主要用于 RME 的用户态驱动程序。这些线程运行在用户态的多个进程之中，并且其时间片都是无限的。这一部分高度安全的线程在 RME 中拥有最高的优先级，并且和虚拟机完全隔离开来。即使虚拟机监视器和所有的虚拟机都已崩溃，这些线程仍能保证正常运行，从而维护被控设备的安全。

1.3.3 虚拟机监视器部分

这一部分是 RME 系统中优先级最低的部分。这一部分管理各个虚拟机，包括了错误处理线程、时间处理线程、中断处理线程和超调用处理线程、虚拟机中断向量线程和虚拟机用户线程六个线程。该部分是 RVM 的核心，实现了 RVM 的绝大部分功能。

---

[1] 如 Intel VT-x、AMD AMD-V 等

[2] 如 Intel VT-d 等

1.4 手册总览

本手册从用户的角度解释了 RVM 的基本组成与运行原理。在本手册中，所有的 typedef 类型之前的“rvm_”前缀均被省去。

1.5 本章参考文献

[1] P. Patel, M. Vanga, and B. B. Brandenburg, "TimerShield: Protecting High-Priority Tasks from Low-Priority Timer Interference," in Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE, 2017, pp. 3-12.

[2] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in Local Computer Networks, 2004. 29th Annual IEEE International Conference on, 2004, pp. 455-462.

[3] S. Popov, "The Tangle. url: https://iota. org," IOTA _ Whitepaper. pdf, 2017.

第 2 章 用户态库部分

2.1 简介

由于 RME 的系统调用是通过软件中断指令执行的，因此需要一些汇编代码才能访问 RME 的系统调用。由于在实际应用程序中使用汇编代码过于麻烦，因此 RVM 提供了用户态库对于这一部分做了封装，向外提供了这些系统调用的 C 语言接口。这些接口对用户态都是无副作用的，并且是线程安全的。通常而言，RVM 的一般用户不需要直接调用如下函数中的任何部分，而仅需要关注第三章所述的虚拟化接口部分即可。如果要进行 RVM 的定制，使用这些函数才是必须的。关于 2.2-2.8 所提到的函数的返回值和参数的具体意义请参见 RME 本身的手册。具体的函数列表如下所述：

2.2 权能表相关系统调用

这些接口向外提供了这些权能表相关调用的 C 语言接口。具体的函数列表如下所述：

表 2-1 权能表相关系统调用

名称	接口函数原型
创建权能表	ret_t RVM_Captbl_Crt(cid_t Cap_Captbl_Crt, cid_t Cap_Kmem, cid_t Cap_Captbl, ptr_t Raddr, ptr_t Entry_Num)
删除权能表	ret_t RVM_Captbl_Del(cid_t Cap_Captbl_Del, cid_t Cap_Del)
传递普通权能	ret_t RVM_Captbl_Add(cid_t Cap_Captbl_Dst, cid_t Cap_Dst, cid_t Cap_Captbl_Src, cid_t Cap_Src, ptr_t Flags) 备注：该函数只能用来传递页目录权能、内核功能调用权能、内核内存权能之外的权能。
传递页表权能	ret_t RVM_Captbl_Pgtbl(cid_t Cap_Captbl_Dst, cid_t Cap_Dst, cid_t Cap_Captbl_Src, cid_t Cap_Src, ptr_t Start, ptr_t End, ptr_t Flags) 备注：该函数不能被用来传递其他类型的权能。Start 和 End 两个参数标志了新的页表权能的起始槽位和终止槽位，也即其操作范围。
传递内核功能调用权能	ret_t RVM_Captbl_Kern(cid_t Cap_Captbl_Dst, cid_t Cap_Dst, cid_t Cap_Captbl_Src, cid_t Cap_Src, ptr_t Start, ptr_t End) 备注：该函数不能被用来传递其他类型的权能。Start 和 End 两个参数标志了新的内核功能调用权能的起始槽位和终止槽位，也即其操作范围。
传递内核内存权能	ret_t RVM_Captbl_Kmem(cid_t Cap_Captbl_Dst, cid_t Cap_Dst, cid_t Cap_Captbl_Src, cid_t Cap_Src, ptr_t Start, ptr_t End, ptr_t Flags)

名称	接口函数原型
	备注：该函数不能被用来传递其他类型的权能。Start 和 End 两个参数标志了新的内核内存权能的起始地址和终止地址，它们被强制对齐到 64 Byte。
权能冻结	ret_t RVM_Captbl_Frz(cid_t Cap_Captbl_Frz, cid_t Cap_Frz)
权能移除	ret_t RVM_Captbl_Rem(cid_t Cap_Captbl_Rem, cid_t Cap_Rem)

2.3 内核功能相关系统调用

这些接口向外提供了这些内核功能相关调用的 C 语言接口。具体的函数列表如下所述：

表 2-2 内核功能相关系统调用

名称	接口函数原型
内核调用激活	ret_t RVM_Kern_Act(cid_t Cap_Kern, ptr_t Func_ID, ptr_t Sub_ID, ptr_t Param1, ptr_t Param2)

2.4 页表相关系统调用

这些接口向外提供了这些页表相关调用的 C 语言接口。具体的函数列表如下所述：

表 2-3 页表相关系统调用

名称	接口函数原型
页目录创建	ret_t RVM_Pgtbl_Crt(cid_t Cap_Captbl, cid_t Cap_Kmem, cid_t Cap_Pgtbl, ptr_t Raddr, ptr_t Start_Addr, ptr_t Top_Flag, ptr_t Size_Order, ptr_t Num_Order)
删除页目录	ret_t RVM_Pgtbl_Del(cid_t Cap_Captbl, cid_t Cap_Pgtbl)
映射内存页	ret_t RVM_Pgtbl_Add(cid_t Cap_Pgtbl_Dst, ptr_t Pos_Dst, ptr_t Flags_Dst, cid_t Cap_Pgtbl_Src, ptr_t Pos_Src, ptr_t Index)
移除内存页	ret_t RVM_Pgtbl_Rem(cid_t Cap_Pgtbl, ptr_t Pos)
构造页目录	ret_t RVM_Pgtbl_Con(cid_t Cap_Pgtbl_Parent, ptr_t Pos, cid_t Cap_Pgtbl_Child, ptr_t Flags_Childs)
析构页目录	ret_t RVM_Pgtbl_Des(cid_t Cap_Pgtbl, ptr_t Pos)

2.5 进程相关系统调用

这些接口向外提供了这些进程相关调用的 C 语言接口。具体的函数列表如下所述：

表 2-4 进程相关系统调用

名称	接口函数原型
创建进程	ret_t RVM_Proc_Crt(cid_t Cap_Captbl_Crt, cid_t Cap_Kmem, cid_t Cap_Proc, cid_t Cap_Captbl, cid_t Cap_Pgtbl, ptr_t Raddr)
删除进程	ret_t RVM_Proc_Del(cid_t Cap_Captbl, cid_t Cap_Proc)
更改权能表	ret_t RVM_Proc_Cpt(cid_t Cap_Proc, cid_t Cap_Captbl)
更改页表	ret_t RVM_Proc_Pgt(cid_t Cap_Proc, cid_t Cap_Pgtbl)

2.6 线程相关系统调用

这些接口向外提供了这些线程相关调用的 C 语言接口。具体的函数列表如下所述：

表 2-5 线程相关系统调用

名称	接口函数原型
创建线程	ret_t RVM_Thd_Crt(cid_t Cap_Captbl, cid_t Cap_Kmem, cid_t Cap_Thd, cid_t Cap_Proc, ptr_t Max_Prio, ptr_t Raddr)
删除线程	ret_t RVM_Thd_Del(cid_t Cap_Captbl, cid_t Cap_Thd)
设置执行属性	ret_t RVM_Thd_Exec_Set(cid_t Cap_Thd, ptr_t Entry, ptr_t Param, ptr_t Stack)
设置虚拟属性	ret_t RVM_Thd_Hyp_Set(cid_t Cap_Thd, ptr_t Kaddr)
线程绑定	ret_t RVM_Thd_Sched_Bind(cid_t Cap_Thd, cid_t Cap_Thd_Sched, cid_t Cap_Sig, tid_t TID, ptr_t Prio)
更改优先级	ret_t RVM_Thd_Sched_Prio(cid_t Cap_Thd, ptr_t Prio)
接收调度事件	ret_t RVM_Thd_Sched_Rcv(cid_t Cap_Thd)
解除绑定	ret_t RVM_Thd_Sched_Free(cid_t Cap_Thd)
传递时间片	ret_t RVM_Thd_Time_Xfer(cid_t Cap_Thd_Dst, cid_t Cap_Thd_Src, ptr_t Time)
切换到某线程	ret_t RVM_Thd_Swt(cid_t Cap_Thd, ptr_t Full_Yield)

2.7 信号端点相关系统调用

这些接口向外提供了这些信号端点相关调用的 C 语言接口。具体的函数列表如下所述：

表 2-6 信号端点相关系统调用

名称	接口函数原型
创建信号端点	ret_t RVM_Sig_Crt(cid_t Cap_Captbl, cid_t Cap_Kmem,

名称	接口函数原型
	cid_t Cap_Sig, ptr_t Raddr)
删除信号端点	ret_t RVM_Sig_Del(cid_t Cap_Captbl, cid_t Cap_Sig)
向端点发送	ret_t RVM_Sig_Snd(cid_t Cap_Sig)
从端点接收	ret_t RVM_Sig_Rcv(cid_t Cap_Sig, ptr_t Option)

2.8 线程迁移相关系统调用

这些接口向外提供了这些线程迁移相关调用的 C 语言接口。具体的函数列表如下所述：

表 2-7 线程迁移相关系统调用

名称	接口函数原型
创建线程迁移	ret_t RVM_Inv_Crt(cid_t Cap_Captbl, cid_t Cap_Kmem, cid_t Cap_Inv, cid_t Cap_Proc, ptr_t Raddr)
删除线程迁移	ret_t RVM_Inv_Del(cid_t Cap_Captbl, cid_t Cap_Inv)
设置执行属性	ret_t RVM_Inv_Set(cid_t Cap_Inv, ptr_t Entry, ptr_t Stack, ptr_t Fault_Ret_Flag)
激活线程迁移	ret_t RVM_Inv_Act(cid_t Cap_Inv, ptr_t Param, ptr_t* Retval)
迁移调用返回	ret_t RVM_Inv_Ret(ptr_t Retval)

2.9 其他助手函数

为了方便常用数据结构编写、调试打印、线程栈初始化，RVM 还提供了一些助手函数。这些助手函数的详细说明如下：

2.9.1 创建双向循环链表

本操作初始化双向循环链表的链表头。

表 2-8 创建双向循环链表

函数原型	void RVM_List_Crt(volatile struct RVM_List* Head)
返回值	无。
参数	volatile struct RVM_List* Head 指向要初始化的链表头结构体的指针。

2.9.2 在双向循环链表中删除节点

本操作从双向链表中删除一个或一系列节点。

表 2-9 在双向循环链表中删除节点

函数原型	void RVM_List_Del(volatile struct RVM_List* Prev, volatile struct RVM_List* Next)
返回值	无。
参数	volatile struct RVM_List* Prev 指向要删除的节点（组）的前继节点的指针。
	volatile struct RVM_List* Next 指向要删除的节点（组）的后继节点的指针。

2.9.3 在双向循环链表中插入节点

本操作从双向链表中插入一个节点。

表 2-10 在双向循环链表中插入节点

函数原型	void RVM_List_Ins(volatile struct RVM_List* New, volatile struct RVM_List* Prev, volatile struct RVM_List* Next)
返回值	无。
参数	volatile struct RVM_List* New 指向要插入的新节点的指针。
	volatile struct RVM_List* Prev 指向要被插入的位置的前继节点的指针。
	volatile struct RVM_List* Next 指向要被插入的位置的后继节点的指针。

2.9.4 打印单个字符

本操作打印一个字符到控制台。在该函数的实现中，只需要重定向其输出到某外设即可。最常见的此类设备即是串口。这个函数在 RVM 移植时是由移植者负责提供的。

表 2-11 打印单个字符

函数原型	ptr_t RVM_Putchar(char Char)
意义	输出一个字符到控制台。
返回值	ptr_t 总是返回 0。

参数	char Char 要输出到系统控制台的字符。

2.9.5 打印整形数字

本操作以包含符号的十进制打印一个机器字长的整形数字到调试控制台。

表 2-12 打印整形数字

函数原型	cnt_t RVM_Print_Int(cnt_t Int)
返回值	cnt_t 返回打印的字符数量。
参数	cnt_t Int 要打印的整形数字。

2.9.6 打印无符号整形数字

本操作以无前缀十六进制打印一个机器字长的无符号整形数字到调试控制台。

表 2-13 打印无符号整形数字

函数原型	cnt_t RVM_Print_Uint(ptr_t Uint)
返回值	cnt_t 返回打印的字符数量。
参数	ptr_t Uint 要打印的无符号整形数字。

2.9.7 打印字符串

本操作打印一个最长不超过 255 字符的字符串到调试控制台。

表 2-14 打印字符串

函数原型	cnt_t RVM_Print_String(s8_t* String)
返回值	cnt_t 返回打印的字符数量。
参数	s8_t* String 要打印的字符串。

2.9.8 初始化线程栈

在部分架构上[1]，硬件中断机制会在栈上保存一部分包含了一小部分寄存器活动记录，因此在使用该栈之前需要使用本操作将其初始化。初始化一次后的栈在之后的使用中无需再次初始化。此外，该函数也负责根据传入的栈内存区域起始地址和大小，返回真正要用来初始化线程堆栈指针的值。这个函数在移植 RVM 时是由移植者负责提供的。

表 2-15 初始化线程栈

函数原型	ptr_t RVM_Stack_Init(ptr_t Stack, ptr_t Size)
返回值	ptr_t 要用来初始化线程寄存器组的堆栈指针的地址。
参数	ptr_t Stack 线程堆栈内存区域的起始地址。
	ptr_t Size 线程堆栈内存区域的大小。

2.10 本章参考文献

无

[1] 如 Cortex-M

第 3 章　虚拟机监视器架构概述

3.1　简介

本虚拟机监视器是基于准虚拟化技术实现的。它实现了虚拟化的所有基本功能：内存管理、CPU 管理、中断向量虚拟化和虚拟机错误处理。关于这些实现的具体信息请参见下文所述。虚拟化的总架构见下图。图中实现代表硬件隔离机制，虚线代表软件模组的边界。

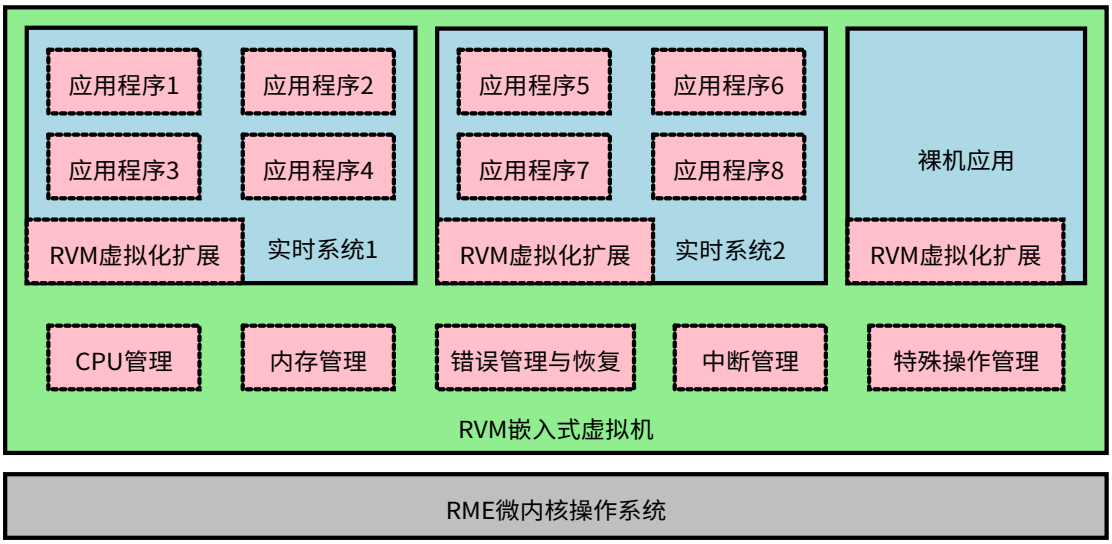

图 3-1　RVM 架构简图

在 RVM 内部，各个虚拟机也有它们各自的优先级。这个优先级是静态指定的。不同优先级的虚拟机之间采取抢占式调度策略，相同优先级的虚拟机之间采取时间片轮转调度策略。

3.2　虚拟机映像

在 RVM 中，每个虚拟机被表示成一个映像。这些映像都有自己的映像头，里面储存有关于此映像的信息。这些信息包括虚拟机的起始执行地址、起始栈地址、中断标志数组地址、参数传递区地址、所使用的内存区域和所使用的特殊内核调用等等。这些信息帮助虚拟机监视器正确加载和配置各个虚拟机。虚拟机映像头的地址被放置于每个虚拟机的映像的第一个字，虚拟机监视器会从这里加载它们。每个映像头还包括一个指针，这个指针指向下一个虚拟机映像所存放的位置。这样，虚拟机监视器就可以一个接一个地加载虚拟机映像了。关于虚拟机映像的信息，请参看第四章。

3.3　虚拟化架构详述

RVM 在启动后会读取所有的虚拟机映像，加载各个虚拟机。然后，四个守护进程会启动，并且负责对虚拟机提供服务。这四个守护进程分别是错误处理守护进程，中断分发守护进程，调度器守护进程和

定时器守护进程。其中，错误处理守护进程会在虚拟机发生错误时直接重启虚拟机本身[1]；中断分发守护进程会把来自设备的中断重定向给对应虚拟机的虚拟中断向量；调度器守护进程负责调度各个虚拟机；定时器守护进程负责向各个虚拟机发送时钟中断。每一个虚拟机内部则运行两个线程，一个负责处理虚拟机内部的中断，另一个则负责运行用户代码。具体的虚拟化原理会在以下各小节分别介绍。

3.3.1 优先级配置

在 RVM 的四个守护进程中，错误处理守护进程永远运行在系统的最高优先级。紧随其后的是 RVM 中的用户定义快速中断处理线程，它们负责快速处理系统中的部分来不及发送给虚拟机处理的中断。再接下来是其他三个守护进程，它们的优先级要求比虚拟机线程高。然后则是在运行的虚拟机的中断处理线程和用户线程，其中中断处理线程的优先级又要高于用户处理线程。之后是系统本身的 Init 进程，该进程负责无限循环调用空闲钩子，在没有虚拟机就绪时让系统进入休眠。最后则是未在运行的虚拟机的中断处理线程和用户程序线程。整个虚拟化体系的各线程的中断优先级如下所示：

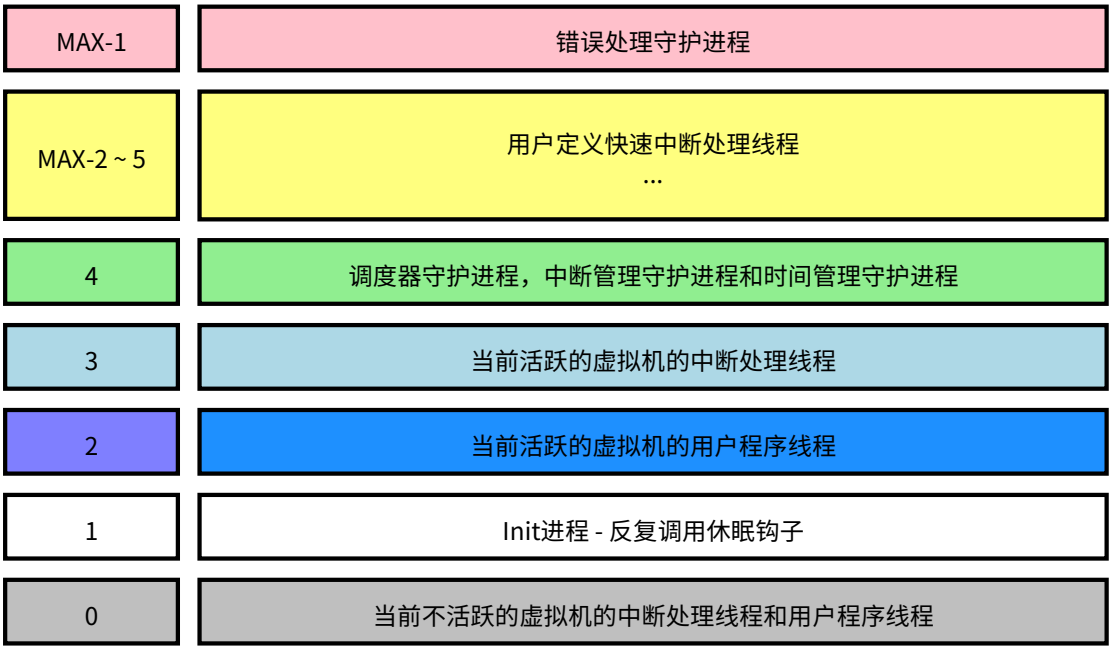

图 3-2 虚拟化体系的中断优先级

各个虚拟机采用固定优先级时间片轮转法调度。它们之间的优先级则是由调度器守护进程决定的。需要注意的是，为了系统安全，虚拟机的时间片和优先级是在系统编译时静态决定好的，RVM 并不提供可以修改这两点的超级调用。这从根本上保证了虚拟机之间的时间隔离，使得没有虚拟机可以破坏整个系统的实时性。

3.3.2 内存管理

[1] RVM 不支持用户自定义错误处理

在虚拟机中，最重要的事项之一就是内存管理。考虑到微控制器应用的一般特点，每个虚拟机的内存，包括私有内存和共享内存，都是在运行之前静态指定的。在虚拟机开始运行之后，不允许再新申请内存。这些内存区域的信息会以页表信息的形式放置在虚拟机的映像头内部。如果处理器仅仅支持一级 MPU，在虚拟机内部则不提供特殊手段设立二级保护区域；如果处理器支持二级 MPU，那么需要各个客户机自行设置自己的二级 MPU。对于每一种处理器，页表信息的格式都是不同的。请参看相应处理器的使用手册确定该处理器的页表形式。

3.3.3 CPU 的虚拟化

RVM 采用双线程的方法进行 CPU 虚拟化。一个线程负责中断管理，另外一个线程则负责处理用户程序。负责中断管理的线程平时不运行，只有当接收到了来自虚拟机监视器的中断信号时才运行。当中断处理完成后，中断处理线程又阻塞，用户程序线程开始运行。

如果需要在中断线程中修改用户线程的寄存器组，由于用户线程的寄存器组在它被切换出时会被保存在某个可指定的地址，因此只要修改那个地址上的寄存器数据即可。具体的修改方法请参见第四章。

3.4 中断的虚拟化和虚拟机间通信

对于虚拟机而言，中断虚拟化是必不可少的。系统中的中断大概可以分成三类，第一类是时钟中断，它负责向操作系统提供时钟嘀嗒；第二类是系统调用中断，它负责响应操作系统的系统调用；第三类则是其他系统中断，它们负责响应其他外设的事务。这三类中断的特点也不同：时钟中断是周期性频繁发生的；系统调用中断本质上是自己发送给自己的软中断；其他系统中断则是不时发生的无规律中断。因此，RVM 对于这三种中断采取了不同的虚拟化策略。

3.4.1 时钟中断的虚拟化

时钟中断由 RME 的底层定时器产生，并且通过内核端点的方式发送到 RVM 的时钟处理守护线程。该守护线程会计算当前经过的时间周期数，并且按时给各个虚拟机发送时钟中断。各个虚拟机的中断处理线程收到时钟中断后负责处理自身内部的定时器，并且调度其线程。

3.4.2 系统调用中断的虚拟化

系统调用中断本质上是一个软中断，它来源于虚拟机自身的内部。这个中断将直接被送往虚拟机的中断线程进行处理，不会经过 RVM。

3.4.3 其他中断的虚拟化

由于不同的虚拟机需要接收不同的外设中断，因此其他外设中断要按需送往各个虚拟机。因此，系统提供了调用接口来让各个虚拟机设置自己的每个虚拟中断向量的物理中断源。每个虚拟机能且只能设置自己的虚拟中断向量对应的物理中断源。一旦该物理中断被触发，登记于这个物理中断上的所有的虚拟机的相应虚拟中断向量均会收到此中断。

3.4.4 虚拟机间通信

虚拟机间以事件的形式进行通信。任何一个虚拟机都能且只能设置自己的各个虚拟中断向量对应的事件源。一旦其他虚拟机发送该事件，在接收的目标虚拟机中该事件会表现为该中断源被触发。关于如何使用事件源，请参看第四章。

3.5 特权级操作

在虚拟机中，有时需要执行一些特权级操作。因此，RVM 提供了一种手段来让虚拟机进行一些在其映像文件头中定义好的特权级操作。虚拟机只能进行这些被定义好的特权级操作而不能进行其他操作，因此这样做仍然是安全的。关于如何定义特权级操作，请参见第四章。

3.6 本章参考文献

无

第 4 章　移植 RTOS 和裸机程序到 RVM

4.1　简介

将一个 RTOS 或裸机程序修改以使其能够在 RVM 上运行的过程称为移植。移植可以分为底层库的配置、CPU 的虚拟化和虚拟机间通信三个部分。对于每种不同的架构，RVM 提供了不同的底层库，其内部有预先实现好的各类函数来帮助用户完成移植。对于常见的操作系统如 FreeRTOS、uC/OS 和 RT-Thread 等，RVM 也提供了一个简单的基本移植。

要使用底层库，首先确定要使用的架构，然后将 Guest 文件夹下该架构的底层库拷贝到欲虚拟化的操作系统的对应硬件抽象层中，并从这里开始替换掉该系统的原本底层操作。对于每种架构，其底层库均包含三个文件：rvm_guest_xxx.c，rvm_guest_xxx.h，rvm_guest_xxx_conf.h，它们应该被放置在同一个文件夹中。C 文件要加入 RTOS 的工程中，H 文件要被包含进任何希望使用 RVM 底层库的函数中。rvm_guest_xxx_conf.h 中包含的宏定义是用户可以修改的，它们决定了该虚拟机的配置。

4.2　底层库的配置

RVM 底层库的配置文件包括如下宏定义和数据结构。正确配置这些宏定义和数据结构在编译时是必须的。

4.2.1　宏定义

RVM 配置文件中包含的宏定义如下所示：

表 4-1　RVM 配置文件中包含的宏定义

宏名称	作用
RVM_VM_NAME	虚拟机的名称，一个长度不超过 16 的字符串。该长度包括了字符串结尾的“\0”字符。 例子： #define RVM_VM_NAME　"Dom1"
RVM_MAX_INTVECT	本虚拟机内部的最大中断向量数目。这个宏决定了本虚拟机有多少个虚拟中断向量。 例子： #define RVM_MAX_INTVECT　32
RVM_DEBUG_MAX_STR	调试打印的字符串的最长大小。该大小决定了 RVM 底层库提供给该虚拟机的调试打印区的长度。 例子： #define RVM_DEBUG_MAX_STR　128
RVM_USER_STACK_SIZE	虚拟机内部的用户线程堆栈大小，单位为字节。对于裸机应用程序，这个大

宏名称	作用
	小应当被设置为后台线程栈的实际尺寸;对于 RTOS 这个值则可以设置得较小些，因为通常而言 RTOS 会为自己的任务单独分配新的栈。 例子： #define RVM_USER_STACK_SIZE 0x100
RVM_INT_STACK_SIZE	虚拟机内部的中断线程堆栈大小，单位为字节。这个大小应当被设置为中断处理线程栈的实际尺寸。 例子： #define RVM_INT_STACK_SIZE 0x400
RVM_VM_PRIO	虚拟机的静态优先级。该数值越大则优先级越高，而且该数值不能超过 RVM 编译时支持的最大优先级数量，也即 RVM_MAX_PREEMPT_PRIO-1。 例子： #define RVM_VM_PRIO 1
RVM_VM_SLICES	该虚拟机的时间片数量。在 RVM 内部相同优先级的虚拟机采用时间片轮转调度，该值即决定了该虚拟机的时间片数量。该时间片的单位是由 RVM 下层的 RME 微内核操作系统的时钟中断频率决定的。 例子： #define RVM_VM_SLICES 10
RVM_PGTBL_NUM	本虚拟机的页表的数量。这个宏决定了页表配置数组的大小。关于页表配置数组的填充，请查看 4.2.2 节相关信息。 例子： #define RVM_PGTBL_NUM 1
RVM_KCAP_NUM	本虚拟机使用到的内核功能调用的数量。这个宏决定了内核功能调用配置数组的大小。关于内核功能调用配置数组的填充，请查看 4.2.2 节相关内容。如果该宏被定义为 0，那么该虚拟机不使用内核特权功能调用。 例子： #define RVM_KCAP_NUM 1
RVM_NEXT_IMAGE	下一个虚拟机映像的存放位置。这个位置应该填写一个十六进制值。如果该值为 0，那么意味着本虚拟机是系统中最后一个虚拟机。 例子： #define RVM_NEXT_IMAGE 0x08020000

4.2.2 数组定义

在配置头文件中还要额外定义两个数组，以确定本虚拟机的内存存取特权和内核功能调用特权。这两个结构体数组分别是页表配置结构体数组和内核功能调用数组。关于它们的具体信息如下所示：

4.2.2.1 页表配置结构体数组

由于每种架构的内存保护单元的结构均不相同，因此页表管理数组的结构也是各不相同的。关于每个架构的内存保护单元的详细信息，请查看该架构对应的 RVM 手册章节。

4.2.2.2 内核功能调用数组

内核功能调用数组储存一系列内核功能调用的调用号。在虚拟机被加载时，这些内核功能调用号对应的内核功能调用权能会被传递到该虚拟机的权能表内，此时即可使用内核功能调用来执行一系列特权级操作。RVM 规定了两类标准内核功能调用：第一类是中断管理相关的内核功能调用，其内核功能调用号从 0 到 512，对应第 0 个到第 511 个物理中断源；第二类是功耗管理相关的内核功能调用，其内核功能调用号为 512。512 号之后的内核功能调用可能随着各种架构的不同而有所不同；关于具体的信息请查看该架构对应的 RVM 手册章节。

通常而言本数组内部包含了允许本虚拟机操作的物理中断的中断号。关于如何使用这些中断号，请参见 4.3 节的相关内容。

4.3 CPU 的虚拟化

CPU 的虚拟化主要需要关注三个部分。第一个部分是 RVM 底层库的初始化，第二个部分是时钟中断的虚拟化，第三个部分是系统调用中断的虚拟化，第四个部分是开关中断函数的虚拟化，第五个部分是外设中断的虚拟化。关于这五个部分我们如下一一介绍。

4.3.1 RVM 底层库的初始化

在虚拟机的执行进入 main 函数后，应当调用 RVM_Init 函数进行虚拟机底层的初始化。这个函数会初始化全部的 RVM 底层库中的变量。

表 4-2 RVM 底层库的初始化

函数原型	void RVM_Init(void)
返回值	无。
参数	无。

4.3.2 虚拟机中断向量表的初始化

在完成虚拟机底层的初始化后，需要填充该虚拟机自身的中断向量表。这个中断向量表和物理中断向量没有任何关系，仅仅用于该虚拟机内部的中断号到中断向量的映射。注册和反注册中断是使用 RVM_Vect_Init 函数进行的：

表 4-3 虚拟机中断向量表的初始化

函数原型	ret_t RVM_Vect_Init(ptr_t Num, void* Handler)
返回值	ret_t 如果成功，返回 0；如果失败返回-1。
参数	ptr_t Num 虚拟机的虚拟中断号。这个中断号的必须在 0 到 RVM_MAX_INTVECT-1 之间。
	void* Handler 要为这个虚拟中断号注册的中断处理函数。该函数可以带一个参数，中断处理函数被调用时，该参数即为该中断的中断号。如果这个参数被置为 0，那么意味着清空该中断向量的位置，该中断向量将被除能。

需要注意的是，时钟中断和系统调用中断的向量必须被注册。时钟中断的向量号永远是 0，系统调用中断的向量号永远是 1。关于时钟中断和系统调用中断的具体信息请参见 4.3.3 节和 4.3.4 节。

在进行系统初始化工作之前，如果需要，可以调用函数 RVM_Disable_Int 来关闭中断；

表 4-4 关闭中断

函数原型	void RVM_Disable_Int(void)
返回值	无。
参数	无。

在完成所有的系统初始化工作之后，需要调用函数 RVM_Enable_Int 来开启中断。

表 4-5 开启中断

函数原型	void RVM_Enable_Int(void)
返回值	无。
参数	无。

4.3.3 时钟中断的虚拟化

通常而言，RTOS 都需要一个周期性的时钟中断来帮助其进行任务调度和维护内部的定时器列表。在 RVM 中，仅需要将其原时钟中断处理函数注册到中断向量号 0 即可在系统启动完成、中断被使能后接收到这个中断。时钟中断的频率可以通过函数 RVM_Tim_Prog 进行配置。

表 4-6 时钟中断的虚拟化

函数原型	ret_t RVM_Tim_Prog(ptr_t Period)
返回值	ret_t

	如果成功，返回 0；如果失败则返回-1。
参数	ptr_t Period 时钟中断的周期，单位是底层 RME 操作系统的时钟中断周期。

4.3.4 系统调用中断的虚拟化

一般的 RTOS 都有一个中断向量专门负责系统中任务的切换[1]。在 RVM 中，仅需要将原向量注册到中断向量号 1 即可接收到此中断。

对于 RTOS，这个中断向量一般用汇编写成，并且需要保存和恢复用户线程的寄存器组。在 RVM 上，由于用户线程在停止执行时其寄存器组已经被保存在了 RVM_Regs 结构体内，因此只要操作这个结构体内部的寄存器即可。因此，整个线程切换过程无需使用汇编语言编写，而可以使用 C 语言编写。

对于每个架构，RVM_Regs 结构体的内容都是不一样的。关于具体信息请查看每个架构对应的 RVM 手册章节。

4.3.5 开关中断函数的虚拟化

一般的 RTOS 都会有两个函数来开中断或者关中断。在 RVM 底层库中，可以使用函数 RVM_Disable_Int 或 RVM_Enable_Int 来关闭或开启虚拟机的中断。这两个函数会通知虚拟机监视器彻底关闭向虚拟机发送的全部中断。关于这两个函数的详细介绍请查看 4.3.2 节相关内容。

此外，RVM 还提供了两个函数用来暂时掩蔽中断。它们拥有比开关中断函数高得多的效率，应当在可以使用时优先使用。这两个函数并不会通知虚拟机监视器彻底禁止向虚拟机发送的全部中断，但是这些中断将被悬起并等待解除掩蔽时一并处理。如果彻底禁止虚拟机监视器发来的中断是不必要的[2]，可以使用这一对函数。

4.3.5.1 掩蔽中断

表 4-7 掩蔽中断

函数原型	void RVM_Mask_Int(void)
返回值	无。
参数	无。

4.3.5.2 解除中断掩蔽

表 4-8 解除中断掩蔽

函数原型	void RVM_Unmask_Int(void)

[1] 比如 Cortex-M 系列上的 PendSV 向量。

[2] 例如在 RTOS 的临界区域保护中

返回值	无。
参数	无。

4.3.6 外设中断的虚拟化

RTOS 一般都提供了一些函数来从中断中发送信息，比如从中断内向线程邮箱或是某信号量发送。要在虚拟化的 RTOS 之中使用外设中断，首先需要配置虚拟机的中断向量表。关于中断向量表的配置请参见 4.3.2 节。

在虚拟机中断向量表配置完成后，还需要进行从物理中断到虚拟中断的映射；如果物理中断源的优先级未配置或未被使能，还要配置物理中断的优先级或者使能物理中断。当不需要使用某个物理中断时则可以将该物理中断关闭。

4.3.6.1 映射物理中断到虚拟中断

要进行物理中断到虚拟中断的映射，可以使用函数 RVM_Reg_Int。

表 4-9 映射物理中断到虚拟中断

函数原型	ret_t RVM_Reg_Int(ptr_t Vect_Num, ptr_t Int_Num)
返回值	ret_t 如果成功，返回中断注册号，该号码可以用于取消映射；如果失败则返回-1。
参数	ptr_t Vect_Num 虚拟机的虚拟中断号。
	ptr_t Int_Num 物理中断的中断号在内核功能调用数组中的位置。

4.3.6.2 取消物理中断到虚拟中断的映射

要取消物理中断到虚拟中断的映射，可以使用函数 RVM_Del_Int。

表 4-10 取消物理中断到虚拟中断的映射

函数原型	ret_t RVM_Del_Int(cnt_t Int_ID)
返回值	ret_t 如果成功，0；如果失败则返回-1。
参数	cnt_t Int_ID 由 RVM_Reg_Int 返回的中断注册号码。

需要注意的是，任何一个物理中断都只能在一个虚拟机中被映射一次。进行映射时如果检测到之前有同样的物理中断被映射到本虚拟机的任何中断向量，都会返回失败。

4.3.6.3 使能物理中断源

要使能一个物理中断源，需要使用函数 RVM_HW_Int_Enable。

表 4-11 使能物理中断源

函数原型	ret_t RVM_HW_Int_Enable(ptr_t Int_ID)
返回值	ret_t 如果成功，0；如果失败则返回-1。
参数	ptr_t Int_ID 物理中断的中断号在内核功能调用数组中的位置。

4.3.6.4 除能物理中断源

要除能一个物理中断源，需要使用函数 RVM_HW_Int_Disable。

函数原型	ret_t RVM_HW_Int_Disable(ptr_t Int_ID)
返回值	ret_t 如果成功，0；如果失败则返回-1。
参数	ptr_t Int_ID 物理中断的中断号在内核功能调用数组中的位置。

4.3.6.5 设置物理中断源的中断优先级

要设置一个物理中断源的中断优先级，需要使用函数 RVM_HW_Int_Prio。

表 4-12 设置物理中断源的中断优先级

函数原型	ret_t RVM_HW_Int_Prio(ptr_t Int_ID, ptr_t Prio)
返回值	ret_t 如果成功，0；如果失败则返回-1。
参数	ptr_t Int_ID 物理中断的中断号在内核功能调用数组中的位置。
	ptr_t Prio 要设置给该物理中断的优先级。

4.4 虚拟机间通信

虚拟机间的通信是以事件的形式进行的。虚拟机之间可以互相发送事件。从源虚拟机发送向目标虚拟机的事件在目标虚拟机内部表现为一个中断信号。要创建事件通道，需要目标虚拟机声明允许源虚拟机向其发送事件，并且要指明该事件对应的中断向量。

4.4.1 注册事件

事件的注册由目标虚拟机通过调用函数 RVM_Reg_Evt 完成。一个虚拟中断向量可以被注册为能够接收来自多个虚拟机的事件，不同的虚拟中断向量也允许接收来自同一个虚拟机的事件。

表 4-13 注册事件

函数原型	ret_t RVM_Reg_Evt(ptr_t Int_Num, ptr_t VMID)
返回值	ret_t 如果成功，返回事件注册号，该号码可以用于取消注册；如果失败则返回-1。
参数	ptr_t Int_Num 该事件要被送到的虚拟中断向量的向量号。
	ptr_t VMID 允许向目标虚拟机发送中断的源虚拟机的虚拟机 ID。关于如何得到这个 ID，请参考 4.4.6 节所述。

4.4.2 删除事件

事件的删除必须由创建了该事件的虚拟机通过调用 RVM_Del_Evt 函数进行。

表 4-14 删除事件

函数原型	ret_t RVM_Del_Evt(cnt_t Evt_ID)
返回值	ret_t 如果成功，返回 0；如果失败则返回-1。
参数	cnt_t Evt_ID 由 RVM_Reg_Evt 返回的事件注册号码。

4.4.3 等待事件

要暂停虚拟机的运行并等待一个中断来临，需要调用函数 RVM_Wait_Evt。RTOS 通常都有一个空闲线程或者空闲钩子函数；此时要在该钩子函数内部反复调用此函数，以在本虚拟机空闲时释放 CPU 资源给更低优先级的虚拟机运行。

表 4-15 等待事件

函数原型	ret_t RVM_Wait_Evt(void)
返回值	ret_t 如果成功，返回 0；如果失败则返回-1。
参数	无。

4.4.4 发送事件

要向一个目标虚拟机发送事件，需要调用函数 RVM_Send_Evt。

表 4-16 发送事件

函数原型	ret_t RVM_Send_Evt(ptr_t Evt_ID)
返回值	ret_t 如果成功，返回 0；如果失败则返回-1。
参数	ptr_t Evt_ID 该事件的事件注册号。在源虚拟机内部它可以通过 RVM_Query_Evt 函数查找得到。

4.4.5 查找事件

要通过虚拟机的虚拟机 ID 查找事件的注册号，可以通过 RVM_Query_Evt 函数进行。

表 4-17 查找事件

函数原型	ret_t RVM_Query_Evt(ptr_t VMID)
返回值	ret_t 如果成功，返回第一个事件的注册号[1]；如果失败则返回-1。
参数	ptr_t VMID 目标虚拟机的虚拟机 ID。

这个调用不是 O(1)实时的，它的用时与被查询的虚拟机上注册的事件数量有关。如果这个值很大，那么该超调用将影响系统的实时性。因此，仅推荐在系统故障恢复时和系统启动时进行该查找。

4.4.6 查找虚拟机 ID

要通过虚拟机的字符串名称查找虚拟机的 ID，可以通过 RVM_Query_VM 进行。

表 4-18 查找虚拟机 ID

函数原型	ret_t RVM_Query_VM(char* Name)

[1] 可能在目标虚拟机内部有多个事件都允许该源虚拟机发送

返回值	ret_t 如果成功，返回虚拟机的 ID；如果失败则返回-1。
参数	char* Name 要查询的虚拟机的字符串名称，最长不超过 16（包括字符串结尾的‘\0’）。如果有多个同名的虚拟机，只有第一个虚拟机的 ID 会被返回。

这个调用不是 O(1)实时的，它的用时与系统中虚拟机的数量有关。如果这个值很大，那么该超调用将影响系统的实时性。因此，仅推荐在系统故障恢复时和系统启动时进行该查找。

4.5 本章参考文献

无

第 5 章　移植 RVM 到新架构

5.1　简介

将 RVM 移植到其他架构的过程称为 RVM 本身的移植。在移植之前，首先要确定 RME 已经被移植到该架构。RVM 的移植分为两个部分：第一个部分是 RVM 本身的移植，第二个部分是 RVM 客户机库的编写。

5.2　虚拟机监视器的移植

RVM 本身的移植包括其类型定义、宏定义、数据结构和函数体。

5.2.1　类型定义

对于每个架构/编译器，首先需要移植的部分就是 RVM 的类型定义。在定义类型定义时，需要逐一确认编译器的各个原始整数类型对应的处理器字长数。RVM 的类型定义一共有如下五个：

表 5-1　RVM 的类型定义

类型	作用
tid_t	线程号的类型。这个类型应该被 typedef 为与处理器字长相等的有符号整数。 例子：typedef tid_t long;
ptr_t	指针整数的类型。这个类型应该被 typedef 为与处理器字长相等的无符号整数。 例子：typedef ptr_t unsigned long;
cnt_t	计数变量的类型。这个类型应该被 typedef 为与处理器字长相等的有符号整数。 例子：typedef cnt_t long;
cid_t	权能号的类型。这个类型应该被 typedef 为与处理器字长相等的有符号整数。 例子：typedef cid_t long;
ret_t	函数返回值的类型。这个类型应该被 typedef 为与处理器字长相等的有符号整数。 例子：typedef ret_t long;

为了使得底层函数的编写更加方便，推荐使用如下的几个 typedef 来定义经常使用到的确定位数的整形。在定义这些整形时，也需要确定编译器的 char、short、int、long 等究竟是多少个机器字的长度。有些编译器不提供六十四位整数，那么这个类型可以略去。

表 5-2　常用的其他类型定义

类型	意义
s8_t	一个有符号八位整形。 例如：typedef char s8_t;

类型	意义
s16_t	一个有符号十六位整形。 例如：typedef short s16_t;
s32_t	一个有符号三十二位整形。 例如：typedef int s32_t;
s64_t	一个有符号六十四位整形。 例如：typedef long s64_t;
u8_t	一个无符号八位整形。 例如：typedef unsigned char u8_t;
u16_t	一个无符号十六位整形。 例如：typedef unsigned short u16_t;
u32_t	一个无符号三十二位整形。 例如：typedef unsigned int u32_t;
u64_t	一个有符号六十四位整形。 例如：typedef unsigned long u64_t;

5.2.2 和 RME 配置项有关的宏

下面列出的宏和 RME 的内核配置有关。它们需要与 RME 在编译时的选项维持一致。关于这些宏的具体意义在此不加介绍，请参看 RME 的手册中的对应宏。

表 5-3 和 RME 配置项有关的宏

宏名称	对应的 RME 宏
EXTERN	EXTERN
RVM_WORD_ORDER	RME_WORD_ORDER
RVM_KMEM_VA_START	RME_KMEM_VA_START
RVM_KMEM_SIZE	RME_KMEM_SIZE
RVM_KMEM_SLOT_ORDER	RME_KMEM_SLOT_ORDER
RVM_MAX_PREEMPT_PRIO	RME_MAX_PREEMPT_PRIO
RVM_KMEM_BOOT_FRONTIER	RME_KMEM_BOOT_FRONTIER
RVM_INT_FLAG_ADDR	RME_INT_FLAG_ADDR
RVM_PGTBL_WORD_SIZE_NOM(X)	RME_PGTBL_WORD_SIZE_NOM(X)
RVM_PGTBL_WORD_SIZE_TOP(X)	RME_PGTBL_WORD_SIZE_TOP(X)

此外，还有一些和架构有关的配置宏。这些配置宏也需要和内核中编译时确定的相关宏维持一致。

5.2.3 和 RVM 配置有关的宏

下面列出的宏和 RVM 的配置有关，可以根据 RVM 的配置状况灵活选择。

表 5-4 和 RVM 配置有关的宏

宏名称	作用
RVM_VMD_PRIO	RVM 的虚拟机监视器的优先级。RVM 的中断管理守护进程，虚拟机调度器守护进程和定时器守护进程都会运行在这个优先级。通常而言，这个值应当被配置为 4，因为不运行的虚拟机在优先级 0，Init 进程在优先级 1，在运行的虚拟机的用户线程和中断处理线程分别在优先级 2 和 3。 例子： #define RVM_VMD_PRIO 4
RVM_IMAGE_HEADER_START	第一个用户虚拟机映像所在的地址。 例子： #define RVM_IMAGE_HEADER_START 0x8020000
RVM_STACK_SAFE_SIZE	各虚拟机运行用线程栈的安全余量大小，单位是机器字。在使用栈的时候，总会在栈顶留下这个数量的空间以备万一。通常而言，设置为 0x20 即可。 例子： #define RVM_STACK_SAFE_SIZE 0x20
RVM_GUARD_STACK_SIZE	虚拟机监视器错误处理守护进程的栈大小。单位是字节。该值可根据需要配置。一般 2kB 即可。 例子： #define RVM_GUARD_STACK_SIZE 2048
RVM_TIMD_STACK_SIZE	虚拟机监视器定时器守护进程的栈大小。单位是字节。该值可根据需要配置。一般 2kB 即可。 例子： #define RVM_TIMD_STACK_SIZE 2048
RVM_VMMD_STACK_SIZE	虚拟机监视器调度器守护进程的栈大小。单位是字节。该值可根据需要配置。一般 2kB 即可。 例子： #define RVM_VMMD_STACK_SIZE 2048
RVM_INTD_STACK_SIZE	虚拟机监视器中断管理守护进程的栈大小。单位是字节。该值可根据需要配置。一般 2kB 即可。

宏名称	作用
	例子： #define RVM_INTD_STACK_SIZE 2048
RVM_MAX_VM_NUM	虚拟机监视器支持的最大虚拟机数量。这个值可以按需配置；通常而言，配置为 32 对小型嵌入式系统是很足够了。如果用不到这么多，可以将这个值配置的更低些，以节省内存。 例子： #define RVM_MAX_VM_NUM 32
RVM_MAX_PREEMPT_VPRIO	虚拟机监视器允许的最大优先级数量。通常配置为一个处理器字长。 例子： #define RVM_MAX_PREEMPT_VPRIO 32
RVM_INT_VECT_NUM	处理器本身具备的硬件外部中断的数量。 例子： #define RVM_INT_VECT_NUM 240
RVM_INT_MAP_NUM	系统中中断映射的总数量。每一个物理中断到虚拟中断的对应记作一次中断映射。通常而言，256 对于小型嵌入式系统是很足够了。 例子： #define RVM_MAX_INT_NUM 256
RVM_EVT_MAP_NUM	系统中注册事件的总数量。每一个事件到虚拟中断的对应记作一次事件映射。通常而言，256 对于小型嵌入式系统是很足够了。 例子： #define RVM_MAX_EVT_NUM 256
RVM_MAX_PERIOD	虚拟机时钟中断的最大间隔，单位是虚拟机监视器时间片。虚拟机不被允许将自己的时钟中断间隔设为比这个要大的值。 例子： #define RVM_MAX_PERIOD 10
RVM_MIN_PERIOD	虚拟机时钟中断的最小间隔，单位是虚拟机监视器时间片。虚拟机不被允许将自己的时钟中断间隔设为比这个要小的值。 例子： #define RVM_MIN_PERIOD 1
RVM_DEF_PERIOD	虚拟机时钟中断的默认间隔，单位是虚拟机监视器时间片。虚拟机在创建时其时钟中断间隔会被设为该值。 例子： #define RVM_DEF_PERIOD 1
RVM_DEBUG_LOG	虚拟机监视器调试输出使能。定义为 RVM_TRUE 使能，定义为

宏名称	作用
	RVM_FALSE 除能。 例子： #define RVM_DEBUG_LOG RVM_TRUE

5.2.4 数据结构定义

RVM 的数据结构只有两个，分别是 RVM_Regs 和 RVM_Hdr_Pgtbl。这两个结构体都与处理器架构有关。

5.2.4.1 RVM_Regs 的实现

RVM_Regs 是处理器的寄存器组结构体，用于在 RVM 中读取和修改虚拟机用户线程的寄存器组。这个结构体应当被定义为和内核中的寄存器组保存顺序一致。

5.2.4.2 RVM_Hdr_Pgtbl 的实现

RVM_Hdr_Pgtbl 是虚拟机文件头中用来保存页表信息的结构体。这个结构体是和 RVM 在该架构上的实现紧密相关的，请参看附录已获得相关信息。

5.2.5 汇编底层函数的移植

RVM 一共要实现以下 6 个汇编函数。这 6 个汇编函数主要负责 RVM 本身的启动和系统调用等。这些函数的列表如下：

表 5-5 RVM 移植涉及的汇编函数

函数	意义
_RVM_Entry	RVM 的入口函数。
_RVM_Jmp_Stub	线程创建和线程迁移用跳板。
_RVM_MSB_Get	得到一个字的最高位位置。
RVM_Inv_Act	进行线程迁移调用。
RVM_Inv_Ret	进行线程迁移返回。
RVM_Svc	调用 RME 的系统调用。

这些函数的具体实现细节和注意事项会被接下来的各段一一介绍。

5.2.5.1 _RVM_Entry 的实现

本函数是 RVM 的入口，负责初始化 RVM 的 C 运行时库并跳转到 RVM 的 main 函数。本函数必须被链接到 RVM 映像的头部，而且其链接地址必须与 RME 内核编译时确定的 Init 线程的入口相同。

表 5-6 _RVM_Entry 的实现

函数原型	void _RVM_Entry(void)
意义	RVM 的入口。
返回值	无。
参数	无。

5.2.5.2 _RVM_Jmp_Stub 的实现

本函数用于在线程第一次运行或者线程迁移调用运行时，跳转到真正的入口。对于通常的架构，这个函数不是必需的，只要将它实现为直接返回即可，对于这些架构实际上我们也不会使用该函数；对于那些如 Cortex-M 等中断退出时 PC 保存在用户栈上的架构，我们无法在创建新线程时通过设置线程的 PC 来确立返回地址。我们只能在 SRAM 中模拟一个返回堆栈，并借由返回堆栈中的 PC 值来设定将返回的地址。由于这个返回用堆栈是可能被反复使用的[1]，我们不希望频繁修改它内部所含的 PC 值。因此，我们将返回堆栈中的 PC 值设置为这个代码段。在这个代码段中，我们会给 PC 赋予线程创建时放在其他寄存器中的值，正确地向线程或者迁移调用传递其参数，并正式跳转到线程的入口执行。

表 5-7 _RVM_Jmp_Stub 的实现

函数原型	void _RVM_Jmp_Stub(void)
意义	辅助跳转到正确的线程或迁移调用入口。
返回值	无。
参数	无。

5.2.5.3 _RVM_MSB_Get 的实现

该函数返回该字最高位的位置。最高位的定义是第一个“1”出现的位置，位置是从 LSB 开始计算的[2]。比如该数为 32 位的 0x12345678，那么第一个“1”出现在第 28 位，这个函数就会返回 28。

表 5-8 _RVM_MSB_Get 的实现

函数原型	ptr_t _RVM_MSB_Get(ptr_t Val)
意义	得到一个与处理器字长相等的无符号数的最高位位置，也即其二进制表示从左向右数第一个数字“1”的位置。
返回值	ptr_t 返回第一个“1”的位置。

[1] 比如我们创建 A 线程，然后销毁 A 线程，再试图在同一个栈上创建 B 线程

[2] LSB 算作第 0 位

参数	ptr_t Val 要计算最高位位置的数字。

由于该函数需要被高效实现，因此其实现方法在不同的处理器上差别很大。对于那些提供了最高位计算指令的架构，直接以汇编形式实现本函数，使用该指令即可。对于那些提供了前导零计算指令（CLZ）的架构[1]，也可以用汇编函数先计算出前导零的数量，然后用处理器的字长-1（单位为 Bit）减去这个值。比如 0x12345678 的前导零一共有 3 个，用 31 减去 3 即得到 28。

对于那些没有实现特定指令的架构，推荐使用折半查找的方法。先判断一个字的高半字是否为 0，如果不为 0，再在这高半字中折半查找，如果为 0，那么在低半字中折半查找，直到确定第一个“1”的位置为止。在折半到 16 位或者 8 位时，可以使用一个查找表直接对应到第一个“1”在这 16 或 8 位中的相对位置，从而不需要再进行折半，然后综合各次折半的结果计算第一个“1”的位置即可。

5.2.5.4 RVM_Inv_Act 的实现

该函数进行一个标准线程迁移调用。在实现时，应当注意将参数按照内核调用约定按顺序放入正确的寄存器。

表 5-9 RVM_Inv_Act 的实现

函数原型	ret_t RVM_Inv_Act(cid_t Cap_Inv, ptr_t Param, ptr_t* Retval)
意义	进行一个标准的线程迁移调用。
返回值	ret_t 线程迁移系统调用本身的返回值。
参数	cid_t Cap_Inv 线程迁移调用权能的权能号。
	ptr_t Param 线程迁移调用的参数。
	ptr_t* Retval 该参数用于输出，是一个指向存放线程迁移调用返回值的指针。如果传入 0（NULL），那么表示不接收由 RVM_Inv_Ret 返回的返回值。

由于 RME 的内核在线程迁移调用中不负责保存除了控制流相关寄存器之外的任何寄存器，因此用户态要将它们压栈。如果用户态没有用到其中的几个寄存器，那么可以不将这些没用到的寄存器压栈。这个函数应当被实现为一个标准的迁移调用，也即压栈所有的通用寄存器组，但是不压栈协处理器寄存

[1] 如 ARM 等

器组。该函数还应当判断传入的 Retval 指针是否为 0（NULL），如果为 0 那么不输出线程迁移调用的返回值给 Retval 指向的地址。

5.2.5.5 RVM_Inv_Ret 的实现

该函数用于从线程迁移调用中返回。在实现时，应当注意将参数按照内核调用约定按顺序放入正确的寄存器。

表 5-10 RVM_Inv_Ret 的实现

函数原型	ret_t RVM_Inv_Ret(ptr_t Retval)
意义	从线程迁移调用中返回。
返回值	ret_t 如果成功，函数不会返回；如果失败，返回线程迁移返回系统调用的返回值。
参数	ptr_t Retval 线层迁移调用的返回值。如果调用端的 ptr_t* Retval 不为 0（NULL）的话，那么该参数将会被赋予调用端的*Retval。

5.2.5.6 RVM_Svc 的实现

该函数用于进行 RME 的系统调用。在实现时，应当注意将参数按照内核调用约定按顺序放入正确的寄存器。

函数原型	ret_t RVM_Svc(ptr_t Op_Capid, ptr_t Arg1, ptr_t Arg2, ptr_t Arg3)
意义	进行 RME 系统调用。
返回值	ret_t RME 系统调用的返回值。
参数	ptr_t Op_Capid 由系统调用号 N 和权能表权能号 C 合成的参数 P0。
	ptr_t Arg1 系统调用的第一个参数 P1。
	ptr_t Arg2 系统调用的第二个参数 P2。
	ptr_t Arg3 系统调用的第三个参数 P3。

5.2.6 C 语言函数的移植

RVM 一共有五个 C 语言函数，涵盖了底层调试打印、线程栈初始化、进入低功耗状态和页表设置等等。这些函数的列表如下：

表 5-11 RVM 移植涉及的 C 语言函数

函数	意义
RVM_Putchar	底层打印函数，打印一个字符到控制台。
RVM_Stack_Init	初始化线程的线程栈。
_RVM_Pgtbl_Setup	初始化虚拟机的一个页目录。
_RVM_Pgtbl_Check	检查内存是否在虚拟机允许访问的空间内。
RVM_Idle	将系统置于休眠状态。

关于底层调试打印函数，请参见 2.9.4；关于线程栈初始化，请参见 2.9.8。本节只介绍余下的函数。

5.2.6.1 _RVM_Pgtbl_Setup 的实现

该函数负责从虚拟机映像头中读取页表信息，初始化一个虚拟机的页表中的一个页目录。该函数还要将该目录构建到页表信息指定的上级页目录内部。

表 5-12 _RVM_Pgtbl_Setup 的实现

函数原型	void _RVM_Pgtbl_Setup(struct RVM_Hdr_Pgtbl* Pgtbl, ptr_t Pos, cid_t Cap_Captbl, ptr_t* Cap_Bump, cid_t Cap_Kmem, ptr_t* Kmem_Bump)
意义	初始化某虚拟机的页表。
返回值	无。
参数	struct RVM_Hdr_Pgtbl* Pgtbl 一个指向某虚拟机映像头中的页表描述信息数组的指针。该虚拟机的页表的描述就在这个数组中存储着。
	ptr_t Pos 当前要初始化的页目录在页表信息数组内部的存储位号。
	cid_t Cap_Captbl 用来存放页表权能的权能表的权能号。
	ptr_t* Cap_Bump 用来存放页表权能的权能表的权能分配指针。本函数应当在该权能表的*Cap_Bump 的位置创建一个页目录，并在创建完成后把*Cap_Bump 增加 1。
	cid_t Cap_Kmem

	用来创建页表的内核内存权能的权能号。
	ptr_t* Kmem_Bump 内核内存分配指针。本函数应当使用从*Kmem_Bump 起始的内核内存创建该页目录，并在创建完成后把*Kmem_Bump 增加该页目录的大小值。

5.2.6.2 _RVM_Pgtbl_Check 的实现

本函数负责检查虚拟机文件头中定义的和 RVM 共享的内存区域是否完全被该虚拟机允许访问的内存范围覆盖。由于 RVM 的调度、打印、中断传递等功能均需要在 RVM 和虚拟机之间共享内存，因此必须确保虚拟机声明的共享内存在自身可访问的内存范围内。

表 5-13 _RVM_Pgtbl_Check 的实现

函数原型	ret_t _RVM_Pgtbl_Check(const struct RVM_Hdr_Pgtbl* Pgtbl, ptr_t Pgtbl_Num, void* Addr, ptr_t Size)
意义	检查虚拟机文件头中定义的和 RVM 共享的内存区域是否完全合法。
返回值	ret_t 如果在范围内，返回 0；如果不在范围内，返回-1。
参数	struct RVM_Hdr_Pgtbl* Pgtbl 一个指向某虚拟机映像头中的页表描述信息数组的指针。该虚拟机的页表的描述就在这个数组中存储着。
	ptr_t Pgtbl_Num 页表描述信息数组的大小，也即其所包含的页目录数量。
	void* Addr 要检查的内存段的起始地址。
	ptr_t Size 要检查的内存段的长度。

5.2.6.3 RVM_Idle 的实现

本函数会由 RVM 的 Init 进程反复调用，在系统无负载时进入休眠状态。一般而言，该函数会直接调用一个 RME 内核功能，将处理器置为停止执行、等待中断的状态。

表 5-14 RVM_Idle 的实现

函数原型	void RVM_Idle(void)
意义	将系统进入休眠状态，直到下一个中断来临。
返回值	无。

参数	无。

5.3 RVM 客户机库的编写

RVM 客户机库的移植包括三个文件，分别是 rvm_guest_xxx.c, rvm_guest_xxx.h 和 rvm_guest_xxx.s。接下来我们讲解客户机库的组成部分和实现原理。关于具体的实现细节，可以参考已经实现好的客户机库，并且将其拷贝一份作为新的移植的起点。

5.3.1 客户机库的组成部分

客户机库主要负责客户机内部的 RVM 虚拟化扩展层服务。它可以从逻辑上主要分成三个部分：第一个部分是初始化部分，这一部分负责在虚拟机启动后初始化 RVM 的数据结构。第二个部分是虚拟化服务部分，这一部分负责向 RVM 发起超级调用完成特权操作。第三个部分是中断处理部分，这一部分负责处理 RVM 发向虚拟机的中断。

5.3.2 客户机库的实现原理

在 RVM 中，每个客户机都是单独的一个进程，并且拥有自己的权能表和页表。页表是在虚拟机创建之时就设置好的，因此无需客户机库关心；权能表在虚拟机被创建时也被创建好，而且内部有两个信号端点。

位于第一个位置[1]的信号端点是给虚拟机发起超级调用使用的。对于虚拟机而言，该端点只允许发送而不允许阻塞。一旦虚拟机向该端点发送了信号，那么阻塞在这个端点上的 RVM 调度器守护进程就会解除阻塞，抢占虚拟机的运行，并且从虚拟机和 RVM 的共享内存区读取参数开始执行超级调用。

位于第二个位置[2]的信号端点是给虚拟机的中断处理线程使用的。平时虚拟机的中断处理线程应该在该端点阻塞，其用户线程也可以向此端点发送。一旦有中断发生，该端点就会接受到从 RVM 的信号，此时中断处理线程要处理中断并负责维护用户线程的寄存器组。

5.3.3 客户机库初始化的实现

客户机库在初始化时，主要负责清空各个与 RVM 之间传递信息用的缓冲区。这些缓冲区包括参数传递缓冲区、中断标志传递缓冲区和中断向量地址缓冲区等。具体的缓冲区大小和变量名由移植者自行决定；关于具体的例子，可以参看任何一个现有的客户机库的实现。

5.3.4 虚拟化服务部分

这一部分主要包括了进行超级调用的方法及其封装。要进行超级调用，首先要把传入的参数放置在 RVM 与客户机共享的参数缓冲区内，然后向第一个[3]信号端点发送信号即可。具体的函数调用封装形式

[1] 权能号为 0

[2] 权能号为 1

[3] 权能号为 0

由移植者决定；一个推荐的实现标准是 4.3 节的客户机库接口描述。通常而言，向端点发送信号的部分会包含一部分汇编代码，此时这部分代码应该被放入汇编文件中；暴露出来的函数接口则应当放置在客户机库的头文件中。

5.3.5 中断处理部分

这一部分主要包括了中断线程的入口及其逻辑。中断线程平时在第二个[1]端点上阻塞。当它收到信号，就表明有中断待处理。它会先读取 RVM 传过来的的标志状态位，然后决定运行哪个中断处理程序，并清除相应的位。这个操作应当被反复进行，直到没有中断就绪为止。

5.4 本章参考文献

无

[1] 权能号为 1