RME General-Purpose RTOS
TRM

Rev. R3T2

突越·中生·真核（七阶）

Mutatus·Mesozoa·Eukaron (R.VII)

M7M1 (Eukaron) R3T2

Microkernel General-Purpose RTOS (Rev.3 Typ. 2)

Technical Reference Manual

System Features

Minimal Yet Efficient Kernel

- Less than 5000 lines of kernel code
- User-level scheduler augmented hierarchical scheduling

Broad Architectural Support

- Supports symmetric or asymmetric multi-processor architectures
- Supports uniform or non-uniform memory access architectures
- All common hardware features covered by kernel function calls

Mature Capability Management

- Capability-based 3rd generation lockless microkernel
- Genuine user-level and kernel-level separation

Flexible Memory Management

- Memory access controlled by normal or path-compressed page tables
- All memory mappings managed at user-level

Advanced Thread and Process Model

- Threads can be transferred between multiple processors
- Threads can migrate between multiple processes
- Can host virtual machines with full- or para-virtualization

Advanced Synchronous and Asynchronous Communication Model

- Inter-process communication with synchronous invocations
- Synchronize threads or process interrupts with asynchronous signals

Table of Contents

List of Tables

List of Figures

Revision History

Revision	Date (DD-MMM-YYYY)	Description
R1T1	12-Jul-2017	Initial release
R2T1	02-Dec-2017	Added porting directions
R3T1	04-Jun-2018	Added appendix
R3T2	17-Aug-2018	Added kernel function call section and updated the manual

Chapter 1 Introduction

1.1 Preface

In modern embedded systems, as the need for computation capability grows, the prevalence of multi-core systems is growing quickly, and the trend of asymmetric computation is also gaining popularity. In the meantime, as the amount of on-chip resource bloats, there are increasing needs for advanced memory management facilities. However, for multi-core systems, the real-time guarantees are often hampered by race conditions, which calls for lockless shared-nothing kernels; for MicroController Units (MCUs), their memory protection mechanism and memory layout is largely disparate and different from larger computation systems, which calls for a unified memory management paradigm to consolidate the different platforms.

In modern high-performance computing systems, the importance of light-weight virtualization also gained attention. In high-end servers, virtualization facilitates centralized resource management and allocation, and enables new possibilities such as hot-migration, which simplifies hardware maintenance; in high-performance embedded systems, virtualization makes it possible for MCUs and Digital Signal Processors (DSPs) to run third-party binary executables securely, or run multiple high-level language virtual machines simultaneously without causing security or access control problems. In both scenarios, it is mandated that the virtual machine should be real-time and highly efficient, while maintaining its scalability and expandability.

RME (RTOS-Mutate-Eukaron) is a general-purpose microkernel Real-Time Operating System (RTOS) that provides extreme expandability and scalability, while being fully-preemptive and highly efficient. It provided all features commonly found on 3rd-generation microkernels: flexible user-level scheduling, low-level memory management, highly-efficient communication mechanisms and the ability to harness special hardware acceleration mechanisms. RME is designed to run on MCUs that have 64kB Read-Only Memory (ROM) and 16kB Random-Access Memory (RAM), and high-performance servers that have hundreds of sockets and hundreds of TBs of memory, while still being as efficient as it is on MCUs.

1.1.1 Design Goals

As a 3rd-generation microkernel, the design goals of RME includes four aspects: flexibility, security, reliability and real-time responsiveness.

Flexibility suggests that the system can be applied to multiple scenarios, and push the different hardwares to their respective limits, while maintaining as much abstraction in common as possible. Thus, RME considered support for a range of platforms spanning from MCUs to

supercomputers at design time, and strict code quality control standards were imposed through the development process to fit any possible compiling environment.

Security means that all the resources in the systems have confidentiality, integrity and availability. To guarantee these properties, RME bases its security facility on capabilities. In the system, all operations are controlled and managed by the capabilities; if kernel objects are to be operated on, corresponding capabilities must be passed in in system calls. At user-level, different types of resource managers manage different categories of capabilities, and thus they are loosely coupled and highly compartmentalized.

Reliability refers to the fact that there are few or none functional bugs in the system, and upon any functional fault, a micro-reboot of subsystems involved can be conducted. As a microkernel, RME is inherently more reliable than monolithic kernels; to reduce the chance of any possible bug, full branch coverage white-box testing is conducted on the architectural dependent portion of RME.

Real-time responsiveness guarantees that all the execution paths are time-bounded and all of these bounds are low enough not to cause any problem. At design time, all the paths are designed to be deterministic, and the hard real-time responsiveness guarantees can be kept even under multi-core environments.

1.1.2 Software Copyright Notice and License

Taking the requirements of microcontroller applications, deeply embedded applications and high-performance general-purpose applications into consideration, RME adopted LGPLv3 as its main license. For some special cases[1], some special terms apply. These special terms will be different for each particular application.

1.1.3 Terms and Definitions

The terms and abbreviations used in this manuals are listed as follows:

1.1.3.1 Operating System

The lowest level software which is responsible for processor, memory and device management.

[1] e.g.security and medical equipment

1.1.3.2 Process

A minimal separated container that possess some resources. These resources can be some kernel objects, some memory or some device. Generally, this container will correspond to a instance of some executing program.

1.1.3.3 Thread

A control flow that have one standalone stack and can be scheduled independent of each other. There can be multiple threads in one process, and they share the same address space.

1.1.3.4 Coroutine

A control flow that only have a independent control flow but does not have a standalone stack. Multiple coroutines can reside in the same thread, and they share the same thread stack.

1.1.3.5 Static Allocation

All resource allocations are finished at compile time.

1.1.3.6 Semi-Static Allocation

All resource allocations are done at boot-time and not changed during run-time.

1.1.3.7 Dynamic Allocation

At least a part of the resources can be dynamically allocated and freed at run-time.

1.1.3.8 Soft Real-Time

A system that meets most of its deadline requirements. Some deadline misses are acceptable, provided that these cases are rare.

1.1.3.9 Hard Real-Time

A system that meets all of its deadline requirements. Any deadline misses are not allowed.

1.1.3.10 Constant Real-Time

All operations are O(1) with regards to user input and system configuration. The constant time factor must be reasonable and small enough. This is the strongest real-time guarantee.

1.1.3.11 Constant Real-Time to a Certain Value

All operations are O(1) when the value is given, and the constant factor must be reasonable and small enough.

1.1.4 Major Reference Systems

Capability table, signal endpoint and thread migration: Composite (@GWU)。

Page Table: Composite (@GWU)。

Dynamic page swapping: uCLinux (@Emcraft)。

Kernel memory capability:Fiasco.OC (@TU Dresden)。

Operation flags and maximum ceiling priority: seL4 (@Data61/CSIRO)。

Light-weight scheduling queues: RMProkaron (@EDI)。

System call interfaces: Linux (@The Linux Foundation/Linus Torvalds)。

Separation kernel implementation: Barrelfish (@Microsoft/ETH Zurich)。

All other references are listed in their respective chapters.

1.2 Background

Operating system is a kind of basic software that is responsible for CPU, memory and device management. For real-time operating systems, all operations must be predictable and always meet their respective deadlines. Generally there are two types of RTOSes: the former being soft real-time systems, which meets its deadline in most cases; the latter being hard real-time systems, which meets its deadline in all cases. Practically all embedded systems can be split in half, one part being soft real-time and the other part being hard real-time. One example is that the GUI part is usually soft real-time, and the motor controller part is usually hard real-time.

Generally speaking, almost all real-time systems are embedded systems, which refers to the systems that are specifically customized for one particular application. These applications may pose many restrictions on system performance, power, size and operating conditions. Typical such systems include highly specialized systems such as programmable logic controllers, flight data computers and rocket trajectory computers; there are also systems that are relative general-purpose, e.g. industry computers.

Traditionally, due to restrictions of hardware performance, real-time systems are relatively simple and it is viable to go with a simple operating system or without any operating system altogether. With the advent of new microcontrollers and microprocessors, the complexity of embedded systems have greatly increased, which necessitates new RTOSes.

The newer generation of RTOSes are supposed to be stronger than current RTOSes in terms of reliability, portability and flexibility; they should also have inherent design considerations respecting multi-core support and parallelism. Considering all these requirements, a microkernel design is the only choice. Microkernels usually implement a minimal set of primitives that can help export the most of operating system kernel services to the user level servers; should any of

these services fail, the fault can be contained within the server boundary and will not propagate to other modules. Different servers can be reboot independently of each other, and can have their own private copies for redundancy, both of which greatly boost system security and reliability.

Moreover, a microkernel design makes multi-core parallel design easier due to minimalism of the kernel, thus requiring less synchronization points. This also facilitates large employments Read-Copy-Update (RCU) techniques in the kernel. All capabilities are aligned to the cache-line boundary, thus the cache-line contention is reduced as much as possible to avoid false-sharing. This greatly boosts the performance under multi-socket Non-Unified Memory Access (NUMA) architectures.

We will review the categories of operating systems first. In this manual, we will divide all operating systems into five categories: dedicated systems, ultra-light-weight systems, basic real-time operating systems, general-purpose operating systems and general-purpose real-time operating systems. The details of these concepts are listed hereinafter.

1.2.1 Dedicated Systems

These types of systems are usually designed for a specific purpose and thus does not possess any system services and software abstraction layers found in common operating systems. There are no separation between kernel space and user space as well; the applications directly runs on the bare metal. Most unikernels falls well within this category, and most microcontroller frontend-backend bare-metal applications also falls within this category.

Typical such operating systems include Rump (unikernel) and Mirage OS (unikernel).

1.2.2 Ultra-Light-Weight Systems

Ultra-Light-Weight Systems (ULWS) are the minimal systems that can be regarded as real operating systems. They generally run on 8-bit or even 4-bit machines, and does not require a system timer; they have no user-level and kernel-level separation, and does not even require porting to run on multiple architectures. It usually consists of a few lines of code responsible for context switching, and does not need a customized linker script to compile an run.

A typical ULWS consists of only a while() main loop, in which it calls its task functions one by one. The task functions are implemented as state machines, and on each entrance of a task it will pick a state to run. All the tasks usually share the same stack, and are linked with the kernel statically. The tasks are cooperative coroutines and usually does not preempt each other. The tasks does not necessarily need to be reentrant.

The priority support of such operating systems is implemented with hardware interrupt priorities, and the IRQ handlers will process everything other than postponing processing to

another dedicated thread. The interrupts are completely transparent to the operating system, which means that the operating system is not interrupt aware at all.

Typical such systems include RMS (ultra-light-weight coroutine library) by EDI and Sloth (enhanced light-weight coroutine library) by FAU.

1.2.3 Basic Real-Time Systems

Basic hard-real-time operating systems are the systems which exhibit all the basic features of a RTOS. They are usually deployed on high-end 8/16-bit machines and low-end 32-bit machines, requiring a dedicated system tick timer. These systems do not have genuine kernel space and user space separation; however, it is possible to configure the Memory Protection Unit (MPU) or Memory Management Unit (MMU) to protect some ranges of memory. The hardware abstraction layer of these systems include some simple assembly for context switching, which must be modified when porting to other architectures. The porting usually also involves system tick timer, context switching, interrupt management and coprocessor management. Some of these systems can use a customized linker script; however this is not always necessary, except in the case of memory protection.

In these systems, a task is always a thread, and might be reentrant. Threads have their own private stacks. The application code can be either linked with the kernel or linked as standalone modules. There are no system calls and the system API uses just normal function calls.

These systems provide priorities, and the threads on the same priority level are scheduled with round-robin algorithm, while threads on different priorities will preempt each other. Some of these systems also have primitive memory management support, which is usually based on SLAB and buddy system.

Interrupts can be totally transparent to the system, and when so the system will be totally unaware about the interrupts. When it is needed to context switch in interrupt, we must insert context switching assembly into the interrupt routine, and some assembly prologue/epilogue is required.

Typical such systems include RMP, RT-Thread, FreeRTOS, uC/OS, Salvo and ChibiOS.

1.2.4 General-Purpose Operating Systems

General-purpose operating systems are systems that exhibit most features of a typical operating system but usually does not have outstanding real-time performance. These systems usually run on 32-bit or 64-bit machines and require a system timer. They enforce strict separation of user-level and kernel-level, and require hardware memory management facilities such as MMUs or MPUs. They also require sophisticated customized linker scripts, and complex procedures must be followed when porting them to new architectures. The porting usually

involves system timer, context/protection domain switch, interrupt management and coprocessor management.

Tasks present themselves as processes on such systems. One task can contain multiple threads. Due to the existence of virtual memory, there is no requirement regarding whether the task functions need to be reentrant or not. The kernel is usually compiled separately from the user applications, and the applications make system calls by using software interrupts or dedicated system call instructions[1].

These systems usually provides priorities; however, the real-time performance is not always guaranteed.

These systems usually feature two-level memory management; the OS is responsible for low-level page management, and the language run-time libraries are responsible for run-time heap & stack management.

Interrupts are not transparent to the systems. The OS requires that the context is saved and restored upon entering & exiting the interrupt vector; sometimes the co-processor context and MMU/MPU context must be saved or restored as well.

Typical such systems include Windows, Linux, Minix, FreeBSD, Mac OSX and Amiga.

1.2.5 General-Purpose Real-Time Operating Systems

General-purpose real-time operating systems are the most complex and powerful variant of all the operating systems. The most distinct feature of such systems is the added real-time guarantee, and the timing of all the kernel execution paths is completely predictable.

Typical such systems include RME, Composite, Fiasco.OC plus many L4 variants, RTLinux and VxWorks. RME is designed as such a system, thus its design considerations are the most complex and difficult.

In the kernel designs described in 1.2.4 and 1.2.5, two design patterns exist, respectively being monolithic kernels and microkernels. A typical implementation of the former is the Linux kernel, which links all its system services to the kernel level. The advantage of this is that all the cross-module calls are simple function calls and no protection domain crossing costs are incurred. Thus, these kernels are relatively faster and provides more functionality. A typical implementation of the latter is L4, which exports as many services to user-level as possible, and the kernel itself is only responsible for basics such as message-passing and scheduling. In later evolution of the designs, even scheduler and kernel memory management is placed at user-level. The benefit of such a design is that the modules are separated as radically as possible, and thus have better maintainability, security and reliability. It exposes a smaller attack surface for

[1] e.g. SYSCALL and SYSRET of x86-64

malicious hackers as well. Once necessary, the kernel can go through formal verification to guarantee its correctness[4]. The typical architecture of the two designs are as follows:

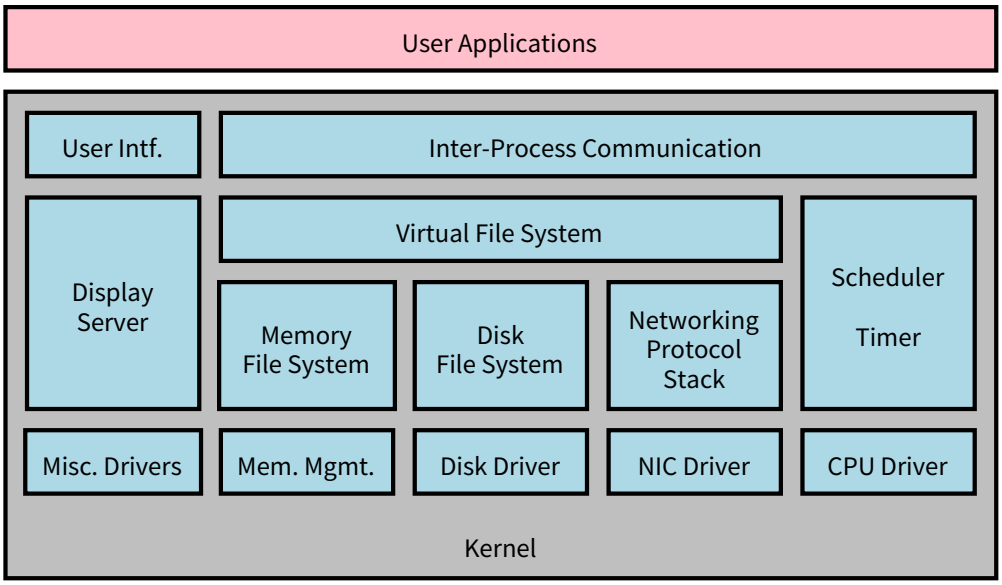

Figure 1-1 Monolithic Kernel Architecture

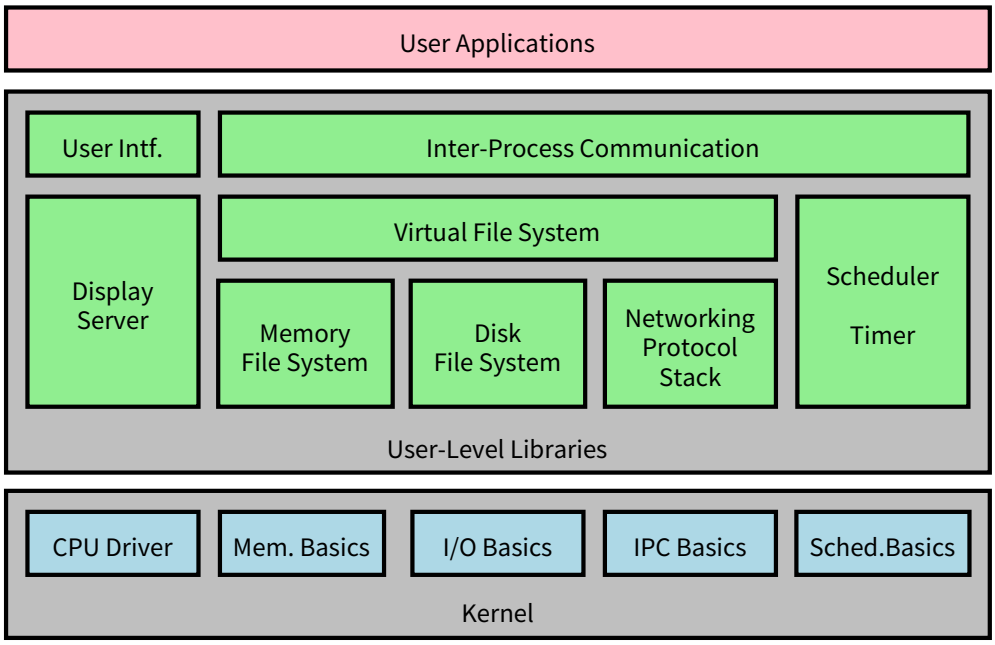

Figure 1-2 Microkernel Architecture

RME adopted the microkernel approach. On the one hand, the only advantage of monolithic kernels which is no need to frequently cross protection domains, has diminished on modern architectures[1]; on the other hand, the excellent cache and parallel performance of microkernels,

[1] e.g. the SYSCALL/SYSRET dedicated system call instruction pair costs only 60 cycles on latest x86-64 machines, which is equivalent to a single level 3 cache miss.

which is due to excellent code and data locality and simplicity, have constantly surpassed that of the monolithic kernels. Later microkernels such as seL4[4] can constantly outperform Linux tens of times in terms of Inter-Process Communication (IPC) performance; its worst-case response time is even better depending on the configuration. In terms of application performance, microkernel-backed Network Function Virtualization (NFV) applications also outperformed Linux by an order or magnitude, which have negated the assertion that monolithic kernels have better performance.

In terms of RME itself, it is 40 times faster than Linux in IPC performance on some single-core architectures[1]; when the overhead of user-level libraries are also included, RME is still 25 times faster than Linux.

1.3 Performance Specs of RTOSes and Its Components

There are hundreds of different RTOSes out there on the market, and kernels developed by hobbyists are virtually uncountable. The performance and reliability of these systems vary, and we need some measures to benchmark them. All the measurements listed below can only be directly compared against each other when the processor architecture, compiler, and compiling options are all the same. If different processors, compilers or compiler options are used, then the results cannot be directly compared to each other and can only serve as a reference. One recommended approach is to use industry-standard ARM or MIPS processors and GCC -O2 compilation option. Simulators such as Chronos are also useful. When performing the evaluation, system load will also influence the results, thus the system load must be the same when comparing the results.

1.3.1 Kernel Size

Kernel size is a key aspect of RTOSes. RTOSes are usually deployed on resource-constrained devices, thus a small kernel size is critical. The size of the kernel will be evaluated in two dimensions, respectively being the read-only memory size and read-write memory size. The read-only size includes the code and constant data segment, while the read-write size includes the modifiable data segment. On Flash-based MCUs, the read-only segments will consume Flash, and the read-write segments will consume SRAM[1].

RTOSes are highly configurable, thus their kernel size is rarely a fixed number and these numbers are tied closely to a specific configuration. Therefore, to measure this performance, you should measure the kernel size under the minimal kernel configuration, common kernel configuration, and full-featured kernel configuration[1].

[1] especially on microcontrollers, e.g. Cortex-M

Obtaining kernel size data is as simple as compiling the kernel with a compiler and then using a dedicated binary viewer[1] to inspect the size of each section of the target file.

1.3.2 Execution Time, Worst-Case Execution Time and Jitter

The execution time refers to the time consumption of a particular RTOS system call. The Worst Case Execution Time (WCET) refers to the maximum length of execution time under the most unfavorable conditions. WCETs is usually achieved when the most tedious system call is made and the most number of cache misses occur. RTOSes generally disable interrupts when executing of system calls; the worst execution time is usually the longest interval in which the system disables interrupts, so the impact of WCETs on the real-time performance is enormous.

These WCETs can be divided into two categories: the first is the WCETs of system calls, and the second is the WCETs of inter-thread synchronizations.

To get the first type of WCETs, before calling a system call, jot down the time stamp Ts at this time, and then after the end of the system call, read the timer to get the time stamp Te. Then, read the timer twice in a row to get two timestamps, Tts and Tte, and obtain the extra cost of reading the timer as Tte-Tts. In this case, the execution time is Te-Ts-(Tte-Tts). Repeat this on all system calls, then the WCET will be largest number among all measurements.

To get the second type of WCETs, jot down the timestamp Ts at the sending side of the communication channel, then at the receiving end, read the timer get the timestamp Te. The measurement for the cost of reading the timer is similar to the first-type WCETs. The resulting Te-Ts-(Tte-Tts) is the execution time. Repeat this on all communication mechanisms calls, then the WCET will be largest number among all measurements.

Jitter of execution time is also very important. We often get a distribution when conduct the same measurement multiple times. The average of this distribution is the execution time, and its standard deviation[2] is called the jitter.

For a RTOS, we usually wish the execution time, the WCET and the jitter to be as small as possible. Execution time can be further divided into the following categories in detail[1]:

1.3.2.1 Intra-Process Thread Context Switch Time

Intra-process thread context switch time refers to the total time to switch from one thread to another in the same process. We use the following method to measure this value. In the measurement, except for the method of using T_e-T_s, it is also possible to use the difference between two consecutive Ts divided by 2. There are two types of context switching, one is to

[1] Such as Objdump

[2] In some cases we also use range

switch between threads with the same priority, and the second is to wake up a high-priority thread in low-priority thread[2].

In the first case, we assume that the two threads have the same priority, and at the beginning of the measurement we are executing thread A, which was just switched to by thread B.

Table 1-1 The First Case of Intra-Process Thread Context Switch

Process 1 : Thread A	Process 1 : Thread B
Loop Forever	Loop Forever
{	{
>> Read T_s;	Read T_e;
Switch to thread B;	>> Switch to thread A;
}	}

In the second case, we assume that the thread B have a higher priority, and at the beginning of the measurement we are executing the thread A, which was just switched to by thread B.

Table 1-2 The Second Case of Intra-Process Thread Context Switch

Process 1 : Thread A	Process 1 : Thread B
Loop Forever	Loop Forever
{	{
>> Read T_s;	Read T_e;
Wake up (conventional) or switch to (microkernel) thread B;	>> Sleep (conventional) or switch to (microkernel) thread A;
}	}

1.3.2.2 Inter-Process Thread Context Switch Time

Inter-process thread context switch time refers to the total time to switch from one thread to another in different processes. The measurement method and two possible cases are similar to what is described in the previous section, with the only difference being the two threads now belong to different processes.[2]

In the first case, we assume that the two threads have the same priority, and at the beginning of the measurement we are executing thread A, which was just switched to by thread B.

Table 1-3 The First Case of Inter-Process Thread Context Switch

Process 1 : Thread A	Process 2 : Thread B
Loop Forever	Loop Forever
{	{
>> Read T_s;	Read T_e;
Switch to thread B;	>> Switch to thread A;
}	}

In the second case, we assume that the thread B have a higher priority, and at the beginning of the measurement we are executing the thread A, which was just switched to by thread B.

Table 1-4 The Second Case of Inter-Process Thread Context Switch

Process 1 : Thread A	Process 2 : Thread B
Loop Forever	Loop Forever
{	{
>> Read T_s;	Read T_e;
Wake up (conventional) or switch to (microkernel) thread B;	>> Sleep (conventional) or switch to (microkernel) thread A;
}	}

1.3.2.3 Intra-Process Thread Synchronous Communication Time

Intra-process thread synchronous communication time refers to the total time to do a synchronous communication between different threads in the same process. The measurement below assumes that the thread B have already blocked at the receive endpoint. Thread A will attempt to send to thread B, and thread B have a higher priority. For systems that employ thread migration for synchronous communication[1], this measurement is not necessary because there is no such need to use synchronous communication in the same process.[2]

Table 1-5 Intra-Process Thread Synchronous Communication Time

Process 1 : Thread A	Process 1 : Thread B
Loop forever	Loop forever
{	{
>> Read T_s;	Read T_e;

[1] Such as RME and some variants of L4

Process 1 : Thread A	Process 1 : Thread B
Send to synchronous endpoint P;	>> Receive from synchronous endpoint P;
}	}

1.3.2.4 Inter-Process Synchronous Communication Time

Inter-process synchronous communication time refers to the synchronous communication time between different threads across different processes. The measurement procedure below applies to traditional operating systems only. We assume that the thread B has already blocked at the receive endpoint, and thread A will attempt to send to it. We also assume that thread B have a higher priority than thread A.[2]

Table 1-6 Inter-Process Synchronous Communication Time for Conventional OSes

Process 1 : Thread A	Process 2 : Thread B
Loop forever	Loop forever
{	{
>> Read T_s;	Read T_e;
Send to synchronous endpoint P;	>> Receive from synchronous endpoint P;
}	}

For systems such as RME and some L4 variants that employ thread migration techniques, the appropriate measurement method is shown below. Note that when the thread A invokes function F, we are still running the thread A, but the execution is in the process 2, and the execution stack is newly allocated in process 2.

Table 1-7 Inter-Process Synchronous Communication Time for Invocation-based OSes

Process 1 : Thread A	Process 2 : Thread A
Loop forever	Function F
{	{
>> Read T_s;	Read T_e;
Invoke function F;	Return;
}	}

1.3.2.5 Intra-Thread Asynchronous Communication Time

Intra-thread asynchronous communication time refers to the total time to send and receive asynchronous signals in the same thread. Common asynchronous signals include RME's

asynchronous signal endpoints, or semaphores & mailboxes & message queues & pipes that other operating system may provide. The measurement method is shown below[2]. In the final measurement results, T_i-T_s is the time cost of the send operation, T_e-T_i is the time cost of the receive operation, and T_e-T_i is the total communication cost.

Table 1-8 Intra-Thread Asynchronous Communication Time

Thread A

Loop forever
{
Read T_s;
Send to asynchronous endpoint P;
Read T_i;
Receive from asynchronous endpoint P;
Read T_e;
}

1.3.2.6 Intra-Process Inter-Thread Asynchronous Communication Time

Intra-process inter-thread asynchronous communication time refers to the time to send and receive asynchronous signals between different threads in the same process. The measurement method is shown below. We assume that thread B is already blocked at the receive endpoint, and thread A will attempt to send to it. The thread B have a higher priority than thread A[2].

.

Table 1-9 Intra-Process Inter-Thread Asynchronous Communication Time

Process 1 : Thread A	Process 1 : Thread B
Loop forever	Loop forever
{	{
>> Read T_s;	Read T_e;
Send to asynchronous endpoint P;	>> Receive from asynchronous endpoint P;
}	}

1.3.2.7 Inter-Process Asynchronous Communication Time

Inter-process asynchronous communication time refers to the time to send and receive asynchronous signal across processes. The measurement method is the same as what is stated in the section above, with the only difference being that the two threads under test now belong to different processes[2].

Table 1-10 Inter-Process Asynchronous Communication Time

Process 1 : Thread A	Process 2 : Thread B
Loop forever	Loop forever
{	{
>> Read T_s;	Read T_e;
Send to asynchronous endpoint P;	>> Receive from asynchronous endpoint P;
}	}

1.3.2.8 Page Table Operation Time

Page table operation time refers to the time to perform a page table operation. Because operating systems differ greatly in the mechanism details and the functionality provided, a direct comparison is not possible. Usually a microkernel will allow direct control over physical memory, and a monolithic kernel will provide system calls to map pages to a processes' address space. Some microcontroller-targeted operating systems will provide block-based memory allocation operations. All in all, the method to measure such operations is fairly simple: read the timer before and after the system call, and the difference between these two timestamps is the operation overhead[1][2].

Table 1-11 Page Table Operation Time

Thread A
Loop forever
{
Read T_s;
Make the page table system call;
Read T_e;
}

1.3.3 Interrupt Response Time, Worst-Case IRT and Jitter

Interrupt response time refers to the time between the occurrence of an interrupt and the wakeup of the corresponding processing thread. The Worst-Case Interrupt Response Time (WCIRT) refers to the maximum value that the interrupt response time can reach under the most unfavorable conditions. The WCIRT is usually reached when a large number of cache misses and Trans Look-aside Buffer (TLB) misses occur during interrupt processing. Interrupt response time is the most important performance metric of the RTOS, and it can even be asserted that

everything else should be designed around it. This measurement is the most exhibited reflection of the system real-time responsiveness.

To obtain the WCIRT, a timer can be read in the first assembly instruction[1] of the interrupt vector, resulting in a time stamp T_s; read the timer at the first line of code of the interrupt processing thread to get a timestamp T_e. The measurement of the timer read cost is the same as the sections above. The resulting T_e-T_s-(T_{te}-T_{ts}) is the interrupt response time. The worst-case interrupt response time is the largest of all the values.

The jitter of these measurements is also very important. We often get a distribution when we measure the interrupt response time of the same system multiple times. The average of this distribution is the average interrupt response time, and its standard deviation[2] is called interrupt response time jitter.

For a RTOS, we usually wish the interrupt response time, the worst interrupt response time and jitter to be as small as possible. Interrupt response time measurement is usually conducted as follows[1][3]:

Table 1-12 Interrupt Response Time

Kernel	Thread A
Hardware interrupt handler	Loop forever
{	{
>> Read T_s;	Read T_e;
Send to endpoint P from kernel;	>> Receive from asynchronous endpoint P;
}	}

1.3.4 Realistic IRT, Realistic Worst-Case IRT and Jitter

The realistic interrupt response time refers to the time between the external stimulus's input and the corresponding IO operation's completion. The realistic WCIRT refers to the maximum length that an realistic interrupt response time can reach under the most unfavorable conditions. In addition to the factors that can affect the WCIRT, CPU and IO hardware's inherent overhead will also affect realistic WCIRT.

To get realistic IRT, we will need some extra hardware to conduct the measurement. For example, we need to measure the actual interrupt response time of an I/O line. We can connect the output pin of a FPGA to the input pin of a CPU or motherboard, then connect the input pin of

[1] We cannot wait until the C handler starts execution because register and stack maintenance are also part of the interrupt response time

[2] Sometimes we also use range

the same FPGA to the CPU or motherboard's output pin. Firstly, the FPGA sends a signal on its output pin. At this time, the high-resolution timer in the FPGA starts counting. When the FPGA receives the signal on its input pin, the high-resolution timer in the FPGA stops counting. The resulting internal FPGA timer value is the realistic IRT. The realistic WCIRT is the one with the highest response time among all tests.

For a cyber-physical system, we wish that the realistic IRT, the realistic WCIRT and the jitter are as small as possible. It is noteworthy that the realistic WCIRT will generally be approximately equal to the WCET plus the WCIRT plus the hardware's inherent overhead. For example, when a system is just starting to execute a system call at the time of stimulus, the hardware interrupt vector cannot be executed immediately, and the system call must be completed before we can respond to it. After the system call is completed, the hardware interrupt vector begins to execute, after this we will switch to the processing thread and produce the output. The realistic IRT is usually measured as follows[1]:

Table 1-13 Realistic Interrupt Response Time

FPGA (or oscilloscope)	System under test
Loop forever	Loop forever
{	{
>> Send stimulus and start the timer;	Receive signal from I/O;
Wait until receipt of the response;	Minimal processing routine;
Stop the timer;	Send response to I/O;
}	}

1.3.5 Input/Output Performance

I/O performance measurements are necessary for those operating systems who provide dedicated I/O subsystems, especially the systems that support virtualization. Common I/O subsystems include disk control systems, networking systems, parallel/serial port systems and data acquisition card systems. On microcontrollers, such systems include GPIO system, PWM generator system and LCD controller system, etc. Different evaluation standards apply to different subsystems. Usually, two standards are prevalent: bandwidth and latency. Bandwidth refers to the average data rate of the I/O system, while latency refers to the time between the assertion of the command and the arrival of the data.

1.3.6 Virtualization Performance

For those systems that support virtualization[1] of other operating systems[2], virtualization performance is also an important aspect. Virtualization performance usually include two parts: the functionality completeness and performance. Inter-VM communication can also be a very important aspect.

In terms of virtualization functionality completeness, we will evaluate whether all the functionality of the guest operating systems is correctly implemented and supported. The more functionality implemented and supported, the better.

In terms of virtualization performance, the measurements needed is similar to those listed in section 1.3.2. We also need additional measurements on the performance and storage overhead of virtualization: the less overhead, the better.

Inter-VM communication can be important in some cases as well. Generally speaking, inter-VM communication is more expensive than intra-VM communications, and they usually require dedicated drivers or virtual networks. The method to measure this performance is similar to measuring I/O performance, and is thus not detailed here.

1.4 RME System Call Interface

Making system call is the only method to invoke the functionality of the system. For RME, this is the only way to use its functionality. System calls are generally implemented by using software interrupts[3]. They can alternatively be implemented with dedicated instructions[4]. In the software interrupt implementation, when the software interrupt is triggered, the system will jump to the software interrupt vector to continue interrupt processing. The parameters of the system calls will be passed to the kernel[5], then the kernel will respond to the request. In the dedicated instruction implementation, the kernel will switch to the kernel stack and jump to the entry of the system call stub directly to handle the system call. Parameter passing is the same as in the software interrupt case.

RME supports both parameter passing methods. In x86-64 we support the former, while on ARM we support the latter.

1.4.1 System Calling Convention

[1] Both full virtualization and para-virtualization

[2] Such as RME, NOVA[5] and some L4 variants

[3] Such as ARM's SWI and SVC

[4] Such as x86-64's SYSCALL/SYSRET pair

[5] Usually shared memory or registers

To make a RME system call, first place the 4 arguments into 4 registers, then use the software interrupt instruction or dedicated instruction. RME always use 4 registers to pass arguments on any architecture. This is due to the fact that common C calling conventions allow up to 4 registers without using stack to pass them[1].

None of RME's system calls use more than 4 registers; RME does not pass arguments in shared memory[2], either. Passing arguments by registers may cause a kernel dereference of a user pointer, which will cause kernel-level segmentation fault. This is difficult to handle correctly, and may cause kernel panic or privilege escalation.

1.4.2 Parameter Passing and Position Encoding

RME passes its system call arguments by registers. However, some registers are too long for a single argument, and this makes passing only one argument in one register very luxurious as we can pass more arguments in this case. Hence we cut the registers into multiple bitfields; in RME, a single register will be cut into up to 8 bitfields. The designator and definition of each bitfield is as follows[3]:

Table 1-14 System Call Parameter Encoding

[31			32-bit machine word				0]
D1				D0			
Q3		Q2		Q1		Q0	
O7	O6	O5	O4	O3	O2	O1	O0

There are also some RME system calls that use special argument passing methods. These methods will be illustrated at the system calls' corresponding sections.

1.4.2.1 System Call Number

The system call number denotes the system functionality we are calling. This number is always located at the D1 field of the first register (P0) , and we also denote this field specially as N. RME have 35 system calls in total, whose system call number spans from 0-34 as listed below:

Table 1-15 List of RME System Calls

[1] I.e. MIPS and ARM architecture passes first 4 word-size arguments by registers and the arguments that follow by stack.

[2] Linux and some early variants of L4 will

[3] The example is based on 32-bit machines; the same goes for 64-bit

System call name	ID	Explanation
RME_SVC_INV_RET	0	Return from an invocation port
RME_SVC_INV_ACT	1	Activate an invocation port
RME_SVC_SIG_SND	2	Send to a signal endpoint
RME_SVC_SIG_RCV	3	Receive from a signal endpoint
RME_SVC_K_ERN	4	Call a kernel function
RME_SVC_THD_SCHED_PRIO	5	Changing thread priority
RME_SVC_THD_SCHED_FREE	6	Free a thread from a CPU core
RME_SVC_THD_TIME_XFER	7	Transfer time to a thread
RME_SVC_THD_SWT	8	Switch to another thread
RME_SVC_CAPTBL_CRT	9	Create a capability table
RME_SVC_CAPTBL_DEL	10	Delete a capability table
RME_SVC_CAPTBL_FRZ	11	Freeze a capability
RME_SVC_CAPTBL_ADD	12	Delegate a capability
RME_SVC_CAPTBL_REM	13	Remove a capability
RME_SVC_PGTBL_CRT	14	Create a page table
RME_SVC_PGTBL_DEL	15	Delete a page table
RME_SVC_PGTBL_ADD	16	Add a page to a page table
RME_SVC_PGTBL_REM	17	Remove a page from a page table
RME_SVC_PGTBL_CON	18	Construct a page table into another
RME_SVC_PGTBL_DES	19	Destruct a page table from its parent
RME_SVC_PROC_CRT	20	Create a process
RME_SVC_PROC_DEL	21	Delete a process
RME_SVC_PROC_CPT	22	Change a process's capability table
RME_SVC_PROC_PGT	23	Change a process's page table
RME_SVC_THD_CRT	24	Create a thread
RME_SVC_THD_DEL	25	Delete a thread
RME_SVC_THD_EXEC_SET	26	Set execution attributes of a thread[1]
RME_SVC_THD_HYP_SET	27	Set hypervisor attributes of a thread[2]

[1] Entry and stack

[2] Register saving location

System call name	ID	Explanation
RME_SVC_THD_SCHED_BIND	28	Bind a thread to the current processor
RME_SVC_THD_SCHED_RCV	29	Try to receive scheduling notifications
RME_SVC_SIG_CRT	30	Create a signal endpoint
RME_SVC_SIG_DEL	31	Delete a signal endpoint
RME_SVC_INV_CRT	32	Create a synchronous invocation port
RME_SVC_INV_DEL	33	Delete a synchronous invocation port
RME_SVC_INV_SET	34	Set execution attributes of an invocation port[1]

1.4.2.2 Capability Table Number

The capability table number denoted the capability table to operate on. This number is always located at the D0 field of the first register (P0), and we denote this field specially as C. Only some system calls will require a capability to a capability table, and this parameter is useful if and only if these system calls are used.

1.4.2.3 Other Parameters

The first parameter is passed in the second register, the second parameter is passed in the third register, and the third parameter is passed in the fourth register. We denote these parameters as P1, P2 and P3. In this manual, P1.D1 denotes the D1 field of P1, and so on.

1.4.3 Special Notes

1. When creating kernel objects, the kernel virtual address must be aligned to the RME_KMEM_SLOT_ORDER order of 2.

2. System call with number 0-8 will cause a potential context switch. For branch prediction optimization, their system call numbers are contiguous.

3. All system calls will return a non-negative value upon a success, and a negative value upon a failure.

4. In this manual, all typedef'ed types' prefix "rme_" is omitted to save table space.

1.5 Manual Overview

In the chapters that follow, we will introduce different subsystems of RME one by one. Chapter 2 describes capability tables and capability management, Chapter 3 describes page tables and memory management, Chapter 4 describes process and thread management, Chapter

[1] Entry and stack

5 describes synchronous communication mechanisms and asynchronous communication mechanisms, Chapter 6 describes kernel function call mechanisms. Then, Chapter 7 details the porting of RME, and Chapter 8 describes the implementation standards of RME kernel function calls. At last, Chapter 9 briefs some known-issues in RME and corresponding workarounds.

In this Technical User Manual (TRM), only the kernel of RME will be described. For details of RME on different architectures, please refer to the corresponding manual dedicated to the architecture; for information on RME user-level library[1], please refer to the TRM of itself.

This manual assumes that the user have some familiarity with the basic concepts of operating systems. The abstruse concepts which are related to microkernel and RME will be introduced at the beginning of each chapter.

1.6 RME Architecture Overview

Shown below is a brief overview of the system architecture. Included in the graph are the RME kernel, its user-level libraries, and various components within the library. In the figure below, The black dotted rectangles represent logically independent software modules, the black solid rectangles represent hardware protected software modules, and the grey solid rectangles represent boundaries of software packages. The rectangles with ellipsis (.....) represents the subsystems that are omitted because is the same with what is on its left side. Due to space limitations, only the important components are shown in the system, and some components are not displayed here; for detailed user-level architecture figures, please refer to the TRM of the user-level library of RME.

[1] That is, RVM (M7M2)

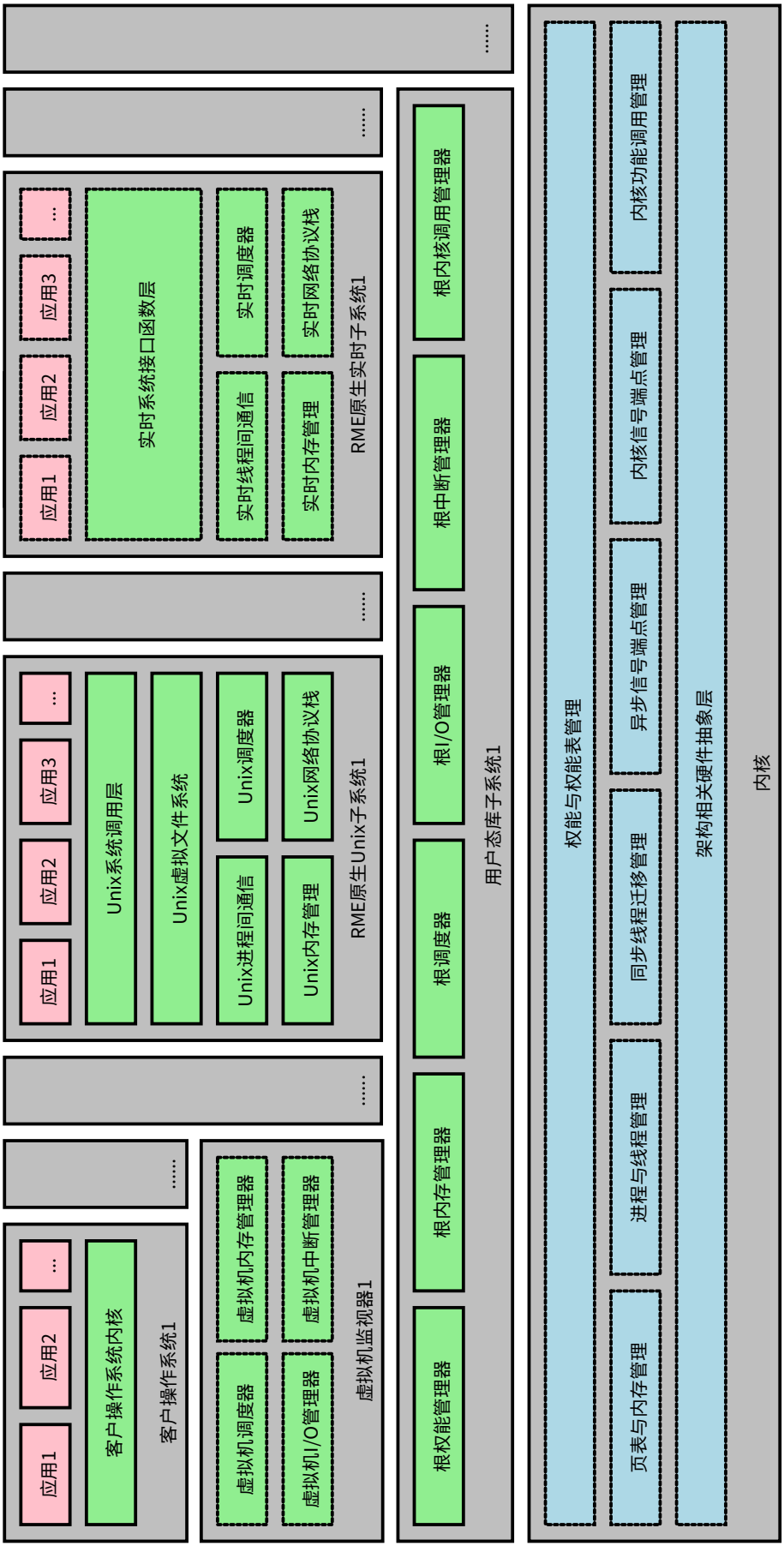

Figure 1-3 RME Architecture Overview

1.7 Bibliography

[1] T. N. B. Anh and S.-L. Tan, "Real-time operating systems for small microcontrollers," IEEE micro, vol. 29, 2009.

[2] R. P. Kar, "Implementing the Rhealstone real-time benchmark," Dr. Dobb's Journal, vol. 15, pp. 46-55, 1990.

[3] T. J. Boger, Rhealstone benchmarking of FreeRTOS and the Xilinx Zynq extensible processing platform: Temple University, 2013.

[4] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, et al., "seL4: formal verification of an OS kernel," presented at the Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, Big Sky, Montana, USA, 2009.

[5] U. Steinberg and B. Kauer, "NOVA: a microhypervisor-based secure virtualization architecture," in Proceedings of the 5th European conference on Computer systems, 2010, pp. 209-222.

Chapter 2 Capability Table and Capability Management

2.1 The Concept of Capability

Capabilities are a kind of access permission control mechanism originally introduced in multi-user computer systems[1]. They can be interpreted as unforgeable tokens that uniquely point to some resources and carry the operations allowed on these resources. Unix file descriptors are capabilities to some extent[2]; The access permissions of Windows are capabilities. In other words, capabilities are fat pointers to a certain system resource.

We guarantee the systems' security with the three following rules[2]:

1. Capabilities cannot be forged or tempered with at user level;
2. Processes can only be granted capabilities through well-defined interfaces;
3. Capabilities will only be given to the resource managers decided at system design-time.

3rd-generation microkernels apply capability-based access control over all of their kernel resources. In RME, all kernel resources are managed with capabilities, which are all located in the kernel address space. Every process has its own capability table, and when a system call is performed, the system will lookup the corresponding resource from the capability table. Every capability have a number of bits indicating the operations allowed on this capabilities. If the corresponding bit is set, then the operation is allowed[3]. Most microkernels, like seL4, etc.[5], have these bits as well. some other systems[1] don't because there is only one operation on most kernel resources, thus a bitfield is not needed[4].

There are also other alternatives to capabilities, e.g. Access Control List (ACL)[6]. It can also be applied to manage access permissions. The drawback of this mechanism includes coarser grain and bigger list size. Its advantage is the easiness of granting and revoking permissions[2].

In RME, by leveraging the concept of capability, it is easy to implement Discretionary Access Control (DAC) and Mandatory Access Control (MAC), and multi-level security mechanisms. In user-level application designs, the concept of minimal privilege should be applied.

2.2 Capability Table Operations and States of Capabilities

Capability table is a kernel object which stores capabilities. In RME these tables are linear arrays, and the size of every slot is 8 machine words. Each slot can hold a capability. The information regarding the type, kernel virtual memory address[3], its parent capability, reference

[1] E.g. Composite

[2] Especially revoking operations

[3] A pointer to kernel address space

count and state is stored in each capability slot. There is an additional timestamp counter to ensure the quiescence of capability operations to cope with multi-core environments.

2.2.1 Capability Types

In RME, there are 8 types of capabilities[1], as listed below. For detailed information about each capability type, please consult the corresponding chapter for details. This chapter will only detail capability table capabilities.

Table 2-1 Types of Capabilities

权能类型号	权能类型	用途
RME_CAP_NOP	空白权能	权能表的这个位置是空白权能。
RME_CAP_KERN	内核权能	调用特殊功能内核函数的必备权能。
RME_CAP_KMEM	内核内存权能	使用一段内核内存创建内核对象的必备权能。
RME_CAP_CAPTBL	权能表权能	指向一个权能表对象，可用来进行权能表管理。
RME_CAP_PGTBL	页表权能	指向一个页表对象，可用来进行内存管理。
RME_CAP_PROC	进程权能	指向一个进程对象，可用来进行进程管理。
RME_CAP_THD	线程权能	指向一个线程对象，可用来进行线程管理。
RME_CAP_INV	调用权能	指向一个迁移调用对象，可用来进行线程迁移调用。
RME_CAP_SIG	信号权能	指向一个信号端点对象，可进行信号的发射和接收。

Each capability represents the power to operate on the corresponding kernel object. It also represents the functionality that this kernel object can perform. It is worth noting that capabilities that correspond to capability tables[2] are meta-capabilities of the system, because they have the power to modify capability tables and thus decide the contents of capability tables.

2.2.2 Reference Count of Capabilities and Parent Capabilities

The reference counter and parent capability pointer of a capability is used to track capability delegation. A capability can be passed from one capability table to another through delegation, where case we call the source capability as parent and the destination capability as child. The child's parent pointer always points to its parent, and the parent's reference count will be incremented by 1. When removing capabilities, the child capability must be removed before the

[1] Without counting in the empty capability

[2] Capability table capabilities, for short

parent capability gets removed. When the initial capability to a kernel object is created, its parent pointer will be NULL, and its reference count will be set to 0.[4]

We call the initial capability to a kernel object as the root capability, and these capabilities that are created by delegation as non-root capabilities. The distinction of a root capability is that its parent pointer is NULL.

2.2.3 States of Capabilities

There are four states for a capability: empty, creating, valid and frozen. The state of a capability can change between the four by making system calls.

To create a capability, the corresponding kernel object creation system call must be performed. A capability table capability is needed to designate the target capability table[1], and other necessary information including the address of the kernel object must be passed in. The creation operation will be Compare-And-Swaped (CAS) to "creating"; when the creation is finished, the state will be marked as "valid". If an error happens during creation, the slot will be reset to "empty", and an error code is returned. When creating capabilities, if kernel memory allocations are needed, a kernel memory capability is necessary; see next chapter for details on kernel memory capabilities.

To delete or remove a capability, we need to freeze the slot first. The freezing operation is conducted by a special system call. Then, the slot will become "frozen". If freeze operation failed, a corresponding error code is returned. Once successful, after a quiescence period, the capability can be deleted or removed. The deletion operation will delete the kernel object together with the capability, while the removal operation will only remove the capability. After the capability gets deleted or removed, it will return to "empty" state. For non-root capabilities, only removal operations can be used; for root capabilities, only deletion operations can be used.

There are two design styles regarding capability revoking. One of them being the implementation of seL4 and Fiasco.OC, which supports system-level capability revoking. Upon kernel object deletion, the whole capability delegation tree will be traversed and all the capabilities that points to the same kernel object gets removed. RME uses the second implementation, which separates capability removal and deletion. The operating system is not responsible for tree iteration and revoking, and this work must be conducted at the user level. The benefit is that no kernel preemption points are needed; the drawback is that the user-level must track every capability delegation to gather the necessary information for revoking.[4]

[1] The creation operation adds a capability into the capability table, which is a modification to the table itself, thus requiring a capability table capability; for the same reason, when removing capabilities you also need such a capability.

2.2.4 Timestamp Counter and Multi-core Scalability

Multi-core execution paths are highly complex. There are some kernel data structures that shouldn't be modified simultaneously in a multi-core environment, and these data structures require that all operations on them are atomic. Therefore, two solutions exist: the first being taking a lock for each modification, the second being using atomic instructions for each modification. The first solution will incur extra overhead hence slower speed, and they also cause cache line contention. Cache line contention refers the phenomena that the CPU1 tries to modify the cache line, while the CPU2 tries to read the cache line, thus the cache of CPU2 gets invalidated frequently. This is equivalent to a lower memory bandwidth, and will hamper frequent capability operation performance. The atomic instruction solution has less cache issues but are more challenging to implement. In RME, we use the latter method to achieve multi-core scalability.

RME includes a large number of Compare-And-Swap (CAS) atomics and Fetch-And-Add (FAA) atomics. The kernel also employs a timestamp counter to ensure that no kernel operations conflict, i.e. one capability under use on one CPU gets deleted on another CPU.

To delete or remove a capability, we must ensure that the capability have been frozen for a time period. This period is called capability quiescence period. The length of this period can be configured, and must be longer than twice the kernel WCET. Thus, we can ensure that when we delete or remove the capability, the capability have been stabilized. In other words, all kernel operations using this capability have already finished on all CPU cores, and there would not be a conflict between usage and deletion or removal. For example, when CPU1 is trying to delete the capability A, CPU2 tries to use it simultaneously. CPU2 may have already completed the operation validation, and is progressing. If is capability gets deleted on CPU1 without freezing it, the operation of CPU2 may be affected. The timestamp counter of each capability is designed to log the freeze time; it will guarantee that no deletion or removal can occur before the capability is completely frozen.

A complete capability state transition diagram is shown below. When the creation starts, the CPU will perform a CAS operation in the empty slot, which marks the start of the creation and the occupation of the slot. After the creation is finished, the CPU will mark the slot as usable, and now we can start using this capability. If this capability is not referenced, then it can be deleted or removed after usage. A capability can only be frozen after a quiescence period has passed since its creation, and it can only be removed or deleted after a quiescence period has passed since the freezing operation.

When deleting or removing capabilities, we need to check all necessary conditions to confirm that the deletion or removal can be performed. After this, a CAS operation will return this

capability slot to an empty state, and we guarantee that only one CPU can proceed. Then, the parents' reference count will be decreased by one (removal) or the kernel object will be deleted (deletion). In RME, new capabilities can be created on the slot immediately after deletion or removal. This is due to the fact that RME cache all the information before we change slot state to empty, and all the operations that follow will use the cached information instead of accessing the capability slot. In Composite[4], it may still use these information in the slot, thus another quiescence period needs to be inserted here to avoid being overwritten by newly created capabilities.

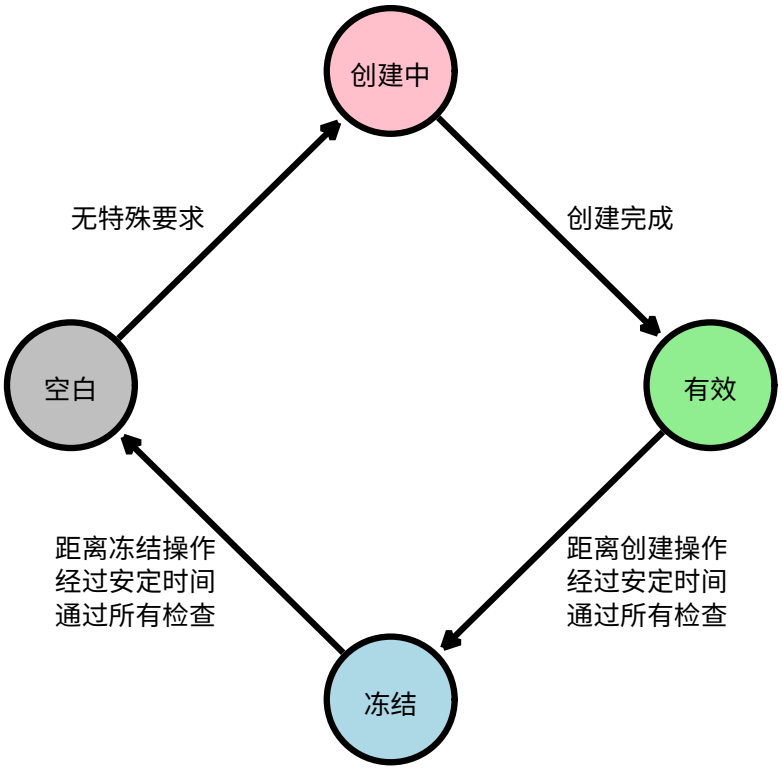

Figure 2-1 Life Cycle of Capabilities

The reason why the quiescence period must be at least twice the kernel WCET is, when we freeze the capability, we will update its timestamp first, then we will mark it as frozen with atomic instructions. Thus we have the following timeline:

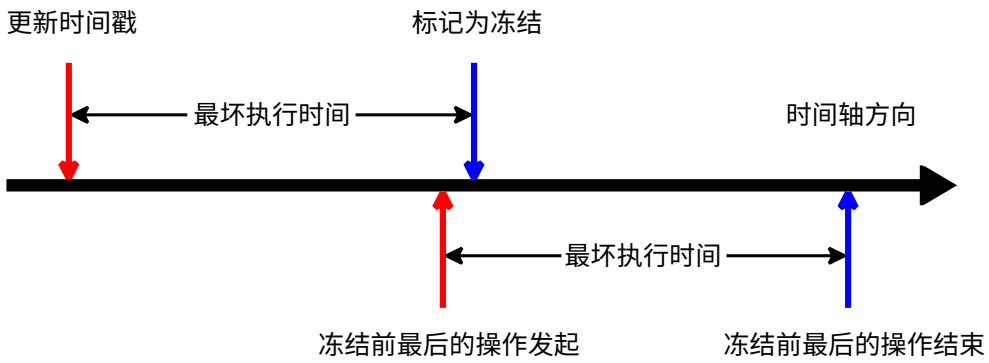

Figure 2-2 Timeline of Capability Freezing

We can see from the timeline that two WCETs can occur from the update of the timestamp to the end of the last operation. In real-world applications, we recommend that the quiescence period be configured to at least 10 times[1] the estimated WCET, because usually only the order estimations of WCET are correct.

The same principle also applies to the creation-freezing quiescence period of capabilities, and is not explained at length here.

2.2.5 Capability Table Structure

The capability tables can be organized as a multi-level structure. When we place capability table capabilities into capability tables, the capability tables are organized as a radix trie. A capability number which designates the in-table position of the capability needs to be passed in. The capability number can encode two levels of lookup at most. We call the first level "master capability table", and the second level "expanded capability table". In 32-bit systems, capability number is a 16-bit value; in 64-bit systems, capability number is a 32-bit value, and this goes for systems with higher number of bits as well. The detailed encoding is shown hereinafter:

Table 2-2 Encoding of Capability Numbers

系统位数	编码类型	编码方法
32 位系统	一级查找编码	[15:8]保留 [7]固定为 0 [6:0]位置
	二级查找编码	[15]保留 [14:8]子表位置 [7]固定为 1 [6:0]在子表中的位置
64 位系统	一级查找编码	[32:16]保留 [15]固定为 0 [14:0]位置
	二级查找编码	[32]保留 [31:16]子表位置 [15]固定为 1 [14:0]在子表中的位置

From the table we can see that there can be at most 2^7=128 capabilities in a capability table under 32-bit, and 2^{15}=32768 capabilities under 64-bit. This value is represented by macro

[1] An order of magnitude is recommended

RME_CAPID_2L in the system. Different from systems like seL4, RME does not support radix-trie based look-ups that are more than two levels. In other words, the capabilities with in the capability table[1] that is pointed to by the capability table capability in the expanded capability table are not considered to be within the expanded capability table, and cannot be used directly with second-level encoding[2]. If you need to use them, you must delegate them to the expanded capability table first, then they can be accessed through second-level encoding.

In the figure below, the main capability table contains a capability table capability that points to a expanded capability table, and it resides in position 4. The capabilities in this extended capability table can be accessed by using second-level encoding. In the position 2 of the expanded capability table, there is a capability that points to a secondary expanded capability table. The capability itself can be accessed by using the second-level encoding 4:2, but the capabilities within the secondary expanded capability table which it points to cannot be accessed with second-level encoding. The portion that cannot be accessed by second-level encoding is illustrated with grey color.

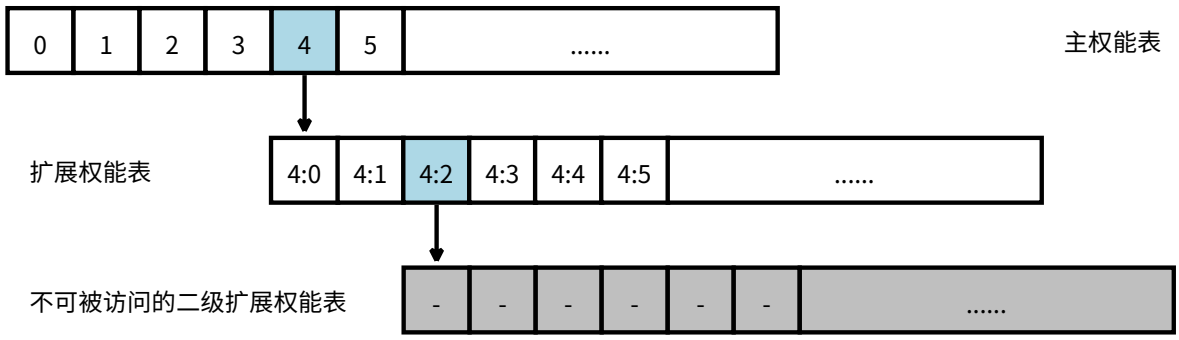

Figure 2-3 Capability Table Structure

2.3 Capability Table System Calls

The system calls that are related to the capability tables are shown hereinafter:

Table 2-3 Capability Table Related System Calls

调用号	类型	用途
RME_SVC_CAPTBL_CRT	系统调用	创建权能表
RME_SVC_CAPTBL_DEL	系统调用	删除权能表
RME_SVC_CAPTBL_ADD	系统调用	权能传递
RME_SVC_CAPTBL_FRZ	系统调用	权能冻结

[1] That is, the secondary expanded capability table

[2] That is, “my vassal's vassal is not my vassal”

调用号	类型	用途
RME_SVC_CAPTBL_REM	系统调用	权能移除

权能表权能的操作标志如下：

Table 2-4 Capability Table Operation Flags

标志	位	用途
RME_CAPTBL_FLAG_CRT	[0]	允许在该权能表中创建权能。
RME_CAPTBL_FLAG_DEL	[1]	允许删除该权能表中的权能。
RME_CAPTBL_FLAG_FRZ	[2]	允许冻结该权能表中的权能。
RME_CAPTBL_FLAG_ADD_SRC	[3]	允许该权能表在权能传递作为来源表。
RME_CAPTBL_FLAG_ADD_DST	[4]	允许该权能表在权能传递作为目标表。
RME_CAPTBL_FLAG_REM	[5]	允许移除该权能表中的权能。
RME_CAPTBL_FLAG_PROC_CRT	[6]	允许在创建进程时将该权能表作为进程的权能表。
RME_CAPTBL_FLAG_PROC_CPT	[7]	允许用该权能表替换某进程的权能表。

For detailed information about bit[6] and bit[7], please refer to 4.4.1 and 4.4.3.

2.3.1 Capability Table Creation

该操作会创建一个权能表，并将其权能放入某个已存在的权能表。创建权能表操作需要如下几个参数：

Table 2-5 Parameters Needed for Capability Table Creation

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_CAPTBL_CRT。
Cap_Captbl_Crt	cid_t	C	一个对应于必须拥有 RME_CAPTBL_FLAG_CRT 属性的权能表权能的权能号，该权能号对应的权能指向要接受此新创建的权能表权能的权能表。该权能号可以是一级或二级查找编码。
Cap_Kmem	cid_t	P1.D1	一个内核内存权能号，其标识的内核内存范围必须能够放下整个权能表，并且要拥有 RME_KMEM_FLAG_CAPTBL 属性。该权能号可以是一级或二级查找编码。
Cap_Crt	cid_t	P1.D0	一个对应于接受该新创建的权能表权能的权能表的某位置的权能号。该权能号对应的权能必须是空白的。该权能号只能是一级查找编码。

Vaddr	ptr_t	P2	新创建的权能表要使用的内核空间起始虚拟地址。
Entry_Num	ptr_t	P3	该权能表包含的表项数目，必须在 1 到 RME_CAPID_2L 之间。如果 RME_CAPTBL_LIMIT[1]被定义为一个小于 RME_CAPID_2L 且不为 0 的值，则该数目必须在 1 到 RME_CAPTBL_LIMIT 之间。

该操作的返回值可能如下：

Table 2-6 Possible Return Values for Capability Table Creation

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	传入的权能表权能数目参数超出了操作系统允许的范围。
	Cap_Captbl_Crt 的一级/二级查找超出了范围。
	Cap_Kmem 的一级/二级查找超出了范围。
	Cap_Crt 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl_Crt 的一级/二级查找的权能已经被冻结。
	Cap_Kmem 的一级/二级查找的权能已经被冻结。
	Cap_Crt 被冻结，或者其它核正在该处创建权能。
RME_ERR_CAP_TYPE	Cap_Captbl_Crt 不是权能表权能。
	Cap_Kmem 不是内核内存权能。
RME_ERR_CAP_FLAG	Cap_Captbl_Crt 无 RME_CAPTBL_FLAG_CRT 属性。
	Cap_Kmem 无 RME_KMEM_FLAG_CAPTBL 属性，或范围错误。
RME_ERR_CAP_EXIST	Cap_Crt 不是空白权能。
RME_ERR_CAP_KOTBL	分配内核内存失败。

2.3.2 Capability Table Deletion

该操作会删除一个权能表。被删除的权能表必须不含有权能，也即其全部权能位置应该都是空白的。删除权能表需要以下几个参数：

Table 2-7 Parameters Needed for Capability Table Deletion

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_CAPTBL_DEL。
Cap_Captbl_Del	cid_t	C	一个对应于必须拥有 RME_CAPTBL_FLAG_DEL 属性的权能表权能的权能号，该权能号对应的权能指向含有正被删除的权能表权能的

[1] 见 7.3.2

参数名称	类型	域	描述
			权能表。该权能号可以是一级或者二级查找编码。
Cap_Del	cid_t	P1	一个对应于将被删除的权能表权能的权能号。该权能号对应的权能必须是一个权能表权能。该权能号只能是一级查找编码。

该操作的返回值可能如下：

Table 2-8 Possible Return Values for Capability Table Deletion

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Captbl_Del 的一级/二级查找超出了范围。
	Cap_Del 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl_Del 的一级/二级查找的权能已经被冻结。
	Cap_Del 未被冻结。
RME_ERR_CAP_TYPE	Cap_Captbl_Del 不是权能表权能。
	Cap_Del 不是权能表权能。
RME_ERR_CAP_NULL	Cap_Del 为空白权能。
	两个核同时试图删除该权能表，此时未成功的核返回该值。
RME_ERR_CAP_FLAG	Cap_Captbl_Del 无 RME_CAPTBL_FLAG_DEL 属性。
RME_ERR_CAP_QUIE	Cap_Del 不安定。
RME_ERR_CAP_EXIST	Cap_Del 对应的权能表内还有权能。
RME_ERR_CAP_REFCNT	Cap_Del 的引用计数不为 0，或者不为根权能。

2.3.3 Capability Delegation

该操作会将一个权能表中的某个权能传递到另外一个权能表的空白位置中。新创建的目标权能的父权能是源权能，并且源权能的引用计数会增加 1。在权能表之间进行权能传递需要以下几个参数：

Table 2-9 Parameters Needed for Capability Delegation

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_CAPTBL_ADD。
Cap_Captbl_Dst	cid_t	P1.D1	一个对应于必须拥有 RME_CAPTBL_FLAG_ADD_DST 属性的权能表权能的权能号，该权能号对应的权能指向目标权能表。该权能号可以是一级或者二级查找编码。
Cap_Dst	cid_t	P1.D0	一个对应于将接受被传递的权能的权能号。该权能号对应的权能

参数名称	类型	域	描述
			必须是空白的。该权能号只能是一级查找编码。
Cap_Captbl_Src	cid_t	P2.D1	一个对应于必须拥有 RME_CAPTBL_FLAG_ADD_SRC 属性的权能表权能的权能号，该权能号对应的权能指向源权能表。该权能号可以是一级或者二级查找编码。
Cap_Src	cid_t	P2.D0	一个对应于将传递的权能的权能号。该权能号对应的权能必须不为空白而且没有冻结。该权能号只能是一级查找编码。
Flags	ptr_t	P3	要传递的操作标志属性。只有这个操作标志允许的操作才能被新创建的权能执行。

需要注意的是，对于内核内存权能，其传递时还需要置于系统调用号 N 和权能表权能号 C 中的额外位来辅助确定其操作标志属性。具体的参数传递方法请参见 3.4。

该操作的返回值可能如下：

Table 2-10 Possible Return Values for Capability Delegation

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Captbl_Dst 或 Cap_Captbl_Src 的一级/二级查找超出了范围。
	Cap_Dst 或 Cap_Src 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl_Dst 或 Cap_Captbl_Src 的一级/二级查找的权能被冻结。
	Cap_Src 被冻结。
RME_ERR_CAP_TYPE	Cap_Captbl_Dst 或 Cap_Captbl_Src 不是权能表权能。
RME_ERR_CAP_NULL	Cap_Src 为空白权能。
RME_ERR_CAP_FLAG	Cap_Captbl_Src 无 RME_CAPTBL_FLAG_ADD_SRC 属性。
	Cap_Captbl_Dst 无 RME_CAPTBL_FLAG_ADD_DST 属性。
	Cap_Src 的操作标志属性与传入的操作标志属性冲突，也即传入的属性包括了 Cap_Src 不允许的操作或者操作范围。
	传入的操作标志属性是不合法的，比如操作范围上下限冲突，或者不允许在传递产生的权能上做任何操作。
RME_ERR_CAP_EXIST	Cap_Dst 不是空白权能。
RME_ERR_CAP_REFCNT	Cap_Src 的引用计数超过了系统允许的最大范围。在 32 位系统中上限是 2^{23}-1，在 64 位系统中上限是 2^{46}-1。通常这是足够的。

2.3.4 Capability Freezing

该操作会将一个权能表中的某个权能冻结。如果一个权能被冻结，那么在安定时间之后，能够保证从这个权能发起的，对这个权能指向的内核对象的操作在内核中全部停止，此时可以删除或移除该权能。注意，这并不等价于该权能指向的内核对象的全部操作都停止，因为还可能有其他权能指向这个内核对象，而从这些权能发起的内核对象操作仍然可以进行。如果根权能被冻结，那么才能保证该内核对象上的所有操作都停止，此时才可以删除该权能和内核对象。冻结一个权能需要如下参数：

Table 2-11 Parameters Needed for Capability Freezing

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_CAPTBL_FRZ。
Cap_Captbl_Frz	cid_t	C	一个对应于必须拥有 RME_CAPTBL_FLAG_FRZ 属性的权能表权能的权能号，该权能号对应的权能指向含有正被冻结的权能的权能表。该权能号可以是一级或者二级查找编码。
Cap_Frz	cid_t	P1	一个对应于将被冻结的权能的权能号。该权能号只能是一级查找编码。

该操作的返回值可能如下：

Table 2-12 Possible Return Values for Capability Freezing

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Captbl_Frz 的一级/二级查找超出了范围。
	Cap_Frz 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl_Frz 的一级/二级查找的权能已经被冻结。
	Cap_Frz 已经被冻结，无需再次冻结，或者正在被创建。
RME_ERR_CAP_TYPE	Cap_Captbl_Frz 不是权能表权能。
RME_ERR_CAP_NULL	Cap_Frz 为空白权能。
RME_ERR_CAP_FLAG	Cap_Captbl_Frz 无 RME_CAPTBL_FLAG_FRZ 属性。
RME_ERR_CAP_QUIE	Cap_Frz 不安定。
RME_ERR_CAP_EXIST	两个核同时试图冻结该权能，此时未成功的核返回该值。
RME_ERR_CAP_REFCNT	Cap_Frz 的引用计数不为 0。

2.3.5 Capability Removal

该操作会将一个权能表中的某个权能移除。被移除的权能必须不是根权能[1]，而且必须不被引用。移除一个权能不会导致与之相关联的内核对象被移除，被移除的仅仅是权能本身[2]。移除一个权能需要如下参数：

Table 2-13 Parameters Needed for Capability Removal

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_CAPTBL_REM。
Cap_Captbl_Rem	cid_t	C	一个对应于必须拥有 RME_CAPTBL_FLAG_REM 属性的权能表权能的权能号，该权能号对应的权能指向含有正被移除的权能的权能表。该权能号可以是一级或者二级查找编码。
Cap_Rem	cid_t	P1	一个对应于将被移除的权能的权能号。该权能号只能是一级查找编码。

该操作的返回值可能如下：

Table 2-14 Possible Return Values for Capability Removal

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Captbl_Rem 的一级/二级查找超出了范围。
	Cap_Rem 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl_Rem 的一级/二级查找的权能已经被冻结。
	Cap_Rem 未被冻结。
RME_ERR_CAP_TYPE	Cap_Captbl_Rem 不是权能表权能。
RME_ERR_CAP_NULL	Cap_Rem 为空白权能。
	两个核同时试图移除该权能，此时未成功的核返回该值。
RME_ERR_CAP_FLAG	Cap_Captbl_Rem 无 RME_CAPTBL_FLAG_REM 属性。
RME_ERR_CAP_QUIE	Cap_Rem 不安定。
RME_ERR_CAP_REFCNT	Cap_Rem 的引用计数不为 0，或者为根权能。

2.4 Bibliography

[1] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," Communications of the ACM, vol. 9, pp. 143-155, 1966.

[1] 对根权能应当使用删除操作

[2] 删除操作则只能对不被引用的根权能使用，并且会同时删除根权能和内核对象

[2] J. S. Shapiro, J. M. Smith, and D. J. Farber, EROS: a fast capability system vol. 33: ACM, 1999.

[3] R. J. Feiertag and P. G. Neumann, "The foundations of a provably secure operating system (PSOS)," in Proceedings of the National Computer Conference, 1979, pp. 329-334.

[4] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A kernel for scalable predictability," in Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE, 2015, pp. 121-132.

[5] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, et al., "seL4: formal verification of an OS kernel," presented at the Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, Big Sky, Montana, USA, 2009.

[6] B. W. Lampson, "Protection," ACM SIGOPS Operating Systems Review, vol. 8, pp. 18-24, 1974.

Chapter 3 Page Table and Memory Management

3.1 Introduction to Memory Management

内存管理（Memory Management）指的是对物理内存（Physical Memory）及虚拟地址空间（Virtual Address Space，VAS）进行的分配和回收操作。要保证内存管理的安全性，硬件上的支持是必要的。RME 操作系统的内存管理支持内存保护单元（Memory Protection Unit，MPU）环境和内存管理单元（Memory Management Unit，MMU）环境，并且它们被抽象成了相同的页表数据结构。本章中所用到的术语的定义如下：

Table 3-1 Page Directory Glossary

术语	定义
总页表（Page Table，PT）	指的是整个页表本身，包括了顶层页目录和中间的各个级别的页目录，是一棵地址树的总称。
页目录（Page Directory，PD）	指的是页表中的一级，其中最顶层的一级被称为顶层页目录。
页表项（Page Entry，PE）	指的是页目录表格中的一项，它可能指向一个页，也可能指向下一级页目录。其中，我们把指向页的叫做末端页表项（Page Terminal Entry，PTE），把指向下一级页目录的叫做中间页表项（Page Intermediate Entry，PIE）。

3.2 Page Table Operations and Structure

页表是由一系列页目录组成的多层基数树（Radix Trie/Radix Tree）结构。页目录的每一个槽位都被固定为一个机器字的长度。页目录中的每一个槽位都可以放置以下三种数据之一：下级页目录物理地址、页表项物理地址及属性或空页表项。如果存放的是下级页目录物理地址，那么代表此处有一个下级页目录，该部分虚拟地址的映射关系要查询该页目录决定；如果存放的是页表项物理地址及属性，则代表此虚拟地址处有一个页表项被映射，并且可以得知该页的访问属性；如果存放的是空页表项，那么则说明这个虚拟地址处没有任何东西被映射。在 RME 中，页表是需要用户手动构造的，这和 Composite 等一系列微内核的解决方法是一致的[1]。

3.2.1 Kernel Memory and User Memory

RME 的系统内存被分为两部分：一部分是内核内存，一部分是用户内存。和 Composite、L4 等微内核不同，在 RME 系统中，内核内存映射是在一开始就完全建立[1]的，并且不可修改。这使得 RME 完全不需要内存动态类型机制[2]，而且内存管理系统完全可并行化，同时彻底免去了内核内存内容泄露的可能。对于 MMU 环境，在创建顶层页目录时，系统会将在启动时就创建好的内核页目录映射到顶层页

[1] 静态或半静态分配

[2] 这种机制被 Composite、seL4 等系统采用[1][3]。在 RME 中它其实也可以被受信任的用户态服务器实现

目录之内；对于 MPU 环境，由于内核态通常都有对整个内存的访问权限，因此在所有页目录中我们只需要用户页就可以了[1]。

在系统启动时，所有的用户物理内存页都被加入了启动进程（Init）的页表之中。在这种添加结束之后，通常而言不再允许系统凭空地创造物理内存页框，除非使用自定义内核调用[2]。在创建新进程时，新进程的页表的页表项是必须从其他进程处添加过来的。在添加时，可以指定这个页的访问属性，并且所指定的访问属性一定要是父页面的访问属性的一个子集。RME 系统中，页访问的标准属性如下表：

Table 3-2 Standard Access Permissions for Page Accesses

名称	标识符	意义
可读	RME_PGTBL_READ	这个页面是可读取的。
可写	RME_PGTBL_WRITE	这个页面是可写入的。
可执行	RME_PGTBL_EXECUTE	这个页面的是可作为代码执行的。
可缓存	RME_PGTBL_CACHEABLE	这个页面的内容可以被缓存。
可缓冲	RME_PGTBL_BUFFERABLE	这个页面的写入可以被缓冲。
静态	RME_PGTBL_STATIC	这个页面是总被映射的静态页。这意味着，MPU 环境下或在手动更新 TLB 的 MMU 环境下，这个页总是被映射，而非等到缺页中断来临时映射。

值得注意的是，在某些架构中，上面的某些位可能不会全部都具有意义。比如，对于绝大多数自动更新快表（Trans Look-aside Buffer，TLB）的 MMU 环境，静态属性是没有意义的；对于某些架构，读和写是一起实现的，因此不具有分立的读写控制。

为了实现用户态对在创建内核对象时对内核内存的管理，系统中的内核对象使用一个内核对象登记表进行管理。内核对象登记表是一个位图，里面存储了内核对象对于虚拟地址的占用。这个位图保证了在同一段内核虚拟地址上，不可能同时存在两个内核对象。

为了防止某些有权创建内核对象的系统组件在出错或被入侵时大量创建内核对象从而耗尽内核内存，发动拒绝服务（Denial of Service，DoS）攻击，因此引入了内核内存权能来管理内核内存。内核内存权能的概念参考了 Fiasco.OC 的内核对象工厂（Factory）[4]。在创建任何一个内核对象时，都需要内核内存权能；该内核内存权能标志了允许用来创建内核对象的内核虚拟内存地址范围，以及允许在这段内存上创建哪些对象。只有当被创建的内核对象完全落在这个范围之内，并且该内核内存权能的标志位允许创建该种对象时，创建操作才能够被继续进行，否则将返回一个错误，从而限制某些被感染的恶意内核组件发起拒绝服务攻击。

[1] 某些架构（如 Tricore 等）需要保留一个或者多个 MPU 寄存器组给内核态，但是这些页并不会反映在页表中

[2] 见 8.1

3.2.2 Page Table Properties

在 RME 操作系统中，页目录有四个属性。这四个属性唯一决定了页目录的状态。我们下面将分别介绍这四种属性。

3.2.2.1 Mapping Start Address

映射起始地址指该层页目录开始映射的虚拟地址。这个页目录的第一个槽位的物理内存或者映射的二级页表的起始地址，就是这个虚拟地址。当我们试图把一个更底层的页目录映射到某高层次页目录的某位置时，我们可能需要检查底层的虚拟地址是否和高层的虚拟地址匹配，从而决定能否进行该映射。当然，这个检查仅仅在使用 MPU 的系统中是必须的。在使用 MMU 的系统中，由于一个页目录经常会被映射进不同的页目录的不同位置，这个检查可以被配置为不进行，此时映射起始地址一项无效，此时也无法使用路径压缩页表（Path-Compressed Page Table，PCPT）格式[1]。有关路径压缩页表格式、MPU 系统和 MMU 系统的差别请见后续章节。

3.2.2.2 Top-level Page Directory Flag

标志着该页目录为最顶层的页目录。只有最顶层的页目录才可以被用来创建进程。

3.2.2.3 Page Directory Size Order

页目录大小级数决定了页目录每个槽位代表的虚拟地址的大小。如果某页目录的大小级数为 12，那么就意味着该页目录中的每个槽位都对应 2^{12}=4096 字节大小的一个页。

3.2.2.4 Page Directory Number Order

页目录数量级数决定了页目录中的槽位数量。如果某页目录的数量级数为 10，那么就意味着该页目录中一共有 2^{10}=1024 个槽位。

3.2.3 Basic Page Directory Operations

在页目录上一共有六种基本操作，分别如下：

Table 3-3 Basic Operations of Page Directories

操作	含义
创建页目录（Create）	创建一个新的空页目录。
删除页目录（Delete）	删除一个页目录。
映射内存页（Add）	添加一个物理内存页到页目录的某虚拟地址处。
移除内存页（Remove）	删除页目录某虚拟地址处的一个物理内存页。
构造页目录（Construct）	添加一个子页目录到父页目录的某虚拟地址处。

[1] MMU 系统一般也不支持此格式

操作	含义
析构页目录（Destruct）	删除父页目录某虚拟地址处映射的一个子页目录。

这六种操作在 MPU 和 MMU 上的实现是很不同的，也有不同的限制。请参看下面两节的解释以理解具体差别。

3.2.4 Implementation of Normal Multi-Level Page Table on MMU-Based Architectures

对于内存管理单元，页目录的实现是非常简单的，就是一个简单的线性表。比如，对于 x86-64 的页表，其第一级页目录（PML4）是固定的 512 个槽位，每个槽位代表 2^{39} 字节；第二级页目录（PDP）也是固定的 512 个槽位，每个槽位代表 2^{30} 字节；第三级页目录（PGD）也是固定的 512 个槽位，每个槽位代表 2^{21} 字节；第四级页目录（PTE）也是固定的 512 个槽位，每个槽位代表 2^{12} 字节。这四级页目录组成的基数树就是整个页表，如下图所示。其中，蓝色槽位代表指向子页目录的条目，而灰色槽位则代表被映射的页。此外，这些页目录的内存起始地址都应该对齐到 4kB，这样它们都正好占据一个页。

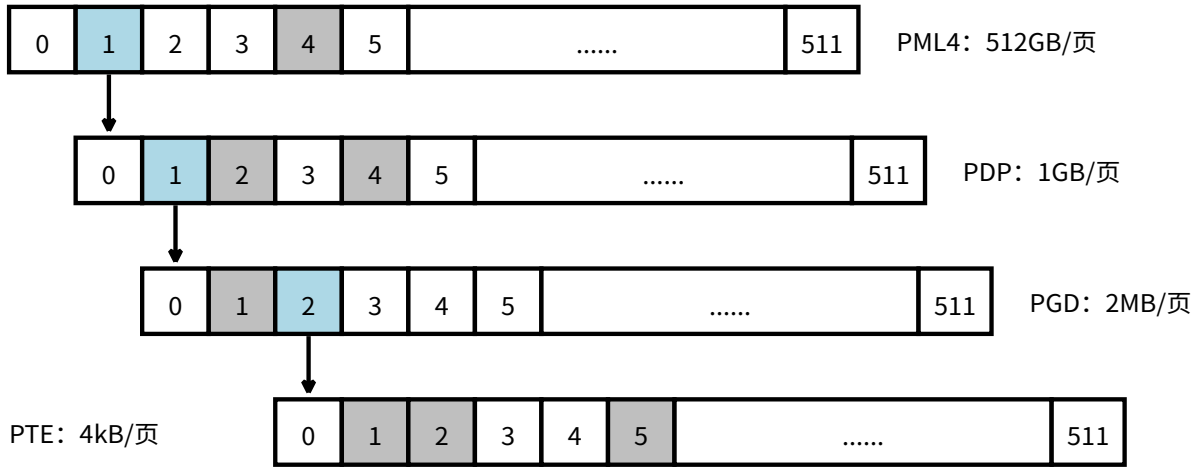

Figure 3-1 An Instance of Normal Multi-level Page Table

此外，如果允许一个页目录被构造进更高级页目录的任意虚拟地址槽位[1]，那么我们可以将内核配置为不检查起始虚拟地址是否合适。此时，我们使用的是正常多级页表，无需实现路径压缩。

由于处理器具备直接处理页表的硬件，因此我们不需要专门针对处理器生成页表元数据。但是需要注意，调用页表创建功能时，创建的页表应该合乎硬件页表查找机制的要求。对于那些纯软件填充 TLB 的 MMU，则没有这个要求了，可以随意创建逻辑上符合页表形式的树结构，也可以把它们看成是基于区域的 MMU，按照 3.2.5 所述的办法解决。

一些常见的 MMU 的特性如下：

Table 3-4 Specifications of Common MMUs

[1] 只要大小和数量级数合适

处理器	页表级数	页大小	其他特性
ARM926EJ-S	2 或 3 级	1MB，64kB，4kB，1kB	TLB 部分表项手动锁定
x86-64（AMD64）	3 级	1GB，2MB，4kB	额外的段式内存管理单元
Itanium（IA-64）	4 级	256MB，16MB，4MB，1MB，256kB，64kB，8kB，4kB	可部分手动填充的 TLB
e200（PowerPC）	不适用	1kB-4GB 的所有 2 的次方	纯软件填充的 TLB
ARMv7-A（32-bit）	2 或 3 级	4kB，64kB，1MB，16MB	TLB 部分表项手动锁定
ARMv8-M（64-bit）	3 或 4 级	4kB，16kB，64kB	虚拟化下可选的 2 阶转换
MIPS64	不适用	1kB-256MB 的所有 4 的次方	纯软件填充的 TLB
TMS320C66X	不适用	4kB-4GB 的所有 2 的次方	纯软件填充的 TLB

3.2.5 Implementation of Path-Compressed Page Table in MPU-Based Architectures

在内存保护单元[1]下，处理器往往不能直接识别多层的页表。这使得我们必须从页表生成 MPU 元数据用来在进程切换时高效地设置 MPU。而且，在 MPU 环境下，还有如下的几个特点：

1. MPU 的区域个数往往是有限的，比如 Cortex-M3 有 8 个区域，每个区域又可以划分为 8 个子区域。因此，我们在一个进程中最多只能允许同时映射 64 个区域，而且还要满足一系列苛刻条件。因此，我们可以考虑把页分成两类，一类是静态页，它们总是被映射，要求可预测性的应用可以使用它们，静态页的最大数目就是处理器允许的最大 MPU 区域个数；另一类是动态页，它们只在使用时被映射，不保证在任何时候都被映射，动态页的最大数目是没有限制的。如果访问到了一个当前没有映射的动态页，那么处理器会进入内存保护错误中断向量，然后我们手动查找页表来将该动态页加入 MPU 元数据，此时如果 MPU 区域不够可能会替换掉其他的动态页。动态页和 Emcraft 的 uCLinux 使用 MPU 的方法是非常相似的[2]。

2. 在一个页表里面，往往只有一两项是存在的。对于使用 MPU 的微控制器而言，维持多级页表的存在是没有必要的资源浪费，因此应该想办法对页表进行压缩。压缩页表和通常的页表相比，同一个页目录的不同中间页表项转换的地址位数可以是不同的。比如，在某个虚拟地址处，有一个很小的页，我们需要把它添加进访问范围。对于通常的页表，我们需要多级中间页目录，然后在最后一级页目录处，将这个页添加进去。而对于压缩页表，我们只要一级页目录就足以寻址该页。我们可以注明这个页目录的起始地址，数量级数和大小级数，然后直接将其构造进上级页目录中。当然，此时我们要求这一级页表所表示的虚拟地址范围落在上一级页目录的相应页表项允许的虚拟地址范围内。上述情况有一个例外：当架构实际上具备基于区域的 MMU 单元时，我们通常不使用路径压缩页表。

3. MPU 不能进行物理地址到虚拟地址的转换。因此，虚拟地址总是等于物理地址的。这使得我们在映射页时必须检查页的映射地址是否等于物理地址。

[1] 或者手动填充的、基于区域的内存管理单元，下同

4. 由于我们需要 MPU 元数据来加速 MPU 填充，因此当我们修改任何一级页目录时，我们都必须维持元数据和页表的一致性。不维持这种一致性也是可以的，不过如果如此我们就只能通过缺页中断来更新元数据。关于这种做法的详细信息请参看 3.2.5.2。

一个 Cortex-M4 上的路径压缩页表的例子如下图所示。绿色槽位代表路径压缩的指向子页目录的条目，蓝色槽位代表非路径压缩的指向子页目录的条目，而灰色槽位则代表被映射的页。各级页表前面的十六进制数代表了该页表的起始映射地址。

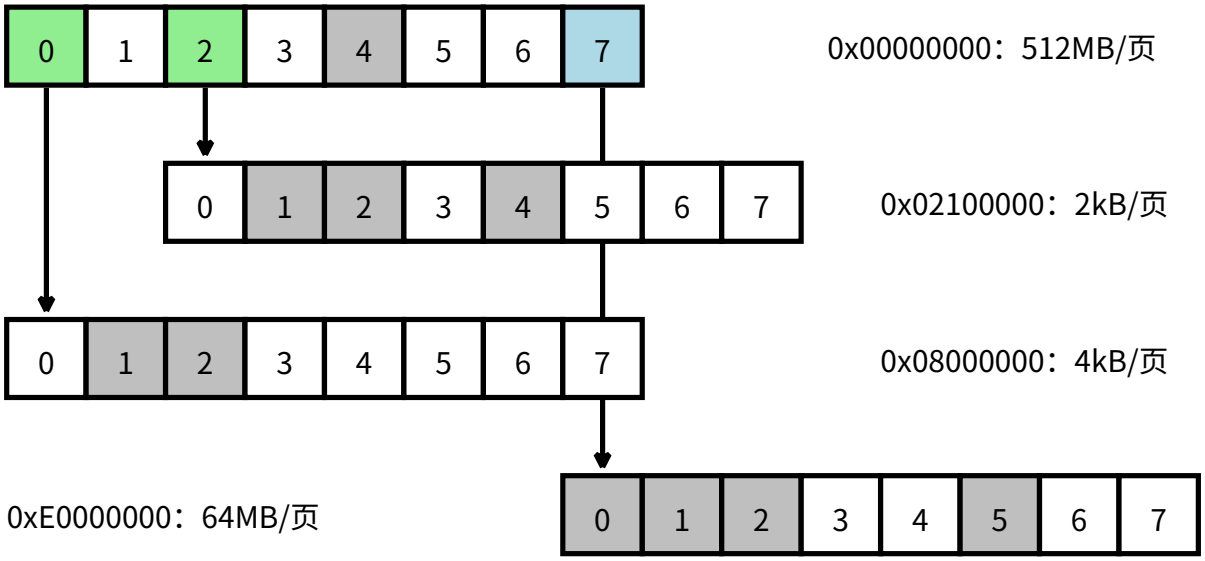

Figure 3-2 An Instance of Path-compressed Page Table

一些常见的 MPU 的特性如下：

Table 3-5 Specifications of Common MPUs

处理器	区域数量	区域组织	大小范围	对齐要求	其他特性
ARM V7-M	0 到 16 个	统一组织	128B-4GB	对齐到大小	8 个子区域
ARM V8-M	0 到 16 个	统一组织	128B-4GB	无	无
Tensilica L106	16 或 32 个	统一组织	4kB-1GB	无	无
MIPS M14k	1 到 16 个	统一组织	任意	无	可锁定为只读
PPC e200z4	32 个	代码/数据各 16 个	任意	无	无
AVR32	8 个	统一组织	4kB-4GB	对齐到大小	16 个子区域
MSP430FRXX	3 个	统一组织	任意	无	三段分段式
Coldfire-MCF	4 个	代码/数据各 2 个	16MB-4GB	对齐到大小	无

综上所述，对于 MPU 下页表的实现，常见的有以下两种形式：

3.2.5.1 Immediately Update MPU Metadata When Updating Page Tables

这种做法仅仅把 MPU 元数据放置在顶层页目录中，而且要求构造时从顶层构造起。如果任何一级页目录没有顶层页目录，自己也不是顶层页目录，那么就无法构造子页目录到这个页目录之内。此外，任意两个页表都不能共享页目录或页目录树。这种实现的制约如下：

Table 3-6 Restrictions on Immediately Update MPU Metadata When Updating Page Tables

操作	制约或缺点
创建进程	无制约。
更换进程页表	无制约。
切换进程	无制约，直接使用顶层页目录中的 MPU 元数据即可。
创建页目录	无制约。
删除页目录	自己不能有子页目录，自己也不能是别人的子页目录。
映射内存页	如果自己有顶层页目录，更新顶层页目录的 MPU 元数据。
移除内存页	如果自己有顶层页目录，更新顶层页目录的 MPU 元数据。
构造页目录	父页目录自己必须有顶层页目录，或者自己是顶层页目录；子页目录必须没有顶层页目录，而且自己不是顶层页目录[1]。添加子页目录的已映射页面到顶层页目录的 MPU 元数据。
析构页目录	子页目录必须有顶层页目录，而且自己不得含有任何子页目录[2]。从顶层页目录的 MPU 元数据中移除子页目录的已映射页面。
内存消耗	仅在顶层页目录有 MPU 元数据。

这个实现是单核系统上推荐的实现。这个实现最大限度地提高了基于 MPU 的微控制器的效率，使得我们往往能使用生成的 MPU 元数据批量设置 MPU 寄存器，而仅有一些不常用的功能的损失。当页表结构变化时，其更新 MPU 元数据的速度也是很快的。

3.2.5.2 Update MPU Metadata in Page Miss Exceptions

这种做法把放置在顶层页目录处的 MPU 的元数据看作是软件填充的 TLB，使用在内存保护中断中的软件页表遍历算法来在每次不命中时填充它。它不试图一次生成整个页表对应的 MPU 元数据，而是选择逐步生成它。在每次页表结构变化或页映射变化时，清空所有的 MPU 元数据，这相当于 MMU 架构下的 TLB 刷新。这种做法也支持静态页和动态页，并且静态页只在第一次访问时会出现缺页中断，后续访问则可以保证实时性。

Table 3-7 Restrictions on Update MPU Metadata in Page Miss Exceptions

[1] 而且也不可能有子页目录

[2] 其子页目录必须从本页目录中提前析构

操作	制约或缺点
创建进程	无制约。
更换进程页表	清空 MPU 元数据，准备重建。
切换进程	无制约。
创建页目录	无制约。
删除页目录	无制约。
映射内存页	无制约。
移除内存页	清空 MPU 元数据，准备重建。
构造页目录	无制约。
析构页目录	清空 MPU 元数据，准备重建。
内存消耗	仅在顶层页目录有 MPU 元数据。

这个实现保留了和 MMU 系统最大的兼容性，也是多核 MPU 架构上推荐的实现方法。它对页表的构造顺序没有要求，也方便在多核环境下进行操作，而且多个页表可以以任意方式共享一部分，但其实时性能比 3.2.5.1 中列出的方案稍差。

3.3 Page Table System Calls

与页表有关的内核功能如下：

Table 3-8 Page Table Related System Calls

调用号	类型	用途
RME_SVC_PGTBL_CRT	系统调用	创建页目录
RME_SVC_PGTBL_DEL	系统调用	删除页目录
RME_SVC_PGTBL_ADD	系统调用	映射内存页
RME_SVC_PGTBL_REM	系统调用	移除内存页
RME_SVC_PGTBL_CON	系统调用	构造页目录
RME_SVC_PGTBL_DES	系统调用	析构页目录

页表权能的操作标志如下：

Table 3-9 Page Directory Operation Flags

标志	位	用途
RME_PGTBL_FLAG_ADD_SRC	[0]	允许该页目录在页框传递作为来源目录。

标志	位	用途
RME_PGTBL_FLAG_ADD_DST	[1]	允许该页目录在权能传递作为目标目录。
RME_PGTBL_FLAG_REM	[2]	允许移除该页目录中的页框。
RME_PGTBL_FLAG_CON_CHILD	[3]	允许该页目录在页表构造中作为子页目录。
RME_PGTBL_FLAG_CON_PARENT	[4]	允许该页目录在页表构造中作为父页目录。
RME_PGTBL_FLAG_DES	[5]	允许析构该页目录。
RME_PGTBL_FLAG_PROC_CRT	[6]	允许在创建进程时将该页表作为进程的页表。
RME_PGTBL_FLAG_PROC_PGT	[7]	允许用该页表替换某进程的页表。
其他位	位段	操作范围属性。

关于上表中的位[6]和位[7]，请参看 4.4.1 和 4.4.4。在页表相关内核功能中填充操作标志时，要使用 RME_PGTBL_FLAG(HIGH,LOW,FLAGS)宏进行填充，其中 HIGH 为操作位置 Pos 的上限，LOW 为操作位置 Pos 的下限，[HIGH, LOW]组成的闭区间即为允许的 Pos 范围。FLAGS 则为位[7:0]中各个被允许的操作标志。

3.3.1 Page Directory Creation

该操作会创建一个页目录，并将其权能放入某个已存在的权能表。创建页目录操作需要如下几个参数：

Table 3-10 Parameters Needed for Page Directory Creation

参数名称	类型	域	描述
Svc_Num	ptr_t	N.D0	必须为 RME_PGTBL_CRT。
Cap_Captbl	cid_t	C	一个对应于必须拥有 RME_CAPTBL_FLAG_CRT 属性的权能表权能的权能号，该权能号对应的权能指向要接受此新创建的页目录权能的权能表。该权能号可以是一级或者二级查找编码。
Cap_Kmem	cid_t	P1.D1	一个内核内存权能号，其标识的内核内存范围必须能够放下整个页目录，并且要拥有 RME_KMEM_FLAG_PGTBL 属性。该权能号可以是一级或二级查找编码。
Cap_Pgtbl	cid_t	P1.Q1	一个对应于接受该新创建的页目录权能的权能表的某位置的权能号。该权能号对应的权能必须是空白的。该权能号只能是一级查找编码。
Vaddr	ptr_t	P2	新创建的页目录要使用的内核空间起始虚拟地址。
Start_Addr	ptr_t	P3	新创建的页目录的映射起始地址，最后一位为顶层标志，见下。
Top_Flag	ptr_t	P3[0]	该页目录是否是顶层页目录。“1”意味着该页目录为顶层。

参数名称	类型	域	描述
Size_Order	ptr_t	P1.Q0	该页目录的大小级数[1]。
Num_Order	ptr_t	N.D1	该页目录的数目级数。

该操作的返回值可能如下：

Table 3-11 Possible Return Values for Page Directory Creation

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Captbl 的一级/二级查找超出了范围。
	Cap_Kmem 的一级/二级查找超出了范围。
	Cap_Pgtbl 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl 的一级/二级查找的权能已经被冻结。
	Cap_Kmem 的一级/二级查找的权能已经被冻结。
	Cap_Pgtbl 被冻结，或者其它核正在该处创建权能。
RME_ERR_CAP_TYPE	Cap_Captbl 不是权能表权能。
	Cap_Kmem 不是内核内存权能。
RME_ERR_CAP_FLAG	Cap_Captbl 无 RME_CAPTBL_FLAG_CRT 属性。
	Cap_Kmem 无 RME_KMEM_FLAG_PGTBL 属性，或范围错误。
RME_ERR_CAP_EXIST	Cap_Pgtbl 不是空白权能。
RME_ERR_CAP_KOTBL	分配内核内存失败。
RME_ERR_PGT_HW	底层硬件制约，不允许创建这样的页目录。

3.3.2 Page Directory Deletion

该操作会删除一个页目录。被删除的页目录必须不含有子页目录。删除页目录需要以下几个参数：

Table 3-12 Parameters Needed for Page Directory Deletion

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_PGTBL_DEL。
Cap_Captbl	cid_t	C	一个对应于必须拥有 RME_CAPTBL_FLAG_DEL 属性的权能表权能的权能号，该权能号对应的权能指向含有正被删除的页目录权能的权能表。该权能号可以是一级或者二级查找编码。

[1] 指每个页表项代表的内存页大小

参数名称	类型	域	描述
Cap_Pgtbl	cid_t	P1	一个对应于将被删除的页目录权能的权能号。该权能号对应的权能必须是一个页目录权能。该权能号只能是一级查找编码。

该操作的返回值可能如下：

Table 3-13 Possible Return Values for Page Directory Deletion

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Captbl 的一级/二级查找超出了范围。
	Cap_Pgtbl 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl 的一级/二级查找的权能已经被冻结。
	Cap_Pgtbl 未被冻结。
RME_ERR_CAP_TYPE	Cap_Captbl 不是权能表权能。
	Cap_Pgtbl 不是页目录权能。
RME_ERR_CAP_NULL	Cap_Pgtbl 为空白权能。
	两个核同时试图删除该页目录，此时未成功的核返回该值。
RME_ERR_CAP_FLAG	Cap_Captbl 无 RME_CAPTBL_FLAG_DEL 属性。
RME_ERR_CAP_QUIE	Cap_Pgtbl 不安定。
RME_ERR_CAP_REFCNT	Cap_Pgtbl 的引用计数不为 0，或者不为根权能。
RME_ERR_PGT_HW	底层硬件制约，不允许删除这个页目录。这可能是因为页目录中含有子页目录或者等等其他原因。

3.3.3 Page Mapping

该操作会将一个页目录中的某个页表项的某一部分传递到另外一个页目录的空白位置中。在页目录之间进行页表项传递需要以下几个参数：

Table 3-14 Parameters Needed for Page Mapping

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_PGTBL_ADD。
Cap_Pgtbl_Dst	cid_t	P1.D1	一个对应于必须拥有 RME_PGTBL_FLAG_ADD_DST 属性的页目录权能的权能号，该权能号对应的权能指向目标页目录。该权能号可以是一级或者二级查找编码。
Pos_Dst	ptr_t	P1.D0	一个该目标页目录中要接受传递的目标页表项位置。该页表项必

参数名称	类型	域	描述
			须是空白的。
Flags_Dst	ptr_t	P3.D1	目标页表项的属性。这个属性限制了目标页表项的特性。
Cap_Pgtbl_Src	cid_t	P2.D1	一个对应于必须拥有 RME_PGTBL_FLAG_ADD_SRC 属性的页目录权能的权能号，该权能号对应的权能指向源页目录。该权能号可以是一级或者二级查找编码。
Pos_Src	ptr_t	P2.D0	一个源页目录中要被传递的源页框位置。该页框必须是被映射的。
Index	ptr_t	P3.D0	要被传递的源页框中的子位置。

该操作的返回值可能如下：

Table 3-15 Possible Return Values for Page Mapping

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Pgtbl_Dst 或 Cap_Pgtbl_Src 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Pgtbl_Dst 或 Cap_Pgtbl_Src 的一级/二级查找的权能被冻结。
RME_ERR_CAP_TYPE	Cap_Pgtbl_Dst 或 Cap_Pgtbl_Src 不是页目录权能。
RME_ERR_CAP_FLAG	Cap_Pgtbl_Src 无 RME_PGTBL_FLAG_ADD_SRC 属性。
	Cap_Pgtbl_Dst 无 RME_PGTBL_FLAG_ADD_DST 属性。
	Cap_Pgtbl_Dst 或 Cap_Pgtbl_Src 的操作范围属性不允许该操作。
RME_ERR_PGT_ADDR	目标页目录的大小级数比源页目录的大小级数大，因此不能映射。
	Pos_Dst 或 Pos_Src 超出了目标页目录或者源页目录的页表项数目。
	Index 超出了子位置的最大编号。
	在开启了物理地址等于虚拟地址的检查时，映射的物理地址和目标虚拟地址不同。
RME_ERR_PGT_HW	源页目录查找失败。这可能是由于源页目录的该位置为空。
RME_ERR_PGT_MAP	尝试映射，由于硬件原因失败。具体的失败原因与硬件有关。
RME_ERR_PGT_PERM	目标页的访问控制标志不是源页的访问控制标志的子集。

3.3.4 Page Removal

该操作会将一个页目录中的某个页表项除去，使该位回归空白状态。移除内存页需要如下参数：

Table 3-16 Parameters Needed for Page Removal

参数名称	类型	域	描述

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_PGTBL_REM。
Cap_Pgtbl	cid_t	P1	一个对应于必须拥有 RME_PGTBL_FLAG_REM 属性的页目录权能的权能号，该权能号对应的权能指向目标页目录。该权能号可以是一级或者二级查找编码。
Pos	ptr_t	P2	一个该目标页目录中要除去的页表项位置。该页表项必须是一个被映射的内存页。

该操作的返回值可能如下：

Table 3-17 Possible Return Values for Page Removal

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Pgtbl 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Pgtbl 的一级/二级查找的权能被冻结。
RME_ERR_CAP_TYPE	Cap_Pgtbl 不是页目录权能。
RME_ERR_CAP_FLAG	Cap_Pgtbl 无 RME_PGTBL_FLAG_REM 属性。
	Cap_Pgtbl 的操作范围属性不允许该操作。
RME_ERR_PGT_ADDR	Pos 超出了目标页目录的页表项数目。
RME_ERR_PGT_MAP	尝试除去，由于硬件原因失败。具体的失败原因与硬件有关。

3.3.5 Page Table Construction

该操作会将指向子页目录的物理地址指针放入父页目录的某个空白位置之中。如果使用压缩页表，子页目录的大小必须小于等于父页目录的一个页，否则子页目录的大小必须正好等于父页目录的一个页。构造页目录需要如下参数：

Table 3-18 Parameters Needed for Page Table Construction

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_PGTBL_CON。
Cap_Pgtbl_Parent	cid_t	P1.D1	一个对应于必须拥有 RME_PGTBL_FLAG_CON_PARENT 属性的页目录权能的权能号，该权能号对应的权能指向父页目录。该权能号可以是一级或者二级查找编码。
Pos	ptr_t	P2	一个该目标页目录中要接受传递的目标页表项位置。该页表项必须是空白的。

参数名称	类型	域	描述
Cap_Pgtbl_Child	cid_t	P1.D0	一个对应于必须拥有 RME_PGTBL_FLAG_CON_CHILD 属性的页目录权能的权能号，该权能号对应的权能指向子页目录。该权能号可以是一级或者二级查找编码。
Flags_Child	ptr_t	P3	子页目录被映射时的属性。这个属性限制了该映射以下的所有页目录的访问权限。对于不同的架构，这个位置的值的意义也不相同。对于有些不支持页目录属性的架构，这个值无效。

该操作的返回值可能如下：

Table 3-19 Possible Return Values for Page Table Construction

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Pgtbl_Parent 或 Cap_Pgtbl_Child 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Pgtbl_Parent 或 Cap_Pgtbl_Child 的一级/二级查找的权能被冻结。
RME_ERR_CAP_TYPE	Cap_Pgtbl_Parent 或 Cap_Pgtbl_Child 不是页目录权能。
RME_ERR_CAP_FLAG	Cap_Pgtbl_Parent 无 RME_PGTBL_FLAG_CON_PARENT 属性。
	Cap_Pgtbl_Child 无 RME_PGTBL_FLAG_CON_CHILD 属性。
	Cap_Pgtbl_Parent 的操作范围属性不允许该操作。
RME_ERR_PGT_ADDR	Pos 超出了父页目录的页表项数目。
	子页目录的总大小大于父页目录的一个页的大小。
	在开启了物理地址等于虚拟地址的检查时，映射的物理地址和目标虚拟地址不相等。
RME_ERR_PGT_MAP	尝试构造，由于硬件原因失败。具体的失败原因与硬件有关，可能是硬件不支持此种映射。

3.3.6 Page Table Destruction

该操作会将一个页目录中的某个子页目录除去，使该位回归空白状态。析构页目录需要如下参数：

Table 3-20 Parameters Needed for Page Table Destruction

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_PGTBL_DES。

参数名称	类型	域	描述
Cap_Pgtbl	cid_t	P1	一个对应于必须拥有 RME_PGTBL_FLAG_DES 属性的页目录权能的权能号，该权能号对应的权能指向目标页目录。该权能号可以是一级或者二级查找编码。
Pos	ptr_t	P2	一个该目标页目录中要除去的子页目录位置。该页表项必须是一个被映射的页目录。

该操作的返回值可能如下：

Table 3-21 Possible Return Values for Page Table Destruction

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Pgtbl 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Pgtbl 的一级/二级查找的权能被冻结。
RME_ERR_CAP_TYPE	Cap_Pgtbl 不是页目录权能。
RME_ERR_CAP_FLAG	Cap_Pgtbl 无 RME_PGTBL_FLAG_DES 属性。
	Cap_Pgtbl 的操作范围属性不允许该操作。
RME_ERR_PGT_ADDR	Pos 超出了目标页目录的页表项数目。
RME_ERR_PGT_MAP	尝试除去，由于硬件原因失败。具体的失败原因与硬件有关。

3.4 Kernel Memory System Calls

与内核内存有关的内核功能只有一个，就是进行内核内存权能的传递。初始的内核内存权能是在系统启动时创建的，并且无法删除。其传递产生的子权能无法被删除，只能被移除。内核内存权能不仅有操作标志，还有一个对齐到 64Byte 的范围值。

内核内存权能的操作标志如下：

Table 3-22 Kernel Memory Operation Flags

标志	位	用途
RME_KMEM_FLAG_CAPTBL	[0]	允许在该段内核内存上创建权能表。
RME_KMEM_FLAG_PGTBL	[1]	允许在该段内核内存上创建页目录。
RME_KMEM_FLAG_PROC	[2]	允许在该段内核内存上创建进程。
RME_KMEM_FLAG_THD	[3]	允许在该段内核内存上创建线程。
RME_KMEM_FLAG_SIG	[4]	允许在该段内核内存上创建信号端点。

标志	位	用途
RME_KMEM_FLAG_INV	[5]	允许在该段内核内存上创建线程迁移调用。

在进行内核内存权能的传递时，由于还需要传入一个范围参数，因此仅用一个参数位置 P3 是无法完全传递所需信息的。此时，需要使用系统调用号 N 的一部分和权能表权能号 C 来传递这些参数。具体的参数传递规则如下：

Table 3-23 Parameters Needed for Kernel Memory Capability Delegation

参数名称	类型	域	描述
Svc_Num	ptr_t	N[5:0]	必须为 RME_SVC_CAPTBL_ADD。
Cap_Captbl_Dst	cid_t	P1.D1	一个对应于必须拥有 RME_CAPTBL_FLAG_ADD_DST 属性的权能表权能的权能号，该权能号对应的权能指向目标权能表。该权能号可以是一级或者二级查找编码。
Cap_Dst	cid_t	P1.D0	一个对应于将接受被传递的权能的权能号。该权能号对应的权能必须是空白的。该权能号只能是一级查找编码。
Cap_Captbl_Src	cid_t	P2.D1	一个对应于必须拥有 RME_CAPTBL_FLAG_ADD_SRC 属性的权能表权能的权能号，该权能号对应的权能指向源权能表。该权能号可以是一级或者二级查找编码。
Cap_Src	cid_t	P2.D0	一个对应于将传递的权能的权能号。该权能号对应的权能必须不为空白而且没有冻结。该权能号只能是一级查找编码。
Flags	ptr_t	P3	描述见下文。
Ext_Flags	ptr_t	N:C	描述见下文。

内核内存权能的传递中，P3 和 N:C 共同决定了新产生的内核内存权能的扩展操作标志属性。N:C 表示将半字 N 和半字 C 组合起来，其中 N 处于高半字，C 处于低半字，共同组成一个字。由于 RME 仅仅使用了 N 的最后六个二进制位表示系统调用号，因此剩余的二进制位可以被用来表示其他信息。N 和 C 组合起来一共有 X-6 个二进制位（X 为按照 Bit 计算的机器字长），加上 P3 提供的 X 个二进制位，一共有 2X-6 个二进制位。其中操作标志会用去 6 位，因此内核内存的上界和下界可以各分配 X-6 位，这正好能表示对齐到 64 字节的内存地址。

P3（Flags）的具体意义如下：

Table 3-24 Detailed Explanation of P3 (Flags) in Kernel Memory Capability Delegation

位段范围	位段意义
高半字（D1）	内核内存地址上限的高半字。
低半字（D0）	内核内存地址下限的高半字。

N:C（Ext_Flags）的具体意义如下：

Table 3-25 Detailed Explanation of N:C (Ext_Flags) in Kernel Memory Capability Delegation

位段范围	位段意义
高半字清零其最后六位（{D1[X-1:6]:0[5:0]}）	内核内存地址上限的低半字，对齐到 64Byte。
低半字清零其最后六位（{D0[X-1:6]:0[5:0]}）	内核内存地址下限的低半字，对齐到 64Byte。
低半字的最后六位（{D0[5:0]}）	内核内存权能的操作标志位。

需要注意的是，传入内核内存地址时，传入的上限值不包括自身。例如，传入一个地址范围 0xC0000000-0xC1000000，那么 0xC1000000 是不包括在可操作的合法地址之内的，也即实际上允许的内核内存范围是 0xC0000000-0xC0FFFFFF。上限必须大于下限，否则会返回错误。上限和下限在传入时都会被掩蔽后六位，对齐到 64Byte。如果内核内存登记表被配置为使用比 64Byte 更大的槽位，那么内核会自动将传入的下限向上取整，上限向下取整，对齐到槽位大小。

本系统调用的可能返回值和一般的 RME_SVC_CAPTBL_ADD 操作完全一致，请参见 2.3.3。

3.5 Bibliography

[1] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A kernel for scalable predictability," in Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE, 2015, pp. 121-132.

[2] Emcraft Systems. uCLinux(2017). https://github.com/EmcraftSystems/linux-emcraft

[3] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, et al., "seL4: formal verification of an OS kernel," presented at the Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, Big Sky, Montana, USA, 2009.

[4] Fiasco.OC website (2017). http://os.inf.tu-dresden.de/fiasco

Chapter 4 Process and Thread Management

4.1 Introduction to Processes and Threads

进程（Process）和线程（Thread）是 RME 对其上执行的程序提供的基本抽象。RME 的进程和线程抽象层次较低，这与 Linux 等传统操作系统的高层次抽象实现是非常不同的。与传统操作系统相比，RME 的实现具备更高的灵活性和性能。

4.1.1 Introduction to Processes

在 RME 中，一个进程指的是拥有一个独立地址空间[1]和一个权能表的最小保护域（Protection Domain）。页表决定了进程的地址空间，权能表则决定了该进程中线程[2]的权限。RME 不在内核态实现关于进程的其他所有信息；这些信息被全部留到用户态实现。

事实上，RME 中的进程与传统操作系统的进程相比几乎没有任何相似点，它们实际上是超轻量级的虚拟机，或者也可以被看作是容器（Container）。各个进程的权能表都有相互独立的权能号命名空间，没有系统全局的权能号。将多个进程合起来看成为一个相互隔离但又有一定联系的软件功能单元也是可以的。RME 进程在被用作轻量级虚拟机时，可以提供近乎于物理机的性能[4]。

4.1.2 Introduction to Threads

在 RME 中，一个线程指拥有一个独立指令流和一个栈的最小可调度实体。线程在 RME 中具有多个状态来标志其运行情况。本章所指的线程都是内核态线程，用户态线程由于与内核本身无关，因此不在此讨论。在不同的系统中，调度系统（Scheduler）主要有四种设计策略，分别如下：

1. 调度器完全在内核态。这一种实现之中，各个线程的时间片是完全自动填充的，并且调度策略的实现也完全处于内核态。采用此类实现的典型系统包括 Linux，全部的 L4 和 VxWorks。此类系统是非常传统的，在此不加叙述。

2. 调度器完全在用户态，即便是优先级的判断和控制逻辑也在内核外部。当中断发生时，就切换到中断对应的线程进行中断处理；至于优先级和该执行哪个中断处理程序则完全由用户态调度器决定。这种实现具有极强的灵活性，并且可以很方便地实现闲置窃取调度（Slack Stealing Scheduling）。这是最原始的用户态调度设想，但是由于它引起了大量的线程切换，有过高的额外开销而没有使用价值。

3. 抢占优先级在内核内部，包括就绪队列的其他部分在内核外部。当中断来临时，进行抢占并且立即运行中断处理线程。中断处理线程会启动中断后处理线程，然后在接收点上阻塞。如果在这个时间段内有其他的低优先级中断到来，那么低优先级中断会转化为送往对应调度器的调度器事件。高优先级的中断后处理线程完成中断处理后，调用调度器，处理在这段时间之内发生的所有调度器事件，并且决定下一个运行的线程。如果在这段时间之内，分配给后处理线程的时间片耗尽，那么我们切换到任意一个还有时间片的线程。它的坏处是，如果有一个低优先级的线程在高优先级线程执行时由于某中断而就绪，

[1] 也即一个独立页表

[2] 或线程迁移调用，见 5.1.2

我们没办法在高优先级结束之后立即执行低优先级线程。我们必须先切换到调度器，然后等待调度器反复从系统中读取出全部的事件，并对这些事件的轻重缓急加以判断之后才能处理低优先级中断或者线程。这在某些场景下是不可忍受的。采用此类设计的典型系统为 Composite[1]。值得一提的是，Composite 的优先级不是一个标量（Scalar）而是一个高维张量（Tensor），而且将时间片作为一种独立资源进行管理[3]。

4. 抢占优先级和运行队列在内核中，包括时间片管理等的其他部分在内核外。这是 RME 选择的解决方案：它更接近传统系统但又实现了用户态调度。好处是可以减小中断延迟，并且能够确保现在运行的线程总是就绪线程中优先级最高的。坏处则是每一次处理任何可能导致上下文切换的操作，都要处理内核的运行队列。因此，RME 的内核队列维护器是极其高效的，这样就可以将这种影响降低到最小。此外，内核仍然要给线程发送调度器事件来配合用户态调度。与 Composite[1]不同，RME 的优先级实现为一个数值标量。这使得系统的优先级必须被平铺（Flattened），不利于在较复杂场景中彻底消除各个子系统之间的干扰，但可以大大加快调度原语的速度。在未来的版本中，RME 可能会在内核实现张量形式的优先级。

在 RME 中，每个线程都有一个抢占优先级，其数值越大，则优先级越高[1]。优先级的数量由宏 RME_MAX_PREEMPT_PRIO 配置，系统中的优先级为从 0 到 RME_MAX_PREEMPT_PRIO -1。其最高可以被配置为 2 的字长的一半次方。比如 32 位系统中，优先级的最大数量为 2^{16}=65536。此外，每个线程在被创建时都会被指定一个优先级上限，一个线程不能通过系统调用创建拥有更高优先级上限的线程。在对一个线程做处理器绑定操作或优先级变更操作时，无法把被操作线程的优先级提高到其优先级上限以上。但是，一个拥有低优先级上限的线程可以把另一个线程的优先级提高到低优先级线程自身的优先级上限以上，只要被操作线程的优先级不被提高到超过被操作线程的优先级上限。优先级上限的实现参考了 seL4[2]。

与其他也使用时间片传递的系统如 NOVA[5]等不同，RME 的线程时间片不会在某个时点自动被系统填充，这也是 RME 的调度器应当被认为在内核外的原因。如果一个线程的时间片耗尽，那么该线程将一直停止执行直到有其它线程向它转移时间片为止。

每个线程都具有一个用户决定的线程标识符（Thread IDentifier，TID）。这个标识符是由用户在绑定（Bind）线程时决定的。在 32 位系统下，该标识符的范围为 0-2^{32-2}；考虑到目前绝大多数 32 位设备现在都是嵌入式设备[2]，因此即便在某些状况下用户态需要全局唯一的 TID，这个范围也是够用的。在 64 位系统下这个范围是 0-2^{64-2}，在可预见的将来也是足够使用的。

此外，RME 不在内核态实现诸如线程本地存储（Thread Local Storage，TLS）等其他功能。这些功能会被用户态库实现。

4.2 Process Operations and States

4.2.1 Process Creation and Deletion

[1] 这与某些系统是相反的；与常见的 FreeRTOS 则是一致的

[2] 基本上是微控制器或低端微处理器

要创建进程，需要一个权能表和一个可以作为顶层的页目录[1]。进程在 RME 中仅仅起到一个隔离保护域的作用；它没有独立的状态。销毁一个进程中所有的线程和线程迁移调用入口[2]并不会导致进程被销毁。

要删除进程，需要该进程中没有任何的线程存在，也没有任何的线程迁移调用入口[3]存在。只要通过进程权能指明要删除的进程就可以了。

4.2.2 Changing Capability Table or Page Table of Processes

进程的权能表和页表是可以在系统运行过程中动态更换的，并且动态更换总是立即生效。动态更换操作通常用来实现虚拟化功能中虚拟机或容器的切换，也可以用来实现快速跨进程通信等其他功能。

4.3 Thread Operations and States

4.3.1 Thread Operation Overview

在 RME 中，线程是需要被绑定到某个 CPU 才能被操作的，而且只有它被绑定的那个 CPU 内核可以操作它。如果想要更改可以操作该线程的 CPU，那么需要修改其绑定。在系统中线程有如下几个状态：

Table 4-1 States of Threads

状态	名称	说明
运行	RME_THD_RUNNING	线程正在运行。
就绪	RME_THD_READY	线程处于就绪态。
超时	RME_THD_TIMEOUT	线程的时间片被用尽。
等待	RME_THD_BLOCKED	线程被阻塞在某个接收点上。
错误	RME_THD_FAULT	线程执行过程中发生了一个错误，被迫中止。

这几个状态是可以互相转换的。当线程被创建时，它处于 RME_THD_TIMEOUT 状态，这表示它没有被绑定到某个 CPU，也没有被分配时间片。接下来，我们将它绑定到某个核，此时它仍然处于 RME_THD_TIMEOUT 状态。然后设置它的入口和栈。最后，我们分配时间片给它。如果它是该 CPU 上优先级最高的线程，那么会抢占当前线程，进入 RME_THD_RUNNING 状态，否则会被放入内核就绪队列，进入 RME_THD_READY 状态。

如果线程在执行过程中在某个接收点上被阻塞，线程会转换成 RME_THD_BLOCKED 状态。这种情况下，当阻塞被解除时线程会视优先级是否为系统中最高的而回到 RME_THD_READY 状态或者 RME_THD_RUNNING 状态。

---

[1] 也即顶层页目录权能，详见 3.2.2.2

[2] 详见 5.1.1

[3] 详见 5.1.1

如果线程在执行过程中出现了一次错误，那么线程会转换到 RME_THD_FAULT 状态，并且向其父线程[1]发送一个调度器事件。如果线程在绑定时还指定了一个信号端点[2]，那么该信号端点也会收到一个信号。要解除错误状态，需要重置其执行栈和入口，才能把线程置于 RME_THD_TIMEOUT 状态。

如果该线程在运行时用尽了自己的所有时间片，或者在时间片传递中将自己的时间片全部传递出去，或者在切换到其他线程时选择放弃当前所有时间片，那么它会进入超时 RME_THD_TIMEOUT 状态，并且向其父线程发送一个调度器事件。如果线程在绑定时还指定了一个信号端点，那么该信号端点也会收到一个信号。

当解除一个线程对某 CPU 的绑定时，该线程必须没有子线程。解除绑定时，对应于该线程的父线程调度器事件如果存在，那么也会被去掉。在 RME_THD_BLOCKED 下被解除绑定，那么当前阻塞会直接返回一个 RME_ERR_SIV_FREE 的错误码[3]。此外，在线程不是 RME_THD_FAULT 状态时，如果解除绑定，那么该线程的状态都将变成 RME_THD_TIMEOUT 状态。如果在 RME_THD_FAULT 状态下解除线程的绑定，那么线程将仍会维持在 RME_THD_FAULT 状态下。

4.3.2 Thread Creation and Deletion

当创建线程时，需要指明线程所在的进程。入口和堆栈等属性是通过其他内核调用设置的。

当线程被删除时，它必须被解除绑定。在删除线程时，我们会清空它的线程迁移调用栈[4]。

4.3.3 Binding and Freeing Threads from CPUs

创建线程后需要把它绑定到某个 CPU 才能够操作，而如果想要更换这种绑定，那么就需要先解除它对当前 CPU 的绑定。

绑定线程到某 CPU 需要指明线程的优先级，线程的父线程和一个 TID。在哪个 CPU 上调用绑定函数，该线程即会被绑定到哪个 CPU。绑定操作通过使用读-改-写（Read-Copy-Update，RCU）原子操作来进行，保证在多个 CPU 同时进行的操作中，只有一个会获得成功。另外还有一个信号端点作为可选参数，如果在绑定时指定了一个信号端点，那么该信号端点将接收到此线程的调度器信号。值得注意的是，TID 并不需要是全局唯一的，用户态可以决定 TID 的分配策略。

解除绑定则仅仅需要指明需要解除绑定线程即可。当一个线程被解除了对某个 CPU 的绑定后，我们就可以把它绑定到其他的 CPU 了，也即实现了线程在不同处理器之间的迁移。这和那些线程在创建时就被永久绑定到某处理器的系统，如 Composite 等[1]不同。

4.3.4 Setting the Execution Properties of Threads

[1] 调度器线程

[2] 详见 5.1.2

[3] 详见 5.1.2

[4] 详见 5.1.1

在完成线程绑定后，我们需要设置线程的入口，堆栈和自定义参数。这三个值在会被传递给线程的寄存器组，在线程第一次运行时，用户态库根据前两个参数来找到用户态的线程入口和线程栈。需要注意的是，在区分虚拟地址和实地址的架构上，这两个值都是虚拟地址。

4.3.5 Setting the Hypervisor Properties of Threads

如果需要内核内建的准虚拟化虚拟机支持，那么需要设置线程的虚拟机属性。线程的虚拟机属性是一个指向专用虚拟机内存的指针。当线程没有设置虚拟机属性时，在线程切换时其寄存器组默认被保存在内核对象中；当虚拟机属性被设置时，则会保存虚拟机属性到该地址，方便运行在用户态的虚拟机监视器随时修改虚拟机线程的运行状态。

如果内核在配置时就决定不使用该功能[1]，那么该功能不能使用。

4.3.6 Timeslice Allocation, Priority Modification and Thread Execution

在设置完现成的入口和栈之后，我们就可以给线程分配时间片，从而开始线程的运行了。RME 系统的时间片分配是由用户态调度器树组织的，而且每个 CPU 都有这样的一个调度器树，用来管理本 CPU 的运行时间分配。首先由系统的各 CPU 上的 Init 线程给用户态调度器分配时间片[2]，然后再由这些用户态调度器按照它们各自的调度算法，把它们的时间片按照合适的比例传递给它们的各个子调度器，依此类推层层分配，从而完成线程的层次化调度。这种组织使得准虚拟化其他操作系统变得非常容易。各个 CPU 上的 Init 线程拥有无限的时间片，也即如果没有任何其他线程可以运行，我们总是去运行这个 CPU 上的 Init 线程[3]。

在线程时间片分配完成后，线程即被放入每个核的就绪序列（Ready Queue），并且会和当前运行的线程进行优先级比较。如果当前运行的线程的优先级较低，那么该线程会被立即投入运行。

我们可以修改一个已经被绑定到某个 CPU 的线程的优先级。在优先级修改后，如果该线程的优先级是最高的，而且它处于 RME_THD_READY 状态，那么它会被立即调度运行。

线程间传递时间片的做法借鉴了 Composite 的 TCap 机制，并且加以简化和改进[3]。按持有时间片的多少和线程创建时间分类，在系统中有三种线程，分别如下表所示：

Table 4-2 Categories of Threads

种类	创建时间	特点
通常线程	在系统启动之后	时间片有限，并且持有总量不超过 RME_THD_MAX_TIME。
无限线程	在系统启动之后	时间片无限，时间片持有量记做 RME_THD_INF_TIME。
初始线程	在系统启动之时	时间片无限，时间片持有量记做 RME_THD_INIT_TIME。

[1] 详见 7.3.2

[2] Init 线程的时间片为 RME_THD_INIT_TIME，是无限的

[3] 这也是为 Init 线程不允许阻塞在任何端点上的原因，详见 5.3

几个宏的意义如下表所示：

Table 4-3 Meaning of Timeslice Macro Definitions

宏名	意义
RME_THD_INIT_TIME	为最大正整数数值，在 32 位系统下为 0x7FFFFFFF。
RME_THD_INF_TIME	为 RME_THD_INIT_TIME-1，在 32 位系统下为 0x7FFFFFFE。
RME_THD_MAX_TIME	等于 RME_THD_INF_TIME。

其中通常线程和无限时间片线程允许阻塞，也允许失去自己的所有时间片；初始线程则不允许这两点。在各个线程之间传递时间片有如下三种：

第一种是通常传递，这种传递会传递有限数量的时间片到其他线程；

第二种是无限传递，这种传递会传递无限数量的时间片到其他线程；

第三种是回收传递，这种传递会将源线程的时间片全部转移给目标线程，并且清零源线程的时间片。

值得注意的是，时间片传递是原子性的，也就是要么指定的量都被传递，要么就都不被传递，不可能发生部分传递的情况。三种传递的规则如下表所示。

Table 4-4 Rules of Normal Timeslice Transfers

通常传递	源线程	初始线程	无限线程	通常线程
目标线程	初始线程	--	--	T-
	无限线程	--	--	T-
	通常线程	-A	TA	TA

Table 4-5 Rules of Infinite Timeslice Transfers

无限传递	源线程	初始线程	无限线程	通常线程
目标线程	初始线程	--	--	S-
	无限线程	--	--	S-
	通常线程	-I	-I	TA

Table 4-6 Rules of Revoking Timeslice Transfers

回收传递	源线程	初始线程	无限线程	通常线程
目标线程	初始线程	--	S-	S-
	无限线程	--	S-	S-
	通常线程	-I	SI	TA

表中：

“--”代表对源线程或目标线程没有影响；

“T”代表如果源线程的所有时间片都被转移出去，那么源线程会超时。

“S”代表源线程一定会超时。

“A”代表如果目标线程的时间片不溢出，那么会接受这些时间片。

“I”代表目标线程一定会变成无限线程。

4.3.7 Thread Scheduling Overview

最后，完善的线程状态转移图如下所示：

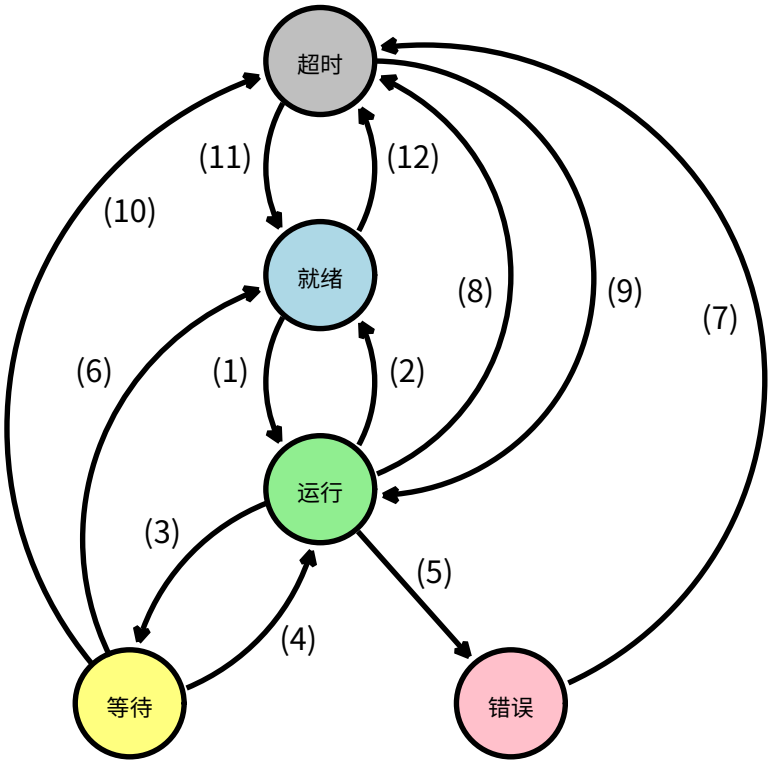

Figure 4-1 Thread State Transition Diagram

图中各个数字标号的意义如下所示：

Table 4-7 Meaning of Different Numbers in Thread State Transition Diagram

标号	代表意义
(1)	当前核上它是优先级最高的线程，因此由就绪态转入运行态。
(2)	当前核上有更高优先级的线程打断了它的执行，因此由运行态转入就绪态。
(3)	线程在一个信号端点处阻塞，因此由运行态转入等待态。
(4)	线程收到了来自信号端点的信号，解除阻塞，而且是当前核上优先级最高的线程，并且其时间片

标号	代表意义
	没有耗尽，因此由等待态转入运行态。
(5)	线程执行过程中发生了一个不可恢复错误，因此从运行态转入错误态。
(6)	线程收到了来自信号端点的信号，解除阻塞，并且其时间片没有耗尽，但是它不是当前核上优先级最高的线程，因此由等待态转入就绪态。
(7)	线程发生错误后藉由重新设置其执行信息而恢复到可正常执行状态，但是其时间片已经在发生错误时被剥夺，因此由错误态转入超时态。
(8)	线程在运行过程中耗尽了自己的时间片，因此由运行态转入超时态。
(9)	线程被其他线程传递了时间片，重新进入可以运行的状态，而且是当前核上优先级最高的线程，因此由超时态转入运行态。
(10)	线程在等待信号端点时，被其他线程把自己的时间片传递出去，使得自己的时间片归零，因此在收到信号解除阻塞后由等待态直接转入超时态。
(11)	线程被其他线程传递了时间片，重新进入可以运行的状态，但是由于自己的优先级并非当前可运行线程中最高的，因此由超时态转入就绪态。
(12)	线程在就绪状态时，被其他线程把自己的时间片传递出去，使得自己的时间片归零，因此由就绪态转入超时态。

4.4 Process System Calls

与进程有关的内核功能如下：

Table 4-8 Process Related System Calls

调用号	类型	用途
RME_SVC_PROC_CRT	系统调用	创建进程
RME_SVC_PROC_DEL	系统调用	删除进程
RME_SVC_PROC_CPT	系统调用	替换进程的权能表
RME_SVC_PROC_PGT	系统调用	替换进程的页表[1]

进程权能的操作标志如下：

Table 4-9 Process Operation Flags

标志	位	用途
RME_PROC_FLAG_INV	[0]	允许在该进程内创建线程迁移调用。

[1] 顶层页目录

标志	位	用途
RME_PROC_FLAG_THD	[1]	允许在该进程内创建线程。
RME_PROC_FLAG_CPT	[2]	允许替换该进程的权能表。
RME_PROC_FLAG_PGT	[3]	允许替换该进程的页表[1]。

关于上表中的位[0]，请参看 5.4.1。

4.4.1 Process Creation

该操作会创建一个进程，并将其权能放入某个已存在的权能表。新创建的进程会引用权能表权能和页目录权能，一旦使用某对权能表/页目录权能创建了一个进程，那么这对权能在该进程被删除前就不能被移除/删除。创建进程操作需要如下几个参数：

Table 4-10 Parameters Needed for Process Creation

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_PROC_CRT。
Cap_Captbl_Crt	cid_t	C	一个对应于必须拥有 RME_CAPTBL_FLAG_CRT 属性的权能表权能的权能号，该权能号对应的权能指向要接受此新创建的进程权能的权能表。该权能号可以是一级或者二级查找编码。
Cap_Kmem	cid_t	P1.D1	一个内核内存权能号，其标识的内核内存范围必须能放下整个进程对象，并且要拥有 RME_KMEM_FLAG_PROC 属性。该权能号可以是一级或二级查找编码。
Cap_Proc	cid_t	P1.D0	一个对应于接受该新创建的进程权能的权能表的某位置的权能号。该权能号对应的权能必须是空白的。该权能号只能是一级查找编码。
Cap_Captbl	cid_t	P2.D1	一个对应于必须拥有 RME_CAPTBL_FLAG_PROC_CRT 属性的权能表权能的权能号，该权能号对应的权能指向要给新创建的进程使用的权能表。该权能号可以是一级或者二级查找编码。
Cap_Pgtbl	cid_t	P2.D0	一个对应于必须拥有 RME_PGTBL_FLAG_PROC_CRT 属性的页表权能的权能号，该权能号对应的权能指向要给新创建的进程使用的页表[2]。该权能号可以是一级或者二级查找编码。
Vaddr	ptr_t	P3	新创建的进程内核对象要使用的内核空间起始虚拟地址。

[1] 顶层页目录

[2] 顶层页目录

该操作的返回值可能如下：

Table 4-11 Possible Return Values for Process Creation

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Captbl_Crt 的一级/二级查找超出了范围。
	Cap_Kmem 的一级/二级查找超出了范围。
	Cap_Captbl 的一级/二级查找超出了范围。
	Cap_Pgtbl 的一级/二级查找超出了范围。
	Cap_Proc 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl_Crt 的一级/二级查找的权能已经被冻结。
	Cap_Kmem 的一级/二级查找的权能已经被冻结。
	Cap_Captbl 的一级/二级查找权能已经被冻结。
	Cap_Pgtbl 的一级/二级查找权能已经被冻结。
	Cap_Proc 被冻结，或者其它核正在该处创建权能。
RME_ERR_CAP_TYPE	Cap_Captbl_Crt 或 Cap_Captbl 不是权能表权能。
	Cap_Kmem 不是内核内存权能。
	Cap_Pgtbl 不是页表权能。
RME_ERR_CAP_FLAG	Cap_Captbl_Crt 无 RME_CAPTBL_FLAG_CRT 属性。
	Cap_Kmem 无 RME_KMEM_FLAG_PROC 属性，或范围错误。
	Cap_Captbl 无 RME_CAPTBL_FLAG_PROC_CRT 属性。
	Cap_Pgtbl 无 RME_PGTBL_FLAG_PROC_CRT 属性。
RME_ERR_CAP_EXIST	Cap_Proc 不是空白权能。
RME_ERR_CAP_KOTBL	分配内核内存失败。
RME_ERR_CAP_REFCNT	Cap_Captbl 或 Cap_Pgtbl 的引用计数超过了系统允许的最大范围。

4.4.2 Process Deletion

该操作会删除一个进程。被删除的进程必须不含有线程或线程迁移调用（关于后者见同步通信机制）。删除进程需要以下几个参数：

Table 4-12 Parameters Needed for Process Deletion

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_PROC_DEL。
Cap_Captbl	cid_t	C	一个对应于必须拥有 RME_CAPTBL_FLAG_DEL 属性的权能表权能的权能

参数名称	类型	域	描述
			号，该权能号对应的权能指向含有正被删除的进程权能的权能表。该权能号可以是一级或者二级查找编码。
Cap_Proc	cid_t	P1	一个对应于将被删除的进程权能的权能号。该权能号对应的权能必须是一个进程权能。该权能号只能是一级查找编码。

该操作的返回值可能如下：

Table 4-13 Possible Return Values for Process Deletion

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Captbl 的一级/二级查找超出了范围。
	Cap_Proc 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl 的一级/二级查找的权能已经被冻结。
	Cap_Proc 未被冻结。
RME_ERR_CAP_TYPE	Cap_Captbl 不是权能表权能。
	Cap_Proc 不是进程权能。
RME_ERR_CAP_NULL	Cap_Proc 为空白权能。
	两个核同时试图删除该进程，此时未成功的核返回该值。
RME_ERR_CAP_FLAG	Cap_Captbl 无 RME_CAPTBL_FLAG_DEL 属性。
RME_ERR_CAP_QUIE	Cap_Proc 不安定。
RME_ERR_CAP_REFCNT	Cap_Proc 的引用计数不为 0，或者不为根权能。
RME_ERR_PTH_REFCNT	该进程的引用计数不为 0[1]。

4.4.3 Changing Process Capability Table

该操作会把进程的权能表替换成另外一个。替换完成后，立即生效。替换进程的权能表需要以下几个参数：

Table 4-14 Parameters Needed for Changing Process Capability Table

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_PROC_CPT。
Cap_Proc	cid_t	P1	一个对应于必须拥有 RME_PROC_FLAG_CPT 属性的进程权能的权能号，该权能号对应的权能指向要修改权能表的进程。该权能号可以是一级或者

[1] 内部有线程或线程迁移调用

参数名称	类型	域	描述
			二级查找编码。
Cap_Captbl	cid_t	P2	一个对应于必须拥有 RME_CAPTBL_FLAG_PROC_CPT 属性的权能表权能的权能号，该权能号对应的权能指向要给进程使用的权能表。该权能号可以是一级或者二级查找编码。

该操作的返回值可能如下：

Table 4-15 Possible Return Values for Changing Process Capability Table

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Proc 的一级/二级查找超出了范围。
	Cap_Captbl 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Proc 的一级/二级查找的权能已经被冻结。
	Cap_Captbl 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Proc 不是进程权能。
	Cap_Captbl 不是权能表权能。
RME_ERR_CAP_NULL	Cap_Proc 或 Cap_Captbl 为空白权能。
RME_ERR_CAP_FLAG	Cap_Proc 无 RME_PROC_FLAG_CPT 属性。
	Cap_Captbl 无 RME_CAPTBL_FLAG_PROC_CPT 属性。
RME_ERR_CAP_REFCNT	Cap_Captbl 的引用计数超过了系统允许的最大范围。
RME_ERR_PTH_CONFLICT	两个核同时试图替换权能表，此时未成功的核返回该值。

4.4.4 Changing Process Page Table

该操作会把进程的页表替换成另外一个。替换完成后，立即生效。替换进程的页表需要以下几个参数：

Table 4-16 Parameters Needed for Changing Process Page Table

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_PROC_PGT。
Cap_Proc	cid_t	P1	一个对应于必须拥有 RME_PROC_FLAG_PGT 属性的进程权能的权能号，该权能号对应的权能指向要修改页表[1]的进程。该权能号可以是一级或者二级查找编码。

[1] 顶层页目录

参数名称	类型	域	描述
Cap_Pgtbl	cid_t	P2	一个对应于必须拥有 RME_PGTBL_FLAG_PROC_PGT 属性的页目录权能的权能号，该权能号对应的权能指向要给进程使用的页表[1]。该权能号可以是一级或者二级查找编码。

该操作的返回值可能如下：

Table 4-17 Possible Return Values for Changing Process Page Table

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Proc 的一级/二级查找超出了范围。
	Cap_Pgtbl 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Proc 的一级/二级查找的权能已经被冻结。
	Cap_Pgtbl 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Proc 不是进程权能。
	Cap_Pgtbl 不是页表[2]权能。
RME_ERR_CAP_NULL	Cap_Proc 或 Cap_Pgtbl 为空白权能。
RME_ERR_CAP_FLAG	Cap_Proc 无 RME_PROC_FLAG_PGT 属性。
	Cap_Pgtbl 无 RME_PGTBL_FLAG_PROC_PGT 属性。
RME_ERR_CAP_REFCNT	Cap_Pgtbl 的引用计数超过了系统允许的最大范围。
RME_ERR_PTH_CONFLICT	两个核同时试图替换页表，此时未成功的核返回该值。

4.5 Thread System Calls

与线程有关的内核功能如下：

Table 4-18 Thread Related System Calls

调用号	类型	用途
RME_SVC_THD_CRT	系统调用	创建线程
RME_SVC_THD_DEL	系统调用	删除线程
RME_SVC_THD_EXEC_SET	系统调用	设置线程的执行属性[3]

[1] 顶层页目录

[2] 顶层页目录

[3] 入口和栈

调用号	类型	用途
RME_SVC_THD_HYP_SET	系统调用	设置线程的虚拟机属性[1]
RME_SVC_THD_SCHED_BIND	系统调用	将线程绑定到某 CPU
RME_SVC_THD_SCHED_RCV	系统调用	收取线程的调度器事件
RME_SVC_THD_SCHED_PRIO	系统调用	更改线程的优先级
RME_SVC_THD_SCHED_FREE	系统调用	将线程从某 CPU 上释放
RME_SVC_THD_TIME_XFER	系统调用	在线程间传递时间片
RME_SVC_THD_SWT	系统调用	切换到某同优先级线程

线程权能的操作标志如下：

Table 4-19 Thread Operation Flags

标志	位	用途
RME_THD_FLAG_EXEC_SET	[0]	允许设置该线程的执行属性。
RME_THD_FLAG_HYP_SET	[1]	允许设置该线程的虚拟机属性。
RME_THD_FLAG_SCHED_CHILD	[2]	允许在该线程在绑定操作中作为子线程。
RME_THD_FLAG_SCHED_PARENT	[3]	允许在该线程在绑定操作中作为父线程。
RME_THD_FLAG_SCHED_PRIO	[4]	允许更改该线程的优先级。
RME_THD_FLAG_SCHED_FREE	[5]	允许解除该线程对某 CPU 的绑定。
RME_THD_FLAG_SCHED_RCV	[6]	允许收取该线程的调度器事件。
RME_THD_FLAG_XFER_SRC	[7]	允许该线程在时间传递中作为源。
RME_THD_FLAG_XFER_DST	[8]	允许该线程在时间传递中作为目标。
RME_THD_FLAG_SWT	[9]	允许切换操作切换到该线程。

4.5.1 Thread Creation

该操作会创建一个线程，并将其权能放入某个已存在的权能表。新创建的线程是没有绑定到任何 CPU，没有 TID，并处于 RME_THD_TIMEOUT 状态的。这个线程会引用创建时指定的进程，一旦使用在某进程内创建了一个线程，那么这个进程在该线程被删除前就不能被删除。创建线程操作需要如下几个参数：

Table 4-20 Parameters Needed for Thread Creation

[1] 寄存器保存位置

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_THD_CRT。
Cap_Captbl	cid_t	C	一个对应于必须拥有 RME_CAPTBL_FLAG_CRT 属性的权能表权能的权能号，该权能号对应的权能指向要接受此新创建的线程权能的权能表。该权能号可以是一级或者二级查找编码。
Cap_Kmem	cid_t	P1.D1	一个内核内存权能号，其标识的内核内存范围必须能够放下线程内核对象，并且要拥有 RME_KMEM_FLAG_THD 属性。该权能号可以是一级或二级查找编码。
Cap_Thd	cid_t	P1.D0	一个对应于接受该新创建的线程权能的权能表的某位置的权能号。该权能号对应的权能必须是空白的。该权能号只能是一级查找编码。
Cap_Proc	cid_t	P2.D1	一个对应于必须拥有 RME_PROC_FLAG_THD 属性的进程权能的权能号，该权能号对应的权能指向包含新创建的线程的进程。该权能号可以是一级或者二级查找编码。
Max_Prio	ptr_t	P2.D0	该线程的优先级上限。
Vaddr	ptr_t	P3	新创建的线程内核对象要使用的内核空间起始虚拟地址。

该操作的返回值可能如下：

Table 4-21 Possible Return Values for Thread Creation

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Captbl 的一级/二级查找超出了范围。
	Cap_Kmem 的一级/二级查找超出了范围。
	Cap_Proc 的一级/二级查找超出了范围。
	Cap_Thd 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl 的一级/二级查找的权能已经被冻结。
	Cap_Kmem 的一级/二级查找的权能已经被冻结。
	Cap_Proc 的一级/二级查找权能已经被冻结。
	Cap_Thd 被冻结，或者其它核正在该处创建权能。
RME_ERR_CAP_TYPE	Cap_Captbl 不是权能表权能。
	Cap_Kmem 不是内核内存权能。
	Cap_Proc 不是进程权能。
RME_ERR_CAP_FLAG	Cap_Captbl 无 RME_CAPTBL_FLAG_CRT 属性。
	Cap_Kmem 无 RME_KMEM_FLAG_THD 属性，或范围错误。

返回值	意义
	Cap_Proc 无 RME_PROC_FLAG_THD 属性。
RME_ERR_CAP_EXIST	Cap_Thd 不是空白权能。
RME_ERR_CAP_KOTBL	分配内核内存失败。
RME_ERR_PTH_PRIO	试图创建比自身拥有更高的优先级上限的线程，或者传入的优先级上限过大，超过了系统配置时允许的上限。

4.5.2 Thread Deletion

该操作会删除一个线程。被删除的线程必须已经被解除绑定。删除线程需要以下几个参数：

Table 4-22 Parameters Needed for Thread Deletion

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_THD_DEL。
Cap_Captbl	cid_t	C	一个对应于必须拥有 RME_CAPTBL_FLAG_DEL 属性的权能表权能的权能号，该权能号对应的权能指向含有正被删除的线程权能的权能表。该权能号可以是一级或者二级查找编码。
Cap_Thd	cid_t	P1	一个对应于将被删除的线程权能的权能号。该权能号对应的权能必须是一个线程权能。该权能号只能是一级查找编码。

该操作的返回值可能如下：

Table 4-23 Possible Return Values for Thread Deletion

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Captbl 的一级/二级查找超出了范围。
	Cap_Thd 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl 的一级/二级查找的权能已经被冻结。
	Cap_Thd 未被冻结。
RME_ERR_CAP_TYPE	Cap_Captbl 不是权能表权能。
	Cap_Thd 不是线程权能。
RME_ERR_CAP_NULL	Cap_Thd 为空白权能。
	两个核同时试图删除该线程，此时未成功的核返回该值。
RME_ERR_CAP_FLAG	Cap_Captbl 无 RME_CAPTBL_FLAG_DEL 属性。
RME_ERR_CAP_QUIE	Cap_Thd 不安定。

返回值	意义
RME_ERR_CAP_REFCNT	Cap_Thd 的引用计数不为 0，或者不为根权能。
RME_ERR_PTH_INVSTATE	该线程仍然处于被绑定状态。

4.5.3 Setting Thread Execution Property

该操作会设置线程的执行属性，也即其入口，栈和参数。被设置的线程必须已经被绑定于某个 CPU，而且设置执行属性必须在这个 CPU 上进行。对于一个已经处于 RME_THD_FAULT 状态的线程，设置线程执行属性会将其置为 RME_THD_TIMEOUT 状态。当传入的 Entry 和 Stack 均为 0（NULL）这个特殊值时，该线程的执行属性不变，仅仅会更改其状态。这在错误处理中是很有用的。设置线程执行属性需要以下几个参数：

Table 4-24 Parameters Needed for Setting Thread Execution Property

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_THD_EXEC_SET。
Cap_Thd	cid_t	C	一个对应于必须拥有 RME_THD_FLAG_EXEC_SET 属性的线程权能的权能号，该权能号对应的权能指向被设置执行属性的线程。该权能号可以是一级或者二级查找编码。
Entry	ptr_t	P1	该线程的入口。这个值是该线程所在的进程内部的一个虚拟地址，线程将从这里开始执行。
Stack	ptr_t	P2	该线程的执行栈。这个值是该线程所在的进程内部的一个虚拟地址，线程将使用这个地址作为栈的起始。具体的栈是递增堆栈还是递减堆栈由用户态库决定。
Param	ptr_t	P3	要传递给该线程的参数。

该操作的返回值可能如下：

Table 4-25 Possible Return Values for Setting Thread Execution Property

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Thd 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Thd 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Thd 不是线程权能，或者为空白权能。
RME_ERR_CAP_FLAG	Cap_Thd 无 RME_THD_FLAG_EXEC_SET 属性。
RME_ERR_PTH_INVSTATE	线程处于未被绑定状态。

4.5.4 Setting Thread Hypervisor Property

该操作会设置线程的虚拟机属性，也即保存寄存器组的地址。在默认状态下[1]，线程的寄存器组默认保存在内核中的线程内核对象中；如果该线程被设置了虚拟机属性，那么线程的寄存器组就会被保存到被设置的地址。设置线程虚拟机属性需要以下几个参数：

Table 4-26 Parameters Needed for Setting Thread Hypervisor Property

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_THD_HYP_SET。
Cap_Thd	cid_t	P1	一个对应于必须拥有 RME_THD_FLAG_HYP_SET 属性的线程权能的权能号，该权能号对应的权能指向被设置虚拟机属性的线程。该权能号可以是一级或者二级查找编码。
Kaddr	ptr_t	P2	要保存寄存器组到的地址。这个地址必须是内核可访问的虚拟地址范围，也即大于或等于 RME_HYP_VA_START，小于 RME_HYP_VA_START+RME_HYP_SIZE，而且要字对齐。如果该值设置为 0，那么线程的虚拟机属性将被清空，也即线程恢复到默认状态，保存寄存器组到内核对象中。

该操作的返回值可能如下：

Table 4-27 Possible Return Values for Setting Thread Hypervisor Property

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Thd 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Thd 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Thd 不是线程权能，或者为空白权能。
RME_ERR_CAP_FLAG	Cap_Thd 无 RME_THD_FLAG_HYP_SET 属性。
RME_ERR_PTH_INVSTATE	线程处于未被绑定状态。
RME_ERR_PTH_PGTBL	Kaddr 不在指定的虚拟机专用虚拟内存范围内或者未对齐。

4.5.5 Binding Thread to CPU

该操作会将线程绑定到某 CPU 上。被设置的线程必须未被绑定于任何 CPU。在哪个 CPU 上调用本函数，绑定就完成在哪个 CPU 上。将线程绑定到某 CPU 需要以下几个参数：

[1] 线程刚刚创建时

Table 4-28 Parameters Needed for Binding Thread to CPU

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_THD_SCHED_BIND。
Cap_Thd	cid_t	C	一个对应于必须拥有 RME_THD_FLAG_SCHED_CHILD 属性的线程权能的权能号，该权能号对应的权能指向被绑定的线程。该权能号可以是一级或者二级查找编码。
Cap_Thd_Sched	cid_t	P1.D1	一个对应于必须拥有 RME_THD_FLAG_SCHED_PARENT 属性的线程权能的权能号，该权能号对应的权能指向被绑定的线程的父线程。该父线程必须已经被绑定于调用本函数的 CPU。该权能号可以是一级或者二级查找编码。
Cap_Sig	cid_t	P1.D0	一个对应于必须拥有 RME_SIG_FLAG_SCHED 属性的信号端点权能的权能号。当调度器事件发生[1]时，该端点会收到一个信号。该权能号可以是一级或者二级查找编码，而且也是可选的。如果你不希望接收该线程的调度器信号，那么可以传入 RME_CAPID_NULL[2]。
TID	tid_t	P2	用户提供的 TID。RME 本身并不检查系统中是否有两个线程拥有相同的 ID；至于是否要做这种检查是用户态的事情。此外，线程 ID 不能超过 2^{N-2}-1，其中 N 是用位计算的处理器字长；它也不能是负值。
Prio	ptr_t	P3	被绑定的线程的抢占优先级。在 RME 中线程的优先级从 0 开始计算，值越大优先级越高。这个值不能超过该线程固有的优先级上限。

该操作的返回值可能如下：

Table 4-29 Possible Return Values for Binding Thread to CPU

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Thd 或 Cap_Thd_Sched 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Thd 或 Cap_Thd_Sched 的一级/二级查找的权能已经被冻结。

[1] 线程超时或错误

[2] 一个负值

返回值	意义
RME_ERR_CAP_TYPE	Cap_Thd 或 Cap_Thd_Sched 不是线程权能，或者为空白权能。
RME_ERR_CAP_FLAG	Cap_Thd 无 RME_THD_FLAG_SCHED_CHILD 属性。
	Cap_Thd_Sched 无 RME_THD_FLAG_SCHED_PARENT 属性。
	当传入 Cap_Sig 时，它没有 RME_SIG_FLAG_SCHED 属性。
RME_ERR_PTH_TID	传入的 TID 不合法。
RME_ERR_PTH_NOTIF	试图注册自己为自己的父线程，不合法。
RME_ERR_PTH_PRIO	抢占优先级超过了该线程固有的优先级上限，不合法。
RME_ERR_PTH_INVSTATE	Cap_Thd 处于被绑定状态或 Cap_Thd_Sched 处于未被绑定状态。
	Cap_Thd_Sched 被绑定到了和调用本函数的 CPU 不同的 CPU。
RME_ERR_PTH_CONFLICT	如果两个 CPU 同时尝试绑定，在失败的 CPU 上返回该值。

4.5.6 Changing Thread Priority

该操作会更改一个已经绑定到某 CPU 的线程的优先级。更改优先级的操作必须在同一个 CPU 上进行。更改线程优先级需要以下几个参数：

Table 4-30 Parameters Needed for Changing Thread Priority

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_THD_SCHED_PRIO。
Cap_Thd	cid_t	P1	一个对应于必须拥有 RME_THD_FLAG_SCHED_PRIO 属性的线程权能的权能号，该权能号对应的权能指向要修改抢占优先级的线程。该权能号可以是一级或者二级查找编码。
Prio	ptr_t	P2	线程的新的抢占优先级。在 RME 中线程的优先级从 0 开始计算，值越大优先级越高。这个值不能超过该线程固有的优先级上限。

该操作的返回值可能如下：

Table 4-31 Possible Return Values for Changing Thread Priority

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Thd 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Thd 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Thd 不是线程权能，或者为空白权能。

返回值	意义
RME_ERR_CAP_FLAG	Cap_Thd 无 RME_THD_FLAG_SCHED_PRIO 属性。
RME_ERR_PTH_PRIO	抢占优先级超过了该线程固有的优先级上限，不合法。
RME_ERR_PTH_INVSTATE	Cap_Thd 处于未被绑定状态。
	Cap_Thd 被绑定到了和调用本函数的 CPU 不同的 CPU。

4.5.7 Freeing Thread from CPU

该操作会解除线程对某个 CPU 的绑定。被解除绑定的线程自身不能有子线程。如果被解除绑定的线程向其父线程发送了调度器事件，那么这个事件会被撤销。如果被解除绑定的线程阻塞，那么它同时会让这次阻塞接收强制结束并返回 RME_ERR_SIV_FREE。解除线程对某 CPU 的绑定需要以下几个参数：

Table 4-32 Parameters Needed for Freeing Thread from CPU

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_THD_SCHED_FREE。
Cap_Thd	cid_t	P1	一个对应于必须拥有 RME_THD_FLAG_SCHED_FREE 属性的线程权能的权能号，该权能号对应的权能指向要解除绑定的线程。该权能号可以是一级或者二级查找编码。

该操作的返回值可能如下：

Table 4-33 Possible Return Values for Freeing Thread from CPU

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Thd 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Thd 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Thd 不是线程权能，或者为空白权能。
RME_ERR_CAP_FLAG	Cap_Thd 无 RME_THD_FLAG_SCHED_FREE 属性。
RME_ERR_PTH_REFCNT	Cap_Thd 仍然是某些线程的父线程[1]。
RME_ERR_PTH_INVSTATE	Cap_Thd 处于未被绑定状态。
	Cap_Thd 被绑定到了和调用本函数的 CPU 不同的 CPU。

4.5.8 Receiving Thread Scheduler Events

[1] 调度器线程

该操作会接收一个线程的调度器事件。该操作不会阻塞，如果该线程暂时没有调度器事件，那么我们返回一个负值。返回的值如果为正，那么由错误标识符[1]和线程标识符[2]两部分组成。接收一个线程的调度器事件需要如下参数：

Table 4-34 Parameters Needed for Receiving Thread Scheduler Events

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_THD_SCHED_RCV。
Cap_Thd	cid_t	P1	一个对应于必须拥有 RME_THD_FLAG_SCHED_RCV 属性的线程权能的权能号，我们试图接收该线程的子线程发来的调度器事件。该权能号可以是一级或者二级查找编码。

该操作的返回值可能如下：

Table 4-35 Possible Return Values for Receiving Thread Scheduler Events

返回值	意义
非负值	操作成功。如果错误标识符为 0，那么接收到的是线程超时事件，表明线程标识符对应的线程耗尽了时间片而停止执行。如果错误标识符为 1，那么表明线程标识符对应的线程在执行中发生了一个错误，被迫停止执行。
RME_ERR_CAP_RANGE	Cap_Thd 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Thd 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Thd 不是线程权能，或者为空白权能。
RME_ERR_CAP_FLAG	Cap_Thd 无 RME_THD_FLAG_SCHED_RCV 属性。
RME_ERR_PTH_NOTIF	Cap_Thd 暂无调度器事件可以接收。
RME_ERR_PTH_INVSTATE	Cap_Thd 处于未被绑定状态。
	Cap_Thd 被绑定到了和调用本函数的 CPU 不同的 CPU。

4.5.9 Transferring Execution Timeslices

该操作被用来在线程之间传递时间片。时间片的传递者[3]和接收者[4]必须位于同一个 CPU 上，而且该函数必须从这个 CPU 上被调用，以保证传递的时间是该 CPU 上的执行时间。传递的时间片必须不等

[1] 处于倒数第二个二进制位

[2] 其他后续二进制位

[3] 源线程

[4] 目标线程

于 0，而且目标线程不能处于因错误而停止执行（RME_THD_FAULT）的状态。传递运行时间片需要以下几个参数：

Table 4-36 Parameters Needed for Transferring Execution Timeslices

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_THD_TIME_XFER。
Cap_Thd_Dst	cid_t	P1	一个对应于必须拥有 RME_THD_FLAG_XFER_DST 属性的线程权能的权能号，该权能号对应的权能指向目标线程。该权能号可以是一级或者二级查找编码。
Cap_Thd_Src	cid_t	P2	一个对应于必须拥有 RME_THD_FLAG_XFER_SRC 属性的线程权能的权能号，该权能号对应的权能指向源线程。该权能号可以是一级或者二级查找编码。
Time	ptr_t	P3	要传递的时间片数量。这个值的单位是时间片，单个时间片的大小在系统编译时被决定。该值不能为 0。 传入 RME_THD_INF_TIME 进行无限传递。 传入 RME_THD_INIT_TIME 进行回收传递。 传入其它非 0 值进行普通传递。

该操作的返回值可能如下：

Table 4-37 Possible Return Values for Transferring Execution Timeslices

返回值	意义
非负值	操作成功，返回的是目标线程现有的时间片数。如果是进行无限传递和回收传递且目标线程变成了无限线程，那么返回 RME_THD_MAX_TIME。
RME_ERR_CAP_RANGE	Cap_Thd_Dst 或 Cap_Thd_Src 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Thd_Dst 或 Cap_Thd_Src 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Thd_Dst 或 Cap_Thd_Src 不是线程权能，或者为空白权能。
RME_ERR_CAP_FLAG	Cap_Thd_Dst 无 RME_THD_FLAG_XFER_DST 属性。 Cap_Thd_Src 无 RME_THD_FLAG_XFER_SRC 属性。
RME_ERR_PTH_FAULT	Cap_Thd_Dst 处于 RME_THD_FAULT 状态。
RME_ERR_PTH_INVSTATE	Cap_Thd_Dst 或 Cap_Thd_Src 处于未被绑定状态。 Cap_Thd_Dst 或 Cap_Thd_Src 被绑定到了和调用本函数的 CPU

返回值	意义
	不同的 CPU。
RME_ERR_PTH_OVERFLOW	接收该时间片的线程的时间片已满[1]。这个错误是很罕见的，因为一般传递的时间片达不到该值。

4.5.10 Switching to Thread

该操作允许用户态调度器直接切换到某个绑定于同一 CPU 上的线程，方便用户态调度的实现。被切换到的线程必须和当前线程的优先级相同，而且必须处于就绪（RME_THD_READY）状态。要进行线程切换，需要如下参数：

Table 4-38 Parameters Needed for Switching to Thread

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_THD_SWT。
Cap_Thd	cid_t	P1	一个对应于必须拥有 RME_THD_FLAG_SWT 属性的线程权能的权能号，该权能号对应的权能指向要切换到的线程。该权能号可以是一级或者二级查找编码。如果不想指定要切换的线程，而要内核决定，那么可以传入 RME_CAPID_NULL[2]。
Full_Yield	ptr_t	P2	此次线程切换是否放弃当前线程的全部时间片。如果该项不为 0，那么此次切换会放弃当前线程的全部时间片。如果是在 Init 线程中调用，那么该项无效，因为 Init 线程的时间片是无限的。

该操作的返回值可能如下：

Table 4-39 Possible Return Values for Switching to Thread

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Thd 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Thd 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Thd 不是线程权能，或者为空白权能。
RME_ERR_CAP_FLAG	Cap_Thd 无 RME_THD_FLAG_THD_SWT 属性。
RME_ERR_PTH_FAULT	Cap_Thd 处于 RME_THD_FAULT 状态。
RME_ERR_PTH_INVSTATE	Cap_Thd 处于未被绑定状态。

[1] 再接收的话就会超过或等于系统允许的最大值 RME_THD_MAX_TIME

[2] 一个负值

返回值	意义
	Cap_Thd 被绑定到了和调用本函数的 CPU 不同的 CPU。
	Cap_Thd 处于阻塞（RME_THD_BLOCKED）状态。
	Cap_Thd 处于超时（RME_THD_TIMEOUT）状态。
RME_ERR_PTH_PRIO	Cap_Thd 的优先级和当前线程的优先级不同。

4.6 Bibliography

[1] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A kernel for scalable predictability," in Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE, 2015, pp. 121-132.

[2] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, et al., "seL4: formal verification of an OS kernel," presented at the Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, Big Sky, Montana, USA, 2009.

[3] P. Gadepalli, R. Gifford, L. Baier, M. Kelly, and G. Parmer, "Temporal capabilities: access control for time" in Real-Time Systems Symposium (RTSS), 2018 IEEE, 2018.

[4] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On, 2015, pp. 171-172.

[5] U. Steinberg, A. Böttcher, and B. Kauer, "Timeslice donation in component-based systems," in Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, 2010, pp. 16-23.

Chapter 5 Synchronous and Asynchronous Communication

5.1 Introduction to Synchronous and Asynchronous Communication

5.1.1 Introduction to Synchronous Communication

操作系统中的同步（Synchronous）通信机制是用来同步两个并行或并发进程的一种机制。在传统操作系统中它通常表现为管道、信号量等机制。同步通信机制的最大特点为其阻塞性，也即一方发送后，另一方不接收，发送方即阻塞，直到接收方接收之后，发送方的阻塞才解除。如果接收方先接收，那么它也阻塞，直到发送方发送之后，接收方的阻塞才解除。采用这种传统实现的同步通信需要两个线程才能在进程之间通信。

在 RME 中，同步通信模型被进一步简化为线程迁移（Thread Migration）调用[1]，其允许一个线程能够进入另一个进程内部执行一段代码后，再返回属于自己的进程继续执行。这是最高效的进程间通信（Inter-Process Communication，IPC）实现之一。由于这种调用允许一个执行流跨越进程边界，因此只能在两个进程相当相互信任的情况下才能使用。其表观结果好像是一个进程内部运行的线程直接调用了另一个进程内部的函数，这样就只需要一个线程，也能够进行进程间通信。如果这个迁移调用要用到多余的参数，那么这些参数由共享内存、寄存器组或协处理器寄存器组传递。线程迁移调用是可以嵌套的，而且嵌套的层数可以是无限的。一个线程迁移的例子如下所示。该线程进行了两次迁移，并且在三个进程之间交替执行。图中的红色正三角代表迁移调用激活，红色倒三角代表迁移调用返回。在进程 2 和进程 3 中，线程执行流上的方框标注着线程的迁移调用栈，黑色三角指向的则是迁移调用栈的栈顶。在返回时，总是先返回到栈顶所指的进程内。

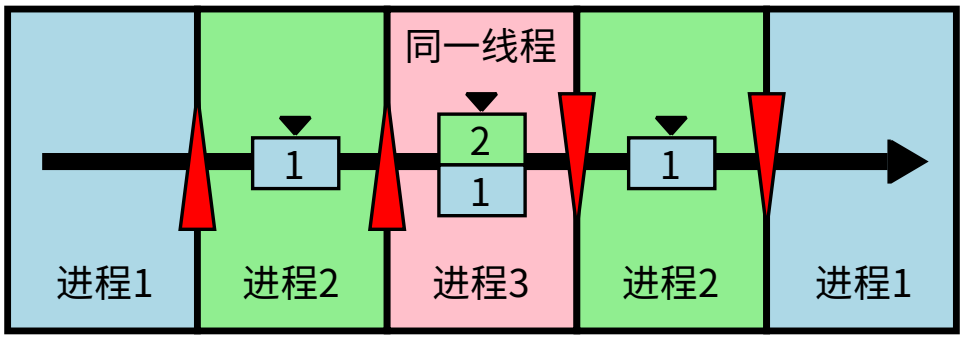

Figure 5-1 Synchronous Invocation Example Usecase

线程迁移调用的设计参考了 Composite[1]和 Mach 3[2]。

5.1.2 Introduction to Asynchronous Communication

操作系统中的异步（Asynchronous）通信机制是用来协调操作系统中的生产者-消费者（Producer-Consumer）问题的。通常而言这样的机制有邮箱（Mailbox）、消息队列（Message Queue）等。异步通信机制的最大特点为其非阻塞性，也即一方发出后，另一方不接收，信息会先被缓存，然后

[1] 或说是本地过程调用

发送者将直接返回。接收者在接收时，如果有信息，那么接收者返回；如果没有信息，那么接收者视系统调用的情况，可以直接返回，可以阻塞，也可以阻塞一段时间直到超时返回。

在 RME 中，异步通信模型被进一步简化为信号端点（Signal Endpoint），其主要机制是使发送者可以向信号端点发送，使接收者可以从信号端点接收。发送总是不阻塞的，接收在有信号之时也是不阻塞的，接收在没有信号之时则可以阻塞也可以不阻塞。信号只携带数目信息[1]，不携带任何其他信息。如果需要传递其他信息，那么需要在用户态通过共享内存（Shared Memory）完成对这些信息的传递。一个异步通信的例子如下所示。其中优先级较低的线程 1 给优先级较高的线程 2 发送了一个异步信号（红色闪电标志），线程 2 在完成处理后又在端点上阻塞（红色长方形标志），使线程 1 继续执行。

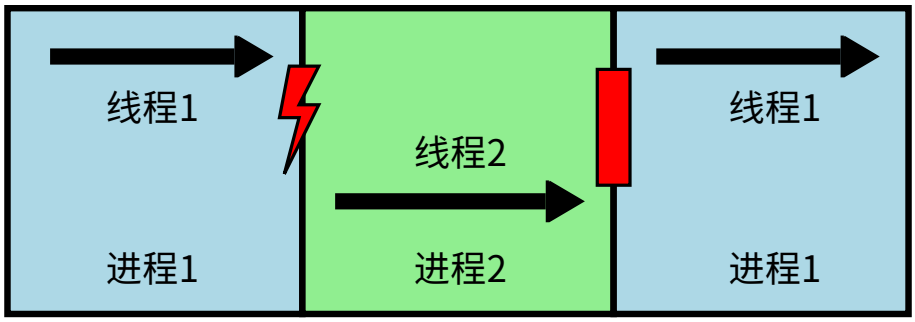

Figure 5-2 Asynchronous Signal Example Usecase

5.2 Synchronous Communication Operations

要使用同步通信功能，那么需要先创建一个线程迁移调用。在创建线程迁移调用时，需要指明进程。在创建后，需要指定这个进程内部调用的入口和给该线程使用的栈。接下来，进行线程迁移调用，线程从一个线程内部过渡到另外一个进程之内执行。如果在这个线程迁移调用之中使用了其他的线程迁移调用，那么线程迁移调用将会进行嵌套（Nesting）。在线程迁移调用执行完毕之后，需要调用线程迁移调用返回功能，从这个迁移调用之中返回。和异步通信相比，同步通信操作具有即时性，保证在调用时一定会以最快速度得到响应，而且在进行调用时线程仅仅是跨到另外一个保护域中执行而不改变其其他属性。需要注意的是，线程迁移调用不会保存任何通用寄存器或者协处理器寄存器，也不会更改协处理器当前的状态；它只会更改那些改变程序执行流必要的寄存器，如堆栈指针和程序计数器等。通用寄存器的值和协处理器状态将会被直接带到线程迁移入口和出口。如果有需要保存的寄存器或者协处理器状态，那么需要用户态自行完成。此外，协处理器寄存器组也可以用来传递额外的参数或者接收额外的返回值。

线程迁移调用不会被绑定到 CPU。如果有多个 CPU 上的线程试图同时激活一个线程迁移调用，那么只有一个 CPU 上的线程会成功，其他的线程都会返回错误码；如果在一个迁移调用被激活时试图再次激活它[2]，那么也会返回错误码。

如果线程迁移调用中发生了嵌套，那么返回时也需要逐级返回。如果线程在进行迁移调用的过程中被解除绑定，那么它仍然处于迁移调用状态，被绑定到新的 CPU 后会继续从那里执行；如果线程在进行

[1] 它只有一个计数器

[2] 不管是在同一个 CPU 上还是在不同的 CPU 上

迁移调用的过程中发生了一个错误，线程的行为是由设置迁移调用执行属性时传入的 Fault_Ret_Flag 参数决定的。如果该参数不为 0，那么它并不会进入 RME_THD_FAULT 状态，而是会直接从这个迁移调用中返回到上一级，并且返回值是指示发生了错误的错误码。如果该参数为 0，那么该线程会进入 RME_THD_FAULT 状态等待错误修复。

5.3 Asynchronous Communication Operations

要使用异步通信功能，那么需要先创建一个信号端点，然后使发送方向其发送信号，接收方从该端点接收即可。接收时，如果该端点没有信号，那么会阻塞；接收如果成功，那么返回值将会是顺利返回时的剩余信号数。

信号端点不会被绑定到 CPU。如果有多个 CPU 上的线程同时试图阻塞在一个端点，那么只有一个会成功，其他的线程都会返回一个错误值。如果在一个线程已经阻塞在一个端点时，其他线程也试图阻塞[1]，那么其他线程的阻塞会失败。在实践中不推荐一个信号端点多接收者的使用方法，因为这往往需要相当复杂的协调。

需要注意的是，当一个线程已经阻塞在某个信号端点时，只有和被阻塞线程同一个 CPU 上的发送操作才能解除该线程的阻塞。其他 CPU 上的线程虽然也能向该端点发送，但是只能增加信号计数而无法解除阻塞。

RME 不允许初始（Init）线程接收任何信号。因为接收信号是潜在的阻塞操作，而 Init 线程一旦阻塞，RME 就无法保证能够在 CPU 空闲时找到一个线程来运行。如果既需要接收信号，又需要接收线程的时间片是无限的，那么可以新建一个线程，然后从 Init 向它进行无限传递，将其变成无限线程。

为了使应用程序的编写更加灵活，RME 允许四种不同的端点信号接收方法，分别如下：

Table 5-1 Four Receiving Options of Endpoint Signals

接收选项	意义
RME_RCV_BS	以阻塞方式试图接收单个信号。有可能造成线程本身阻塞。
RME_RCV_BM	以阻塞方式试图接收多个信号。有可能造成线程本身阻塞，并且如果端点上有信号，会获取端点上全部的信号。
RME_RCV_NS	以非阻塞方式试图接收单个信号。该方法在端点上无信号时会立即返回。
RME_RCV_NM	以非阻塞方式试图接收多个信号。该方法在端点上无信号时会立即返回，并且如果端点上有信号，会获取端点上全部的信号。

5.4 Synchronous Invocation System Calls

与线程迁移调用有关的内核功能如下：

[1] 不管是在同一个 CPU 上还是在不同的 CPU 上

Table 5-2 Synchronous Invocation Related System Calls

调用号	类型	用途
RME_SVC_INV_CRT	系统调用	创建线程迁移调用
RME_SVC_INV_DEL	系统调用	删除线程迁移调用
RME_SVC_INV_SET	系统调用	设置线程迁移调用的执行属性[1]
RME_SVC_INV_ACT	系统调用	激活[2]线程迁移调用
RME_SVC_INV_RET	系统调用	从一个线程迁移调用返回

线程迁移调用权能的操作标志如下：

Table 5-3 Thread Invocation Port Operation Flags

标志	位	用途
RME_INV_FLAG_SET	[0]	允许设置该线程迁移调用的执行属性。
RME_INV_FLAG_ACT	[1]	允许激活该线程迁移调用。

由于线程迁移调用返回操作是不需要线程迁移调用权能的，因此也没有该操作标志位[3]。

5.4.1 Synchronous Invocation Port Creation

该操作会创建一个线程迁移调用，并将其权能放入某个已存在的权能表。新创建的线程迁移调用会引用创建时指定的进程，一旦在某进程内创建了一个线程迁移调用，那么这个进程在该线程迁移调用被删除前就不能被删除。创建线程迁移调用需要如下几个参数：

Table 5-4 Parameters Needed for Synchronous Invocation Port Creation

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_INV_CRT。
Cap_Captbl	cid_t	C	一个对应于必须拥有 RME_CAPTBL_FLAG_CRT 属性的权能表权能的权能号，该权能号对应的权能指向要接受此新创建的线程迁移调用权能的权能表。该权能号可以是一级或者二级查找编码。
Cap_Kmem	cid_t	P1.D1	一个内核内存权能号，其标识的内核内存范围必须能放下线程迁移调用对象，并且要有 RME_KMEM_FLAG_INV 属性。该权能号可以是一级或二级查找编码。

[1] 入口和栈

[2] 进行或调用

[3] 详见 5.4.5

参数名称	类型	域	描述
Cap_Inv	cid_t	P1.D0	一个对应于接受该新创建的线程迁移调用权能的权能表的某位置的权能号。该权能号对应的权能必须是空白的。该权能号只能是一级查找编码。
Cap_Proc	cid_t	P2	一个对应于必须拥有 RME_PROC_FLAG_INV 属性的进程权能的权能号，该权能号对应的权能指向包含新创建的线程迁移调用的进程。该权能号可以是一级或者二级查找编码。
Vaddr	ptr_t	P3	新创建的线程迁移调用要使用的内核空间起始虚拟地址。

该操作的返回值可能如下：

Table 5-5 Possible Return Values for Synchronous Invocation Port Creation

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Captbl 的一级/二级查找超出了范围。
	Cap_Kmem 的一级/二级查找超出了范围。
	Cap_Proc 的一级/二级查找超出了范围。
	Cap_Inv 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl 的一级/二级查找的权能已经被冻结。
	Cap_Kmem 的一级/二级查找的权能已经被冻结。
	Cap_Proc 的一级/二级查找权能已经被冻结。
	Cap_Inv 被冻结，或者其它核正在该处创建权能。
RME_ERR_CAP_TYPE	Cap_Captbl 不是权能表权能。
	Cap_Kmem 不是内核内存权能。
	Cap_Proc 不是进程权能。
RME_ERR_CAP_FLAG	Cap_Captbl 无 RME_CAPTBL_FLAG_CRT 属性。
	Cap_Kmem 无 RME_KMEM_FLAG_INV 属性，或范围错误。
	Cap_Proc 无 RME_PROC_FLAG_INV 属性。
RME_ERR_CAP_EXIST	Cap_Inv 不是空白权能。
RME_ERR_CAP_KOTBL	分配内核内存失败。

5.4.2 Synchronous Invocation Port Deletion

该操作会删除一个线程迁移调用。被删除的线程迁移调用必须不处于正被使用的状态。删除线程迁移调用需要如下几个参数：

Table 5-6 Parameters Needed for Synchronous Invocation Port Deletion

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_INV_DEL。
Cap_Captbl	cid_t	C	一个对应于必须拥有 RME_CAPTBL_FLAG_DEL 属性的权能表权能的权能号，该权能号对应的权能指向含有正被删除的线程迁移调用权能的权能表。该权能号可以是一级或者二级查找编码。
Cap_Inv	cid_t	P1	一个对应于将被删除的线程迁移调用权能的权能号。该权能号对应的必须是一个线程迁移调用权能。该权能号只能是一级查找编码。

该操作的返回值可能如下：

Table 5-7 Possible Return Values for Synchronous Invocation Port Deletion

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Captbl 的一级/二级查找超出了范围。
	Cap_Inv 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl 的一级/二级查找的权能已经被冻结。
	Cap_Inv 未被冻结。
RME_ERR_CAP_TYPE	Cap_Captbl 不是权能表权能。
	Cap_Inv 不是线程迁移调用权能。
RME_ERR_CAP_NULL	Cap_Inv 为空白权能。
	两个核同时试图删除该线程迁移调用，此时未成功的核返回该值。
RME_ERR_CAP_FLAG	Cap_Captbl 无 RME_CAPTBL_FLAG_DEL 属性。
RME_ERR_CAP_QUIE	Cap_Inv 不安定。
RME_ERR_CAP_REFCNT	Cap_Inv 的引用计数不为 0，或者不为根权能。
RME_ERR_SIV_ACT	该线程迁移调用仍然处于被使用状态。

5.4.3 Setting Synchronous Invocation Execution Property

该操作会设置线程迁移调用的执行属性，也即其入口和栈。我们在设置时不关心该线程迁移调用是否已经在被使用。设置线程迁移调用的执行属性需要以下几个参数：

Table 5-8 Parameters Needed for Setting Synchronous Invocation Execution Property

参数名称	类型	域	描述

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_INV_SET。
Cap_Inv	cid_t	P1.D0	一个对应于必须拥有 RME_INV_FLAG_SET 属性的线程迁移调用权能的权能号，该权能号对应的权能指向被设置执行属性的线程迁移调用。该权能号可以是一级或者二级查找编码。
Entry	ptr_t	P2	该线程迁移到用的入口。这个值是该线程迁移调用所在的进程内部的一个虚拟地址，线程迁移调用将从这里开始执行。
Stack	ptr_t	P3	该线程迁移调用的执行栈。这个值是该线程迁移调用所在的进程内部的一个虚拟地址，线程迁移调用将使用这个地址作为栈的起始。具体的栈是递增堆栈还是递减堆栈由用户态库决定。
Fault_Ret_Flag	ptr_t	P1.D1	如果不为 0，在迁移调用中一旦发生错误将会强制从迁移调用返回，不会允许错误修复。如果为 0，则该线程将进入 RME_THD_FAULT 状态等待错误修复。

该操作的返回值可能如下：

Table 5-9 Possible Return Values for Setting Synchronous Invocation Execution Property

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Inv 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Inv 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Inv 不是线程迁移调用权能，或者为空白权能。
RME_ERR_CAP_FLAG	Cap_Inv 无 RME_INV_FLAG_SET 属性。

5.4.4 Synchronous Invocation Port Activation

该操作会进行一个线程迁移调用。对应的该迁移调用必须不正在被使用。进行线程迁移调用需要以下几个参数：

Table 5-10 Parameters Needed for Synchronous Invocation Port Activation

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_INV_ACT。
Cap_Inv	cid_t	P1	一个对应于必须拥有 RME_INV_FLAG_ACT 属性的线程迁移调用权能的权能号，该权能号对应的权能指向被要被激活的线程迁移调用。该权能号可以是一级或者二级查找编码。

参数名称	类型	域	描述
Param	ptr_t	P2	要向该线程迁移调用传入的参数。

该操作的返回值可能如下：

Table 5-11 Possible Return Values for Synchronous Invocation Port Activation

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Inv 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Inv 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Inv 不是线程迁移调用权能，或者为空白权能。
RME_ERR_CAP_FLAG	Cap_Inv 无 RME_INV_FLAG_ACT 属性。
RME_ERR_SIV_ACT	Cap_Inv 已经在激活状态[1]。
	两个 CPU 试图同时进行这个调用，失败的 CPU 返回该值。

5.4.5 Synchronous Invocation Port Returning

该操作从一个线程迁移调用返回。这是一个特殊操作，它不需要除了调用号和线程迁移返回值之外的其他参数。如果有多个线程迁移调用嵌套，该函数返回到上一级线程迁移调用中。如果试图在没有线程迁移调用的情况下调用该函数，则会返回一个错误码，标志着返回未成功。从线程迁移调用返回需要如下参数：

Table 5-12 Parameters Needed for Synchronous Invocation Port Returning

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_INV_RET。

该操作的返回值可能如下：

Table 5-13 Possible Return Values for Synchronous Invocation Port Returning

返回值	意义
0	操作成功。该迁移调用立即返回，该返回值不会被调用者接收。
RME_ERR_SIV_EMPTY	试图在没有线程迁移调用的情况下调用该函数。

[1] 其他线程正在进行调用

5.5 Asynchronous Communication System Calls

与信号端点有关的内核功能如下：

Table 5-14 Asynchronous Communication Related System Calls

调用号	类型	用途
RME_SVC_SIG_CRT	系统调用	创建信号端点
RME_SVC_SIG_DEL	系统调用	删除信号端点
RME_SVC_SIG_SND	系统调用	向信号端点发送信号
RME_SVC_SIG_RCV	系统调用	从信号端点接收信号

信号端点权能的操作标志如下：

Table 5-15 Signal Endpoint Operation Flags

标志	位	用途
RME_SIG_FLAG_SND	[0]	允许向该信号端点发送。
RME_SIG_FLAG_RCV_BS	[1]	允许从该信号端点以阻塞方式接收单个信号。
RME_SIG_FLAG_RCV_BM	[2]	允许从该信号端点以阻塞方式接收多个信号。
RME_SIG_FLAG_RCV_NS	[3]	允许从该信号端点以非阻塞方式接收单个信号。
RME_SIG_FLAG_RCV_NM	[4]	允许从该信号端点以非阻塞方式接收多个信号。
RME_SIG_FLAG_SCHED	[5]	允许该端点被线程绑定用于调度器信号端点。

关于位[5]的描述，请参见 4.5.5。

5.5.1 Signal Endpoint Creation

该操作会创建一个信号端点，并将其权能放入某个已存在的权能表。创建信号端点需要如下几个参数：

Table 5-16 Parameters Needed for Signal Endpoint Creation

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_INV_CRT。
Cap_Captbl	cid_t	C	一个对应于必须拥有 RME_CAPTBL_FLAG_CRT 属性的权能表权能的权能号，该权能号对应的权能指向要接受此新创建的信号端点权能的权能表。该权能号可以是一级或者二级查找编码。

参数名称	类型	域	描述
Cap_Kmem	cid_t	P1	一个内核内存权能号，其标识的内核内存范围必须能放下信号端点对象，并且要有 RME_KMEM_FLAG_SIG 属性。该权能号可以是一级或二级查找编码。
Cap_Sig	cid_t	P2	一个对应于接受该新创建的信号端点权能的权能表的某位置的权能号。该权能号对应的权能必须是空白的。该权能号只能是一级查找编码。
Vaddr	ptr_t	P3	新创建的信号端点要使用的内核空间起始虚拟地址。

该操作的返回值可能如下：

Table 5-17 Possible Return Values for Signal Endpoint Creation

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Captbl 的一级/二级查找超出了范围。
	Cap_Kmem 的一级/二级查找超出了范围。
	Cap_Sig 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl 的一级/二级查找的权能已经被冻结。
	Cap_Kmem 的一级/二级查找的权能已经被冻结。
	Cap_Sig 被冻结，或者其它核正在该处创建权能。
RME_ERR_CAP_TYPE	Cap_Captbl 不是权能表权能。
	Cap_Kmem 不是内核内存权能。
RME_ERR_CAP_FLAG	Cap_Captbl 无 RME_CAPTBL_FLAG_CRT 属性。
	Cap_Kmem 无 RME_KMEM_FLAG_SIG 属性，或范围错误。
RME_ERR_CAP_EXIST	Cap_Sig 不是空白权能。
RME_ERR_CAP_KOTBL	分配内核内存失败。

5.5.2 Signal Endpoint Deletion

该操作会删除一个信号端点。被删除的信号端点必须不处于正被使用[1]的状态。删除信号端点需要如下几个参数：

Table 5-18 Parameters Needed for Signal Endpoint Deletion

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_SIG_DEL。

[1] 有线程阻塞在其上

参数名称	类型	域	描述
Cap_Captbl	cid_t	C	一个对应于必须拥有 RME_CAPTBL_FLAG_DEL 属性的权能表权能的权能号，该权能号对应的权能指向含有正被删除的信号端点权能的权能表。该权能号可以是一级或者二级查找编码。
Cap_Sig	cid_t	P1	一个对应于将被删除的信号端点权能的权能号。该权能号对应的必须是一个信号端点权能。该权能号只能是一级查找编码。

该操作的返回值可能如下：

Table 5-19 Possible Return Values for Signal Endpoint Deletion

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Captbl 的一级/二级查找超出了范围。
	Cap_Sig 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl 的一级/二级查找的权能已经被冻结。
	Cap_Sig 未被冻结。
RME_ERR_CAP_TYPE	Cap_Captbl 不是权能表权能。
	Cap_Sig 不是信号端点权能。
RME_ERR_CAP_NULL	Cap_Sig 为空白权能。
	两个核同时试图删除该信号端点，此时未成功的核返回该值。
RME_ERR_CAP_FLAG	Cap_Captbl 无 RME_CAPTBL_FLAG_DEL 属性。
RME_ERR_CAP_QUIE	Cap_Sig 不安定。
RME_ERR_CAP_REFCNT	Cap_Sig 的引用计数不为 0，或者不为根权能。
RME_ERR_SIV_ACT	该信号端点仍然处于被使用状态。
RME_ERR_SIV_CONFLICT	该信号端点是一个内核端点，不能被删除。具体描述见下章。

5.5.3 Sending to Signal Endpoint

该操作会向一个信号端点发送信号。当有线程在该端点上阻塞时，只有与该线程相同 CPU 上的发送才能唤醒该线程。其他 CPU 也可以向该端点发送，但仅能增加计数值而不能唤醒该线程。向一个信号端点发送信号需要如下几个参数：

Table 5-20 Parameters Needed for Sending to Signal Endpoint

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_SIG_SND。

参数名称	类型	域	描述
Cap_Sig	cid_t	P1	一个对应于必须拥有 RME_SIG_FLAG_SND 属性的信号端点权能的权能号，该权能号对应的权能指向要对其发送信号的端点。该权能号可以是一级或者二级查找编码。

该操作的返回值可能如下：

Table 5-21 Possible Return Values for Sending to Signal Endpoint

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Sig 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Sig 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Sig 不是信号端点权能，或为空白权能。
RME_ERR_CAP_FLAG	Cap_Sig 无 RME_SIG_FLAG_SND 属性。
RME_ERR_SIV_FULL	该信号端点的信号计数已满，不能再向其继续发送。这是很罕见的，因为在 32 位系统中信号计数的上限为 2^{32}-1，64 位系统中则为 2^{64}-1，依此类推。

5.5.4 Receiving from Signal Endpoint

该操作会从一个信号端点接收信号。如果该信号端点上没有信号，那么会阻塞该线程直到信号到来为止。从一个信号端点接收信号需要如下几个参数：

Table 5-22 Parameters Needed for Receiving from Signal Endpoint

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_SIG_RCV。
Cap_Sig	cid_t	P1	一个对应于必须拥有 RME_SIG_FLAG_RCV 属性的信号端点权能的权能号，该权能号对应的权能指向要从其接收信号的端点。该权能号可以是一级或者二级查找编码。
Option	ptr_t	P2	接收方式选项。可以有以下四种方法： RME_RCV_BS：阻塞并试图接收单个信号（Blocking Single）。 RME_RCV_BM：阻塞并试图接收多个信号（Blocking Multi）。 RME_RCV_NS：不阻塞并试图接收单个信号（Non-blocking Single）。 RME_RCV_NM：不阻塞并试图接收多个信号（Non-blocking Multi）。

该操作的返回值可能如下：

Table 5-23 Possible Return Values for Receiving from Signal Endpoint

返回值	意义
非负值	操作成功。该值为本次接收接收到的信号数量。
RME_ERR_CAP_RANGE	Cap_Sig 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Sig 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Sig 不是信号端点权能，或为空白权能。
RME_ERR_CAP_FLAG	Cap_Sig 无对应的接收选项需要的属性。 RME_RCV_BS 需要 RME_SIG_FLAG_RCV_BS 属性。 RME_RCV_BM 需要 RME_SIG_FLAG_RCV_BM 属性。 RME_RCV_NS 需要 RME_SIG_FLAG_RCV_NS 属性。 RME_RCV_NM 需要 RME_SIG_FLAG_RCV_NM 属性。
RME_ERR_SIV_BOOT	试图让 Init 线程从端点接收信号。
RME_ERR_SIV_ACT	已经有一个线程阻塞在该端点。
	提供的接收选项不正确。
RME_ERR_SIV_CONFLICT	两个核试图同时在一个端点上进行接收，发生了冲突，需要重试。

需要注意的是，如果选择非阻塞方式，那么当该端点上无信号时，函数会直接返回 0，代表此次接收没有收到任何东西。此外，如果接收选项是阻塞并试图接收多个信号，而且此次接收的确产生了阻塞，那么在阻塞解除时，总是以接收一个信号的方式执行。

5.6 Bibliography

[1] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A kernel for scalable predictability," in Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE, 2015, pp. 121-132.

[2] B. Ford and J. Lepreau, "Evolving mach 3.0 to a migrating thread model," presented at the Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference, San Francisco, California, 1994.

Chapter 6 Kernel Function and Kernel Asynchronous Signal

6.1 Introduction to Kernel Function

由于 RME 微内核的架构无关部分本身仅实现了对于多种处理器的通用功能，如果某种微处理器具备某种功能，而且该种功能只能在内核态进行操作的话，要利用这种功能就必须使用 RME 的内核调用机制。很多其他系统，例如 Composite[1]和 seL4[2]等，也有类似的机制。

内核功能调用（Kernel Function）机制引入了内核调用权能，它允许调用一个用户定义好的、处理器架构相关的内核函数，并且使其运行于内核态。该权能必须在启动序列[1]中创建，而且无法删除。处理器的高精度定时器、处理器间中断和低功耗运行模式的调整等等都可以通过内核调用机制进行利用，而对于没有这些功能的处理器，内核也不强迫使用者实现这些功能，以实现最大的灵活性。

每一个具体的内核功能都对应着一个内核功能号，在进行内核功能调用时需要传入。关于具体的内核调用机制的实现，请参看下个章节的描述。

6.2 Introduction to Kernel Asynchronous Signal

在 RME 中，由于中断处理函数是在用户态注册的，因此需要某种机制将这些信号传导出来。RME 使用内核异步信号端点（Kernel Asynchronous Signal Endpoint）的方式将这些中断函数导出。内核异步信号端点和普通的信号端点是一样的，其唯一的区别是创建在系统启动时完成，并且在系统运行的整个过程中不可被删除。如果需要接收内核异步信号端点上的信号，那么只要使用与普通端点同样的接收函数到该端点上接收即可。基于同样的原因，RME 没有在内核中实现定时器，而是将时钟中断传递到用户态进行处理。

6.3 Kernel Function System Calls

与内核调用机制有关的内核功能如下：

Table 6-1 Kernel Function Related System Calls

调用号	类型	用途
RME_SVC_KERN	系统调用	调用内核功能

内核调用权能的操作标志如下：

Table 6-2 Kernel Function Operation Flags

标志	类型	用途
所有位	位段	允许的内核功能范围号的范围。注意不要把内核功能号和系统调用号相混淆。在传递

[1] 见 7.8.2.9

标志	类型	用途
		内核调用权能时需要使用宏 RME_KERN_FLAG(HIGH,LOW)来填充新的内核调用权能的标志位，HIGH 为功能号的上限，LOW 为功能号的下限，[HIGH, LOW]组成的闭区间即为允许内核功能号的范围。

6.3.1 Initialization of Kernel Function

关于内核调用机制的初始创建，请参见 7.8.2.9。

6.3.2 Activating Kernel Function

该操作会执行一个内核调用函数。该操作可以携带一个子功能号，还可以带两个额外参数。激活一个内核调用需要如下几个参数：

Table 6-3 Parameters Needed for Activating Kernel Function

参数名称	类型	域	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_KERN。
Cap_Kern	cid_t	C	一个对应于内核调用权能的权能号。该权能号可以是一级或者二级查找编码。
Func_ID	ptr_t	P1.D0	内核功能号。
Sub_ID	ptr_t	P1.D1	子功能号。
Param1	ptr_t	P2	传入的第一个参数。
Param2	ptr_t	P3	传入的第二个参数。

该操作的返回值可能如下：

Table 6-4 Possible Return Values for Activating Kernel Function

返回值	意义
非负值	操作成功。返回值的意义由具体的底层实现决定。
RME_ERR_CAP_RANGE	Cap_Kern 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Kern 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Kern 不是内核调用权能，或为空白权能。
RME_ERR_CAP_FLAG	Cap_Kern 的功能号范围不允许功能号为 Func_ID 的调用。
RME_ERR_CAP_NULL	内核调用因为底层硬件不支持所传入参数而失败。该值一般不会返回，通常被返回的是用户自定义的错误码。

6.4 Kernel Endpoint Related Calls

6.4.1 Kernel Endpoint Initialization

关于内核信号端点的初始创建，请参见 7.8.2.11。

6.4.2 Sending to Kernel Endpoint

关于向内核信号端点的信号发送，请参见 7.12.2.1。

6.4.3 Receiving From Kernel Endpoint

从内核端点接收信号的方法和调用和普通信号端点是一样的，请参见 5.5.4。

6.5 Bibliography

[1] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A kernel for scalable predictability," in Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE, 2015, pp. 121-132.

[2] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, et al., "seL4: formal verification of an OS kernel," presented at the Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, Big Sky, Montana, USA, 2009.

Chapter 7 Porting RME to New Architectures

7.1 Introduction to Porting

操作系统的移植是指将一个操作系统加以修改从而使其能运行在一个新的体系架构上的工作。有时，我们也把使得能用一款新编译器编译该操作系统的工作叫做移植工作。相比较于 Linux 等系统的移植，RME 的移植是非常简单的。RME 的所有代码都用绝大程度上符合 MISRA C 规范[1]的 ANSI/ISO C89 代码写成，并包含有最小量的汇编，因此其移植工作仅仅需要几步。

在移植之前，我们要先做一些准备工作，以确定移植可以进行；然后，分别针对各个部分，编写相应的移植代码即可。最后，还可以用一些测试用例来测试系统是否正确移植成功。

7.2 Checklist Before Porting

7.2.1 Processor

RME 要求所选择的处理器能够容纳一个完整的全功能操作系统。处理器必须具备一定程度的处理能力，以克服进行系统调用的开销，使使用一个全功能操作系统具有实际意义；处理器还必须具备一定的内存访问控制能力，可以是内存管理单元，也可以是内存保护单元。

理想地，这种系统一般都是主频达到 GHz 级别，有百 MB 以上级别 RAM 的 32 位以上单核或多核微处理器平台。但是，RME 也可以支持仅有 16kB RAM 和 64kB ROM 的 32 位单片机。

RME 不能在少于 32kB 存储器的平台上运行，也不能在低于 32 位的处理器上运行。此外，如果所选择的处理器没有内存保护功能，那么运行 RME 是没有意义的；在这种状况下，RMP 可能是一个更好的选择。最后，由于 RME 不支持硬件堆栈机制[2]，堆栈必须是由软件实现的[3]。

7.2.2 Compiler

RME 要求编译器是 C89 标准的，并能够根据一定的函数调用约定生成代码。由于 RME 的代码非常标准，也不使用 C 运行时库中的库函数，因此只要编译器符合 ANSI C89 标准即可。通常的 GCC、CLANG/LLVM、MSVC、ARMCC、ICC、IAR、TASKING 等编译器都是满足这个需求的。RME 没有使用位段、enum 和结构体对齐等各编译器实现差别较大的编译器扩展，也没有使用 C 语言中的未定义操作[4]，因此保证了最大限度的兼容性。

在使用低质量编译器时，要注意关闭死代码消除功能和链接时优化功能，最好也要关闭编译器的循环不变量外提优化。某些激进的优化有可能导致中断处理向量被整个优化掉[5]，引起内核无法工作，也

[1] 为了维护工程规整性，RME 对此规范有一些必要的背离，见 9.2.6

[2] 这是 PIC 单片机等少数架构的典型实现方式

[3] 也即堆栈指针可以由用户修改，堆栈实现在内存中

[4] 除了某一小部分因别无他法造成的未定义行为，见 9.2.6

[5] 因为在函数调用图中它们不会被任何函数调用，某些低质量链接器又不将中断向量纳入函数调用图进行分析

有可能导致内核的其他功能故障。不要使用任何激进的编译优化选项，在一般的编译器上，推荐的优化选项是（如 GCC）-O2 或相当的优化水平。

7.2.3 Assembler

RME 要求汇编器能够引入 C 中的符号，并根据函数调用约定进行调用；此外，也要求汇编器产生的代码能够导出并根据函数调用约定被 C 调用。这通常是非常好满足的要求。如果编译器可以内联汇编，那么不需要汇编器也是可以的。

7.2.4 Linker

RME 要求链接器必须具备链接多个对象[1]文件到一个中间对象文件的能力，而且要求能够接受定制的链接器脚本。通常的 ld、cl、armlink、ilink/xlink 等链接器都足以满足这种需求。每种链接器的链接器脚本往往都不相同，这往往需要根据每种链接器的语法决定。

7.2.5 Debugger

RME 对调试器没有特别的要求。如果有调试器可用的话，当然是最好的，但是没有调试器也是可以进行移植的。在有调试器的情况下可以直接用调试器查看内核变量；在没有调试器的情况下，要先实现内核最底层的__RME_Putchar 函数，实现单个字符的打印输出，然后就可以用该打印输出来输出日志了。关于该函数的实现请看下节所述。

7.3 Architecture-depedent Portions of RME

RME 的架构相关部分代码的源文件全部都放在 Platform 文件夹的对应架构名称下。如 Cortex-M 架构的文件夹名称为 Platform/CortexM。其头文件在 Include/Platform/CortexM，其他架构依此类推。

每个架构都包含一个或多个源文件和一个或多个头文件。内核包含架构相关头文件时，总是会包含 Include/rme_platform.h，而这是一个包含了对应架构顶层头文件的头文件。在更改 RME 的编译目标平台时，通过修改这个头文件来达成对不同目标的相应头文件的包含。比如，要针对 Cortex-M 架构进行编译，那么该头文件就应该包含对应 Cortex-M 的底层函数实现的全部头文件。

在移植之前，可以先浏览已有的移植，并寻找一个与目标架构的逻辑组织最相近的架构的移植。然后，可以将这个移植拷贝一份，并将其当做模板进行修改。

7.3.1 Typedefs

对于每个架构/编译器，首先需要移植的部分就是 RME 的类型定义。RME 的类型定义一共有如下五个：

Table 7-1 Overview of Typedefs

[1] 一般是.o，.obj 或.elf 文件

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;

7.3.2 Regular Macro Defines

其次，需要移植的是 RME 的一般宏定义。RME 的一般宏定义一共有如下几个：

Table 7-2 Overview of Regular Macro Defines

宏名称	作用
EXTERN	编译器的 extern 关键字。某些编译器可能具有不标准的 extern 关键字[1]，此时用这个宏定义来处理它。 例子： #define EXTERN extern #define EXTERN extern "C"
INLINE	编译器的 inline 关键字。某些编译器可能不支持内联函数功能，此时只要留空即可。 例子： #define INLINE inline #define INLINE __inline #define INLINE __forceinline

[1] 或者需要将一些 C++代码和内核链接起来

宏名称	作用
RME_LIKELY(X)	编译器的 likely 关键字，用于指导分支预测，表示此分支很有可能被执行。如果编译器有该功能，就定义此关键字；如果没有，将它定义为(X)即可。 例子： #define RME_LIKELY(X) likely(X) #define RME_LIKELY(X) __builtin_expect(!!(X),1) #define RME_LIKELY(X) (X)
RME_UNLIKELY(X)	编译器的 unlikely 关键字，用于指导分支预测，表示此分支很有可能不被执行。如果编译器有该功能，就定义此关键字；如果没有，将它定义为(X)即可。 例子： #define RME_UNLIKELY(X) unlikely(X) #define RME_UNLIKELY(X) __builtin_expect(!!(X),0) #define RME_UNLIKELY(X) (X)
RME_CPU_LOCAL()	该宏返回一个指向 CPU 本地存储的，struct RME_CPU_Local*类型的指针。对于单核处理器而言，CPU 本地存储可以在架构相关层中直接声明为一个变量，此宏只需要返回该变量的地址即可；对于多核处理器而言，CPU 本地存储结构体需要在初始化系统时按照系统中处理器的个数动态创建。 例子： #define RME_CPU_LOCAL() &RME_XXX_CPU_Local （某单核架构，静态分配） #define RME_CPU_LOCAL() asm("mov some_reg, r0") （某多核架构，动态分配，地址存储在专用段寄存器中）
RME_WORD_ORDER	处理器字长（按 Bit 计算）对应的 2 的方次。比如，32 位处理器对应 5，64 位处理器对应 6，依此类推。 例子： #define RME_WORD_ORDER 5
RME_VA_EQU_PA	处理器是否要求虚拟地址总是等于物理地址。通常而言，对于基于 MMU 的系统，这一项总是填写“否”（RME_FALSE），此时使用常规页表；对于微控制器等基于 MPU 的系统，这一项总是填写“是”（RME_TRUE），此时使用路径压缩页表。 例子： #define RME_VA_EQU_PA RME_TRUE

宏名称	作用
RME_QUIE_TIME	安定时间的长度，单位是时间片。对于单处理器，由于没有真正的并行性，各个权能总是立即安定的，此项填写“0”；对于多处理器系统，理论上此项填写的值应当超过内核最坏执行时间（WCET）的两倍大小，工程中则推荐十倍大小。通常而言，一个时间片的时长（最小 100us 量级）远超过内核的 WCET（10us 量级），因此这里填写 1 即可。 例子： #define RME_QUIE_TIME 1
RME_CAPTBL_LIMIT	权能表的最大长度，单位是权能数量。由于权能表在创建和删除时都需要遍历整个表，因此此宏可通过控制权能表的大小来控制此种遍历的执行时间上限。对于那些对实时性没有过高要求或由受信任的用户态权能管理器控制权能表大小上限的场合，该值可以设置为 0；否则可将该值设置为希望限制的权能表大小的上限。如果该值不是 0，最终的权能表大小上限由该值和处理器字长所允许的上限中较小的那一个值确定。 例子： #define RME_CAPTBL_LIMIT 0 （无限制的场合） #define RME_CAPTBL_LIMIT 128 （限制权能表大小的场合）
RME_KMEM_VA_START	用户可分配的内核虚拟内存的起始地址。填写内核虚拟内存的起始地址即可。创建内核对象时，将从这里开始分配内核内存，并且将这些分配记录在内核内存登记表中。 例子： #define RME_KMEM_VA_START 0xC0000000
RME_KMEM_SIZE	用户可分配的内核虚拟内存的地址空间的大小。填写内核虚拟内存的地址空间大小即可。对于那些需要动态探测内核虚拟内存空间大小的场合[1]，这里填写内核允许的最小大小[2]。 例子： #define RME_KMEM_SIZE 0x30000000
RME_HYP_VA_START	虚拟机监视器专用虚拟内存的起始地址。这段内存是给虚拟机监视器使用的，可以设置线程的虚拟机属性到这段地址，以使其寄存器

[1] 比如 x86-64 等

[2] 如果探测到比这个大小还小的可用内核内存虚拟地址空间，内核可以拒绝启动

宏名称	作用
	在线程切换时被保存至此。 例子： #define RME_HYP_VA_START 0xF000000
RME_HYP_SIZE	虚拟机监视器专用虚拟内存的大小。填写虚拟机专用虚拟内存的实际大小即可。如果不使用这个功能，那么该宏的大小需要设置为 0，此时宏 RME_HYP_VA_START 也无效。 例子： #define RME_HYP_SIZE 0x10000000
RME_KMEM_SLOT_ORDER	内核虚拟内存分配粒度对应的 2 的方次。比如如果内核内存分配的最小粒度为 16Byte，那么这个位置要填写的数字就是 $\log_2(16) = 4$。需要注意的是，内存分配的最小粒度不能小于一个处理器字长。 例子： #define RME_KMEM_SLOT_ORDER 4
RME_KMEM_STACK_ADDR	内核堆栈起始虚拟地址。如果堆栈向下生长，这就是堆栈的顶部；如果堆栈向上生长，这就是堆栈的底部。 例子： #define RME_KMEM_STACK_ADDR 0xF0000000
RME_MAX_PREEMPT_PRIO	内核支持的抢占优先级的最大数量。这个数量必须是处理器字长（按 Bit 计算）的整数倍。通常而言，把这个值定义为处理器字长就可以了。 例子： #define RME_MAX_PREEMPT_PRIO 32
RME_PGTBL_SIZE_NOM(X)	处理器非顶层页目录的大小。这个宏会接受一个参数，该参数的意义是该页目录的表项数目对应的 2 的方次。如果该页目录中含有 1024 个表项，那么 X 的值即为 10，此时该宏为 RME_PGTBL_SIZE_NOM(10)，它会返回该页表的大小，单位为字节。如果每个页表表项的大小是 4 字节，附加在页表上的附加前置数据[1]的大小为 4096 字节，那么该宏应当返回 8192。
RME_PGTBL_SIZE_TOP(X)	处理器顶层页目录的大小。这个宏会接受一个参数，该参数的意义是该页目录的表项数目对应的 2 的方次。这个宏实际上等于 RME_PGTBL_SIZE_NOM(X)加上顶层额外的附加数据[2]的大小。
RME_KOTBL	内核内存登记表所在的内核虚拟内存地址。对于大部分架构，直接

[1] 仅在使用 MPU 时存在

[2] 仅在使用 MPU 时存在

宏名称	作用
	将该宏定义为内核默认位置也即 RME_Kotbl 即可；对于小部分拥有极高内存量的架构[1]，因为 GCC 等编译器最多默认放置内核到高 2GB，内核内存登记表的大小会被限制在 2GB，此时最多支持 1TB 内核内存。因此此时需要重定位该登记表，将其指向不受限制的地址。 例子： #define RME_KOTBL　RME_Kotbl （小内核内存分配表） #define RME_KOTBL　((ptr_t*)0xFFFF800001000000) （大内核内存分配表）
RME_READ_ACQUIRE(X)	内存读屏障。在强顺序一致性架构上这个宏可以直接实现为对 X 地址的读取；在其他情况下，这个值应该被定义为一个实现读屏障的函数。如果编译器有内建函数可以执行该操作，使用内建函数达成此效果也可以。这个宏应当保证宏下方的内存操作在自身的读操作进行完毕之前不会被执行，相当于保证了读-读和读-写的一致性。值得注意的是，RME 并不需要保证写-读的顺序一致性。如果在某些架构上仅有写-读顺序不一致[2]，这个宏也可以实现为一个简单读取。 例子： #define RME_READ_ACQUIRE(X) (*(X)) （简单读取的情况） #define RME_READ_ACQUIRE(X) asm("LDA X") （处理器提供了读获取指令的情况） #define RME_READ_ACQUIRE(X) asm("LDR X; DMB") （处理器提供了内存屏障的情况） #define RME_READ_ACQUIRE(X)　__atomic_read(X) （使用编译器内建函数的情况）
RME_WRITE_RELEASE(X,V)	内存写屏障。在强顺序一致性架构上这个宏可以直接实现为对 X 地址的写入；在其他情况下，这个值应该被定义为一个实现写屏障的函数。如果编译器有内建函数可以执行该操作，使用内建函数达成此效果也可以。这个宏应当保证自身的写操作在宏上方的内存操作进行完毕之前不会被执行，相当于保证了读-写和写-写的一致

[1] 如最新的 x86-64 可以有上百 TB 内存

[2] 比如 x86-64 等

宏名称	作用
	性。值得注意的是，RME 并不需要保证写-读的顺序一致性。如果在某些架构上仅有写-读顺序不一致[1]，这个宏也可以实现为一个简单写入。 例子： #define RME_WRITE_RELEASE(X,V) ((*(X))=(V)) （简单写入的情况） #define RME_WRITE_RELEASE(X,V) asm("STL X,V") （处理器提供了写释放指令的情况） #define RME_WRITE_RELEASE(X,V) asm("DMB;STR X,V") （处理器提供了内存屏障的情况） #define RME_WRITE_RELEASE(X,V) __atomic_write(X,V) （使用编译器内建函数的情况）

7.3.3 Architecture Related Structs

RME 的架构相关结构体一共有三个，分别如下：

Table 7-3 Overview of Architecture Related Structs

结构体	意义
RME_Reg_Struct	进入中断函数时寄存器压栈的结构体，包含了 CPU 的各个寄存器。
RME_Cop_Struct	进入中断函数时协处理器（如 FPU 等）的结构体，包含了其各个寄存器。
RME_Iret_Struct	与程序执行流相关的，在线程迁移调用中要保存和恢复的寄存器的结构体。

这三个结构体的实现和系统中断向量进入段汇编函数的实现有关。

7.3.4 Low-level Assembly Functions

RME 仅要求用汇编或内联汇编实现 4 个短小的底层汇编函数。这些函数的名称和意义如下：

Table 7-4 Overview of Low-level Assembly Functions

函数名	意义
__RME_Disable_Int	禁止处理器中断。

[1] 比如 x86-64 等

函数名	意义
__RME_Enable_Int	使能处理器中断。
_RME_Kmain	内核入口外壳函数。
__RME_Enter_User_Mode	进入用户态执行。

这些函数的具体实现方法和实现次序将在后面章节加以讲解。

7.3.5 System Interrupt Vectors

RME 最低仅仅要求用汇编或内联汇编实现 3 个中断向量。这些中断向量的名称和意义如下：

Table 7-5 Overview of System Interrupt Vectors

中断向量名	意义
系统定时器中断向量	处理系统定时器中断，管理时间片使用。
系统调用中断向量	处理系统调用时使用。
系统错误中断向量	发生访存错误及其他处理器错误时使用。

这些中断向量的具体实现方法和实现次序将在后面章节加以讲解。

7.3.6 Other Low-level Functions

这些底层函数涉及到页表、处理器特殊功能等其他方面。这些函数可以用汇编实现，也可以不用汇编实现，也可以部分使用 C 语言，部分使用内联汇编实现。这些函数的可以分成如下几类：

7.3.6.1 Kernel Debug Print Function

Table 7-6 Overview of Kernel Debug Print Functions

函数	意义
__RME_Putchar	打印一个字符到内核调试控制台。

7.3.6.2 Atomic Operations and Special Operation Defines/Functions

Table 7-7 Overview of Atomic Operation & Special Operation Macros/Functions

宏定义/函数	意义
RME_COMP_SWAP	比较交换原子操作。
RME_FETCH_ADD	加载自增原子操作。

宏定义/函数	意义
RME_FETCH_AND	加载逻辑与原子操作。
RME_MSB_GET	得到一个字的最高位（MSB）位置。

7.3.6.3 Initialization and Booting Functions

Table 7-8 Overview of Initialization and Booting Functions

函数	意义
__RME_Low_Level_Init	底层硬件初始化。
__RME_Boot	创建初始内核对象并启动系统。

7.3.6.4 Register Set Related Functions

Table 7-9 Overview of Register Set Related Functions

函数	意义
__RME_Get_Syscall_Param	从寄存器组中得到系统调用参数。
__RME_Set_Syscall_Retval	向寄存器组中设置系统调用的返回值。
__RME_Thd_Reg_Init	初始化线程或迁移调用的寄存器组。
__RME_Thd_Reg_Copy	将一个寄存器组拷贝到另一个寄存器组。
__RME_Thd_Cop_Init	初始化线程的协处理器寄存器组。
__RME_Thd_Cop_Save	保存线程的协处理器寄存器组。
__RME_Thd_Cop_Restore	恢复线程的协处理器寄存器组。
__RME_Inv_Reg_Save	保存线程迁移调用返回用的必要寄存器。
__RME_Inv_Reg_Restore	恢复线程迁移调用返回用的必要寄存器。
__RME_Set_Inv_Retval	向寄存器组中设置线程迁移调用的返回值。

7.3.6.5 Kernel Function Handler

Table 7-10 Overview of Kernel Function Handler

函数	意义
__RME_Kern_Func_Handler	内核功能调用的实现。

7.3.6.6 Page Table Related Functions

Table 7-11 Overview of Page Table Related Functions

函数	意义
__RME_Pgtbl_Set	切换当前使用的页表[1]。
__RME_Pgtbl_Kmem_Init	初始化内核页表。
__RME_Pgtbl_Check	检查页目录参数是否能被本架构支持。
__RME_Pgtbl_Init	初始化页目录。
__RME_Pgtbl_Del_Check	检查该页目录是否能被删除。
__RME_Pgtbl_Page_Map	映射一个页到页目录内。
__RME_Pgtbl_Page_Unmap	从页目录内删除一个页的映射。
__RME_Pgtbl_Pgdir_Map	映射一个子页目录到一个父页目录内。
__RME_Pgtbl_Pgdir_Unmap	从父页目录内删除一个子页目录的映射。
__RME_Pgtbl_Lookup	在一个页目录内根据相对位置查找一个物理地址页。
__RME_Pgtbl_Walk	从顶层页目录开始查找一个虚拟地址对应的物理地址页属性。

7.4 Porting of Type Definitions and Low-level Assembly Functions

对于类型定义，只需要确定处理器的字长在编译器中的表达方法，使用 typedef 定义即可。需要注意的是，对于某些架构和编译器，long（长整型）类型对应的是两个机器字的长度，而非一个机器字；此时应当使用 int 类型来表达一个机器字的长度。对于另一些架构和编译器，int 是半个机器字的长度，long 是一个机器字的长度，此时应当注意用 long 来定义一个机器字。

在必要的时候，可以使用 sizeof()运算符编写几个小程序，来确定该编译器的机器字究竟是何种标准。

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或者一百二十八位整数，那么这几个类型可以略去。

Table 7-12 Overview of Commonly Used Types

类型	意义
s8_t	一个有符号八位整形。 例如：typedef char s8_t;
s16_t	一个有符号十六位整形。

[1] 顶层页目录

	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;
s128_t	一个有符号一百二十八位整形。 例如：typedef long long s128_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;
u128_t	一个有符号一百二十八位整形。 例如：typedef unsigned long long u128_t;

对于宏定义和结构体类型的定义，需要根据具体系统的配置来决定。具体的决定方法见上节所述，依表格说明填充这些定义即可。

接下来说明对于汇编底层函数的移植过程。

7.4.1 Implementation of __RME_Disable_Int

该函数需要关闭处理器的中断，然后返回。实现上没有特别需要注意的地方，通常而言只需要写一个 CPU 寄存器或者外设地址，关闭中断，然后返回即可。

Table 7-13 Implementation of __RME_Disable_Int

原型	void __RME_Disable_Int(void)
意义	关闭处理器中断。
返回值	无。
参数	无。

7.4.2 Implementation of __RME_Enable_Int

该函数需要开启处理器的中断，然后返回。实现上没有特别需要注意的地方，通常而言只需要写一个 CPU 寄存器或者外设地址，开启中断，然后返回即可。

Table 7-14 Implementation of __RME_Enable_Int

原型	void __RME_Enable_Int(void)
意义	开启处理器中断。
返回值	无。
参数	无。

7.4.3 Implementation of _RME_Kmain

该函数需要将 Stack 的值赋给内核态的堆栈指针，然后跳转到 RME_Kmain 函数即可。这个函数是不会返回的。

Table 7-15 Implementation of _RME_Kmain

原型	void _RME_Kmain(ptr_t Stack)
意义	内核的底层入口函数。
返回值	无。
参数	ptr_t Stack 内核要使用的栈虚拟地址。

在调用这个内核入口函数函数之前，需要进行如下准备工作：

1. 将内核的各个部分通过启动器（Bootloader）正确地加载到内存中，并将处理器置于特权态。

2. 建立最初的系统启动用页表，并使用该页表将系统切换到保护模式。该页表只要实现了内核内存的虚拟地址到内核内存的物理地址的映射即可。这个临时页表仅仅在启动过程中使用一次，在之后就不再使用了，因此在系统启动完成后可以将其删除。如果创建并切换到临时页表的工作没有在本函数之前进行，那么这个工作需要由本函数正确实现。

7.4.4 Implementation of __RME_Enter_User_Mode

该函数实现从特权态到用户态的切换，仅在系统启动阶段的最后被调用。在此之后，系统进入正常运行状态。该函数只要将 Stack_Addr 的值赋给堆栈指针，将 CPUID 赋给调用约定决定的第一个参数的寄存器，然后直接跳转到 Entry_Addr 并进行处理器状态切换即可。该函数将永远不会返回。

Table 7-16 Implementation of __RME_Enter_User_Mode

原型	void __RME_Enter_User_Mode(ptr_t Entry_Addr, ptr_t Stack_Addr, ptr_t CPUID)
意义	进入用户模式，开始执行第一个进程。
返回值	无。
参数	ptr_t Entry_Addr

	第一个用户态应用程序的入口虚拟地址。
ptr_t Stack_Addr	第一个用户态应用程序的栈虚拟地址。
ptr_t CPUID	该线程所属的 CPUID。

7.5 Porting of System Interrupt Vectors

系统中断向量的移植的主要工作包括两部分：一部分是进入中断向量和退出中断向量的汇编代码，另一部分是系统中断向量本身。RME 仅仅要求实现最少三个中断向量。中断向量进入部分要求保存处理器的寄存器到栈上，其退出部分则要求从栈上恢复这些寄存器。在中断向量中还可能涉及对系统协处理器寄存器的保存和恢复。

7.5.1 Entry & Exit of Interrupt Vectors and Architecture Related Structs

中断向量的进入阶段，需要将要由中断保存的处理器的各个寄存器压栈处理，压栈的顺序应当和定义的寄存器结构体一致。在压栈完成后，需要调用相应的处理函数，并且把指向栈上寄存器结构体的指针传给它。在中断向量的退出阶段，只需要从栈上按相反顺序弹出寄存器组即可。在中断向量中，如果涉及到线程切换，系统会判断是否需要保存和恢复协处理器的寄存器组。如果需要的话，协处理器寄存器组会被保存和恢复。协处理器寄存器组不会被压栈，因此协处理器寄存器结构体只要包括协处理器的全部寄存器就可以了，无须关心顺序。

如果栈是向下生长的满堆栈，那么全部压栈完成后，堆栈指针就是指向结构体的指针；

如果栈是向下生长的空堆栈，那么全部压栈完成后，堆栈指针加上处理器字长（以 Byte 为单位）就是指向结构体的指针。

如果栈是向上生长的满堆栈，那么全部压栈完成后，将堆栈指针减去寄存器结构体的大小再加上处理器字长（以 Byte 为单位）就是指向结构体的指针。

如果栈是向上生长的空堆栈，那么全部压栈完成后，将堆栈指针减去寄存器结构体的大小就是指向结构体的指针。

定时器中断处理函数、系统调用处理函数和内存错误处理函数都只接受指向堆栈的指针这一个参数。由于这三个函数一般都用 C 语言写成，因此参数的传入要根据 C 语言调用约定进行。

7.5.1.1 System Timer Interrupt Vector

在定时器中断处理向量中，需要调用如下函数：

Table 7-17 Implementation of System Timer Interrupt Vector

原型	void _RME_Tick_Handler(struct RME_Reg_Struct* Reg)
意义	执行定时器中断处理。

返回值	无。
参数	struct RME_Reg_Struct* Reg 在进入阶段被压栈的处理器寄存器组。

这个函数是系统实现好的，无需用户自行实现。在多处理器系统中，本函数是给主处理器调用的；对于那些从处理器，则需要调用_RME_Tick_SMP_Handler。这两个函数是相同的，唯一的区别是从处理器版本不更新时间戳的值。这是由于一般仅由主处理器维护时间戳计数器，并在时钟中断发生后给其他处理器发送处理器间中断（Inter-Processor Interrupt，IPI）通知它们来重新调度线程。

7.5.1.2 System Call Interrupt Vector

在系统调用中断向量中，需要调用如下函数：

Table 7-18 Implementation of System Call Interrupt Vector

原型	void _RME_Svc_Handler(struct RME_Reg_Struct* Reg)
意义	执行系统调用处理。
返回值	无。
参数	struct RME_Reg_Struct* Reg 在进入阶段被压栈的处理器寄存器组。

这个函数也是系统实现好的，无需用户自行实现。

7.5.1.3 System Fault Handling Vector

在系统错误处理中断向量中，需要调用一个用户提供的系统错误处理函数。该函数的名称可由用户自行决定，但其原型必须如下所示。关于该函数的实现请参看 7.5.2。

Table 7-19 Implementation of System Fault Handling Vector

原型	void _RME_Fault_Handler(struct RME_Reg_Struct* Reg)
意义	执行系统错误处理。
返回值	无。
参数	struct RME_Reg_Struct* Reg 在进入阶段被压栈的处理器寄存器组。

7.5.2 Fault Handling Interrupt Vectors

在该向量中，需要调用一个系统错误处理函数。该函数的描述见上节所述。该函数的实现是与架构紧密相关的，因此需要在移植时重新设计。该函数首先需要判断发生的错误是可恢复错误还是不可恢复错误。如果发生的是不可恢复错误[1]，那么直接调用由系统提供好的如下函数即可：

Table 7-20 Callable Function in System Fault Handling Vector

原型	ret_t __RME_Thd_Fatal(struct RME_Reg_Struct* Reg)
意义	该线程发生了致命的不可恢复错误，或者恢复失败，需要杀死该线程。
返回值	ret_t 总是返回 0。
参数	struct RME_Reg_Struct* Reg 在中断进入阶段被压栈的处理器寄存器组。

如果发生的是可恢复错误[2]，那么可以在进行完相应的处理和恢复工作之后，立即退出中断向量。如果内核本身的恢复失败，那么也需要调用上述函数杀死该线程，然后立即退出中断向量。该函数会通过调度器事件和调度器信号[3]通知用户态来处理这一错误。

7.6 Porting of the Kernel Debug Print Function

内核调试打印函数的底层接口只有一个函数，如下：

Table 7-21 Implementation of __RME_Putchar

原型	ptr_t __RME_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。

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。

7.7 Porting of Processor Specific Operation Macros

[1] 比如未定义指令、访存错误等等

[2] 比如页面交换、缺页中断或者 MPU 动态页的映射等

[3] 如果该线程在绑定时注册了信号端点的话

处理器特殊功能宏定义包括了原子操作和最高位（Most Significant Bit，MSB）查找。原子操作是用来在多核条件下实现无锁内核的。最高位查找则能加快最高优先级的查找。这些宏定义提供的功能可以用 C 语言实现为函数，也可以用汇编或内联汇编实现，视情况而定。如果是用汇编语言实现，要注意遵循 C 语言调用约定，因为这些函数要被 C 语言调用。如果使用通常而言较易出错的内联汇编实现，则需要注意实现的正确性。

7.7.1 Compare-and-Swap

该宏/函数完成一个基本的比较交换（Compare-And-Swap，CAS）原子操作。在某些架构上它有直接的指令支持[1]，此时可以考虑以汇编或内联汇编实现该指令。在某些 RISC 架构上，也可以考虑使用排他性加载和排他性写回指令来支持[2]。对于某些更新的处理器，可以考虑用排他性获取加载和排他性写回释放指令来实现[3]。具体的支持方法随各个处理器而有不同。

Table 7-22 Implementation of RME_COMP_SWAP

宏定义	ptr_t RME_COMP_SWAP(ptr_t* PTR, ptr_t OLD, ptr_t NEW)
意义	进行比较交换原子操作。该操作会比较 OLD 和*PTR 的值，如果 OLD 和*PTR 不相等，那么返回 0；如果 OLD 和*PTR 相等，那么返回 1，并且把 NEW 的值赋给*PTR。
返回值	ptr_t 该宏/函数是否成功的返回值。成功返回 1，失败返回 0。
参数	ptr_t* PTR 指向目标操作地址的指针。
	ptr_t OLD 参加比较的老值。
	ptr_t NEW 如果老值和目标地址的值相同，此时要赋给目标地址的新值。

由于 RME 的内核在读写某些数据结构后使用 CAS 置位一些标志，也在使用 CAS 置位某些标志后读写一些数据结构，因此我们不希望 CAS 与其前后的其他内存操作在其他处理器看来乱序发生。如果目标处理器不是顺序一致性（Strongly-ordered）的[4]，而且原子操作不是内存访问串行化指令（Memory access serializing instruction），那么该宏/函数在实现时，必须在操作的前后都加上一个完全的内存屏障。

[1] 如 x86-64 的 PREFIX LOCK CMPXCHG 指令

[2] 如 ARMv7 的 LDREX 和 STREX 指令

[3] 如 ARMv8 的 LDAEX 和 STLEX 指令

[4] 比如 PowerPC 和 ARMv7、ARMv8

严格地讲，在有排他性加载和写回指令的处理器上，CAS 是分成两个步骤执行的，一个是排他性加载，另外一个是排他性写回。考虑到这一点，RME 的设计允许放宽一点限制，允许排他性加载和 CAS 操作上方的其他操作乱序，也允许排他性写回和 CAS 下方的其他操作乱序。因此，在具备排他性获取加载和排他性写回释放的处理器上，任何额外的内存屏障都是不必要的。

7.7.2 Fetch-and-Add

该宏/函数完成一个基本的加载自增（Fetch-And-Add，FAA）原子操作。在实现时应当注意，当多个处理器同时进行该操作时，该宏/函数在各个处理器上的返回值组成的集合必须是连续的，不能有跳跃和间断。在某些架构上它有直接的指令支持[1]，此时可以考虑以汇编或内联汇编实现该指令。在某些 RISC 架构上，也可以考虑使用排他性加载和排他性写回指令来支持[2]。具体的支持方法随各个处理器而有不同。RME 的设计使得该指令不需要保证有完全内存屏障的作用。

Table 7-23 Implementation of RME_FETCH_ADD

宏定义	ptr_t RME_FETCH_ADD(ptr_t* PTR, cnt_t ADDEND)
意义	进行加载自增原子操作。该操作会把*PTR 的值加上 ADDEND，然后写回*PTR，并且返回加上 ADDEND 之前的*PTR。
返回值	ptr_t 加上 ADDEND 之前的*PTR。
参数	ptr_t* PTR 指向目标操作地址的指针。
	cnt_t ADDEND 目标操作地址要加上的数。该数可以是一个正数也是一个负数。

RME 假设该操作是无等待（Wait-free）的。但是，在仅具备排他性加载和排他性写回指令的处理器上，该操作必须用一个循环尝试实现，使得其无等待性受到一定影响。但是，在实践中这不会破坏系统的实时响应：因为 RME 的系统调用有一个最小执行时间，而该时间限制了系统调用的执行频率，通常而言该频率（即便乘以 CPU 的数量也）远低于循环尝试的频率。因此，该操作循环尝试的次数在实际应用中是有一个可接受的上界的，不会无限循环下去。

7.7.3 Fetch-and-And

该宏/函数完成一个基本的逻辑与（Fetch-And-aNd，FAN）原子操作。在某些架构上它有直接的指令支持[3]，此时可以考虑以汇编或内联汇编实现该指令。在某些 RISC 架构上，也可以考虑使用排他性加

[1] 如 x86-64 的 PREFIX LOCK XADDL 指令

[2] 如 ARMv7 的 LDREX 和 STREX 指令

[3] 如 x86-64 的 PREFIX LOCK ANDL 指令

载和排他性写回指令来支持[1]。具体的支持方法随各个处理器而有不同。RME 的设计该指令不需要保证有完全内存屏障的作用。

Table 7-24 Implementation of RME_FETCH_AND

宏定义	ptr_t RME_FETCH_AND(ptr_t* PTR, ptr_t OPERAND)
意义	进行逻辑与原子操作。该操作会把*PTR 的值和 OPERAND 进行逻辑与，然后写回*PTR，并且返回和 OPERAND 进行逻辑与之前的*PTR。
返回值	ptr_t 与上 OPERAND 之前的*PTR。
参数	ptr_t* PTR 指向目标操作地址的指针。
	ptr_t OPERAND 目标操作地址要与上的无符号数。

RME 假设这个操作也是无等待的。在这方面的具体细节请参见上一节关于 FAA 的描述。

7.7.4 Get the Most Significant Bit Position

该宏/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（LSB 为第 0 位）。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。

Table 7-25 Implementation of RME_MSB_GET

宏定义	ptr_t RME_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。
参数	ptr_t VAL 要计算最高位位置的数字。

由于该宏/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宏/函数，使用该指令即可。对于那些提供了前导零计算指令

[1] 如 ARMv7 的 LDREX 和 STREX 指令

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。

7.8 Porting of Initialization and Startup Functions

初始化与启动函数一共有四个，如下所示。

7.8.1 Implementation __RME_Low_Level_Init

这个函数需要进行处理器时钟、Cache 等除了内存管理单元之外的底层硬件的初始化。这里不需要进行内存管理单元初始化的原因是，内存管理单元实际上已经在_RME_Kmain 退出之前被初始化了。

Table 7-26 Implementation of __RME_Low_Level_Init

原型	ptr_t __RME_Low_Level_Init(void)
意义	进行最底层硬件的初始化。这包括了处理器时钟的初始设置、必要的其他硬件[2]的初始化等等。在这个函数运行完成后，内核数据结构的初始化才开始。
返回值	ptr_t 总是返回 0。
参数	无。

7.8.2 Implementation __RME_Boot

这个函数是 RME 启动过程中最重要的函数。它在内核态运行，创建 Init 进程的权能表、页表，将所有的用户可访问页添加进 Init 进程的页表，创建所有的内核信号端点和内核调用权能。在单核系统下，该函数需要创建一个线程，设置执行属性并在最后调用_RME_Enter_User_Mode 切换到它进行执行。在多核系统下，系统需要初始化其他处理器，并且需要让它们在各自的 CPU 核上创建属于一个自己的线程，然后跳转到该线程中去运行。

Table 7-27 Implementation of __RME_Boot

原型	ptr_t __RME_Boot(void)
意义	该函数启动系统中的第一个进程 Init，并且初始化系统中所有的内核信号端点、内核功能调用权能，而且负责把系统中的所有用户可访问页添加进 Init 的初始页表。

[1] 如提供了 CLZ 指令的 ARMv7

[2] 如 Cache 控制器和中断控制器，或者处理器主板上的必须在上电初期初始化的其他外设

返回值	ptr_t 总是返回 0。
参数	无。

该函数需要调用的各个函数如下。除最后列出的三个函数由用户提供之外，其他函数均是 RME 的内建函数。

Table 7-28 Functions that Need to be Called in __RME_Boot

函数	调用次数	意义
_RME_Kotbl_Init	全系统只需调用一次	在启动时初始化内核内存登记表。
_RME_CPU_Local_Init	每个处理器调用一次	在启动时初始化各处理器的本地存储。
_RME_Captbl_Boot_Init	全系统只需创建一次	在启动时创建初始权能表。
_RME_Captbl_Boot_Crt	视情况而定	在启动时创建其它权能表。
_RME_Pgtbl_Boot_Crt	全系统只需创建一组	在启动时创建页目录。
_RME_Pgtbl_Boot_Con	全系统只需调用一组	在启动时构造页目录。
_RME_Pgtbl_Boot_Add	全系统只需调用一组	在启动时向页目录中添加页。
_RME_Proc_Boot_Crt	全系统只需创建一次	在启动时创建第一个进程。
_RME_Kern_Boot_Crt	全系统只需创建一次	在启动时创建内核功能调用权能。
_RME_Kmem_Boot_Crt	视情况而定	在启动时创建内核内存权能。
_RME_Sig_Boot_Crt	视情况而定	在启动时创建内核信号端点。
_RME_Thd_Boot_Crt	每个处理器调用一次	在启动时创建初始线程。
__RME_Pgtbl_Set	每个处理器调用一次	设置处理器使用当前页表。
__RME_Enable_Int	每个处理器调用一次	使能中断。
__RME_Enter_User_Mode	每个处理器调用一次	进入用户态开始执行。

上述函数是按照调用的逻辑顺序列出的。这些函数的介绍和调用方法如下所示。一旦其中任何一个函数返回失败，那么就需要停止整个系统启动过程。因此，建议使用 RME_ASSERT(func(...) = = 0)的宏判断包裹这些函数，一旦失败即进入死循环，打印内核崩溃信息。

7.8.2.1 Initializing Kernel Memory Allocation Table at Boot-time

该函数用来在系统启动时初始化内核内存登记表。初始化的工作是将该登记表清零，代表没有内核内存被占用。内核启动时会默认调用一次该函数，初始化内核内存登记表的编译时就能确定的部分。对

于某些架构，这就足够了，无需再次调用该函数[1]。但是在另一些些架构上，内核内存的数量需要被动态探测，因此该工作可能要由移植者再次调用该函数进行[2]。

Table 7-29 Parameters Needed for Init. Kernel Memory Allocation Table at Boot-time

原型	ret_t _RME_Kotbl_Init(ptr_t Words)	
参数名称	类型	描述
Words	ptr_t	内核内存登记表的大小，单位是处理器字长。这个值要根据内核内存分配粒度和探测到的内核内存大小动态计算，具体计算方法是将内核内存的地址空间大小除以内核内存分配粒度，然后再除以处理器的位数。

该函数的返回值可能如下：

Table 7-30 Possible Return Values for Init. Kernel Memory Allocation Table at Boot-time

返回值	意义
0	操作成功。
-1	传入的内核内存登记表大小比默认的最小大小要小。默认的最小大小是由宏 RME_KMEM_SIZE 计算得出的。

7.8.2.2 Initializing CPU-local Storage at Boot-time

该函数用来在系统启动时初始化各处理器的本地存储区。本地存储区包含了该处理器的 CPUID、定时器信号端点和中断信号端点，以及该处理器的运行队列。本函数将初始化这些数据结构。对于单核架构，只要调用该函数一次来初始化编译时静态定义的存储区就足够了；对于多核架构，则需要每个处理器调用一次来初始化各自的存储区。

Table 7-31 Parameters Needed for Initializing CPU-local Storage at Boot-time

原型	void _RME_CPU_Local_Init(struct RME_CPU_Local* CPU_Local, ptr_t CPUID)	
参数名称	类型	描述
CPU_Local	...	类型为 struct RME_CPU_Local*，是一个指向要初始化的 CPU 本地存储的指针。
CPUID	ptr_t	该 CPU 的 CPUID，标识了该 CPU 的编号。

该函数无返回值。

[1] 如 Cortex-M 等在编译时确定内核内存登记表大小的架构

[2] 如 x86-64 等在运行时确定内核内存登记表大小的架构

7.8.2.3 Creating Initial Capability Table at Boot-time

该函数用来在系统启动时创建第一个权能表，并且将指向这个权能表的权能放入该权能表中指定的权能槽位。这个函数与创建权能表的系统调用相比，其区别是只能在系统启动时使用，并且不需要一个上级权能表[1]。该函数不需要内核内存权能。此外，该函数创建的权能表大小不受 RME_CAPTBL_LIMIT 限制而仅受处理器字长限制。

Table 7-32 Parameters Needed for Creating Initial Capability Table at Boot-time

原型	ret_t _RME_Captbl_Boot_Init(cid_t Cap_Captbl, ptr_t Vaddr, ptr_t Entry_Num)	
参数名称	类型	描述
Cap_Captbl	cid_t	要接受产生的权能表权能的位置。该权能号只能是一级查找编码。
Vaddr	ptr_t	初始权能表要使用的内核空间起始虚拟地址。
Entry_Num	ptr_t	该权能表包含的表项数目，必须在 1 到 RME_CAPID_2L 之间。

该函数的返回值可能如下：

Table 7-33 Possible Return Values for Creating Initial Capability Table at Boot-time

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	传入的权能表权能数目参数超出了操作系统允许的范围。
	Cap_Crt 的一级查找超出了范围。
RME_ERR_CAP_KOTBL	分配内核内存失败。

这个函数只被调用一次。它会创建最初始的权能表。这个权能表在将来会用于放置在内核初始化过程中创建的其他权能。

7.8.2.4 Creating Other Capability Tables at Boot-time

该函数用来在系统启动时创建其他权能表，并且将指向这个权能表的权能放入指定的权能表中。这个函数与创建初始权能表的系统调用相比，其区别是它需要一个上级权能表来存放指向自己的权能，而并不会把指向自己的权能放入自己。该函数不需要内核内存权能。此外，该函数创建的权能表大小不受 RME_CAPTBL_LIMIT 限制而仅受处理器字长限制。

Table 7-34 Parameters Needed for Creating Other Capability Tables at Boot-time

原型	ret_t _RME_Captbl_Boot_Crt(struct RME_Cap_Captbl* Captbl,

[1] 因为此时系统中还没有其他权能表

		cid_t Cap_Captbl_Crt, cid_t Cap_Crt, ptr_t Vaddr, ptr_t Entry_Num)
参数名称	类型	描述
Captbl	...	类型为 struct RME_Cap_Captbl*，是一个指向上级权能表内核对象的实体指针。所有的权能号都是针对这个权能表而言的。
Cap_Captbl_Crt	cid_t	一个对应于必须拥有 RME_CAPTBL_FLAG_CRT 属性的权能表权能的权能号，该权能号对应的权能指向要接受此新创建的权能表权能的权能表。该权能号可以是一级或者二级查找编码。
Cap_Captbl	cid_t	要接受产生的权能表权能的位置。该权能号只能是一级查找编码。
Vaddr	ptr_t	新创建的权能表要使用的内核空间起始虚拟地址。
Entry_Num	ptr_t	该权能表包含的表项数目，必须在 1 到 RME_CAPID_2L 之间。

该函数的返回值可能如下：

Table 7-35 Possible Return Values for Creating Other Capability Tables at Boot-time

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	传入的权能表权能数目参数超出了操作系统允许的范围。
	Cap_Captbl_Crt 的一级/二级查找超出了范围。
	Cap_Crt 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl_Crt 的一级/二级查找的权能已经被冻结。
（不太可能返回该值）	Cap_Crt 被冻结，或者其它核正在该处创建权能。
RME_ERR_CAP_TYPE	Cap_Captbl_Crt 不是权能表权能。
RME_ERR_CAP_FLAG	Cap_Captbl_Crt 无 RME_CAPTBL_FLAG_CRT 属性。
RME_ERR_CAP_EXIST	Cap_Crt 不是空白权能。
RME_ERR_CAP_KOTBL	分配内核内存失败。

这个函数的调用数目视情况而定。如果除了初始的第一个由_RME_Captbl_Boot_Init 创建的权能表之外，我们还需要其他的权能表，那么就需要调用它。

7.8.2.5 Creating Page Directories at Boot-time

该函数用来在系统启动时创建页目录，并将这个指向页目录的权能放入指定的权能表内。该函数不需要内核内存权能。

Table 7-36 Parameters Needed for Creating Page Directories at Boot-time

原型	ret_t _RME_Pgtbl_Boot_Crt(struct RME_Cap_Captbl* Captbl, cid_t Cap_Captbl, cid_t Cap_Pgtbl, ptr_t Vaddr, ptr_t Start_Addr, ptr_t Top_Flag, ptr_t Size_Order, ptr_t Num_Order)	
参数名称	类型	描述
Captbl	...	类型为 struct RME_Cap_Captbl*，是一个指向上级权能表内核对象的实体指针。所有的权能号都是针对这个权能表而言的。
Cap_Captbl	cid_t	一个对应于必须拥有 RME_CAPTBL_FLAG_CRT 属性的权能表权能的权能号，该权能号对应的权能指向要接受此新创建的页目录权能的权能表。该权能号可以是一级或者二级查找编码。
Cap_Pgtbl	cid_t	一个对应于接受该新创建的页目录权能的权能表的某位置的权能号。该权能号对应的权能必须是空白的。该权能号只能是一级查找编码。
Vaddr	ptr_t	新创建的页目录要使用的内核空间起始虚拟地址。
Start_Addr	ptr_t	新创建的页目录的映射起始地址，最后一位为顶层标志，见下。
Top_Flag	ptr_t	该页目录是否是顶层页目录。“1”意味着该页目录为顶层。
Size_Order	ptr_t	该页目录的大小量级。
Num_Order	ptr_t	该页目录的数目量级。

该函数的返回值可能如下：

Table 7-37 Possible Return Values for Creating Page Directories at Boot-time

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Captbl 的一级/二级查找超出了范围。
	Cap_Pgtbl 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl 的一级/二级查找的权能已经被冻结。
（不太可能返回该值）	Cap_Pgtbl 被冻结，或者其它核正在该处创建权能。
RME_ERR_CAP_TYPE	Cap_Captbl 不是权能表权能。
RME_ERR_CAP_FLAG	Cap_Captbl 无 RME_CAPTBL_FLAG_CRT 属性。
RME_ERR_CAP_EXIST	Cap_Pgtbl 不是空白权能。
RME_ERR_CAP_KOTBL	分配内核内存失败。
RME_ERR_PGT_HW	底层硬件制约，不允许创建这样的页目录。

在启动时，需要多少个页目录，就创建多少个页目录。因此，该函数可能被调用多次，产生一组页目录。在通常启动过程中，只需要一个处理器完成这个功能即可，因此在整个系统中该函数只会被调用一组。

7.8.2.6 Constructing Page Directories at Boot-time

该函数用来在系统启动时构造页目录，将上一步创建的多个页目录组成一棵目录树[1]。在接下来的步骤中，我们会用初始的权能表和页表创造最初的进程。

Table 7-38 Parameters Needed for Constructing Page Directories at Boot-time

原型	ret_t _RME_Pgtbl_Boot_Con(struct RME_Cap_Captbl* Captbl, cid_t Cap_Pgtbl_Parent, ptr_t Pos, cid_t Cap_Pgtbl_Child)	
参数名称	类型	描述
Captbl	...	类型为 struct RME_Cap_Captbl*，是一个指向上级权能表权能的指针。所有的权能号都是针对这个权能表而言的。
Cap_Pgtbl_Parent	cid_t	一个对应于必须拥有 RME_PGTBL_FLAG_CON_PARENT 属性的页目录权能的权能号，该权能号对应的权能指向父页目录。该权能号可以是一级或者二级查找编码。
Pos	ptr_t	一个该目标页目录中要接受传递的目标页表项位置。该页表项必须是空白的。
Cap_Pgtbl_Child	cid_t	一个对应于必须拥有 RME_PGTBL_FLAG_CON_CHILD 属性的页目录权能的权能号，该权能号对应的权能指向子页目录。该权能号可以是一级或者二级查找编码。
Flags_Child	ptr_t	子页目录被映射时的属性。这个属性限制了该映射以下的所有页目录的访问权限。对于不同的架构，这个位置的值的意义也不相同。对于有些不支持页目录属性的架构而言[2]，这个值无效。

该函数的返回值可能如下：

Table 7-39 Possible Return Values for Constructing Page Directories at Boot-time

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Pgtbl_Parent 或 Cap_Pgtbl_Child 的一级/二级查找超出了范

[1] 也即页表

[2] 比如所有的基于 MPU 的系统

返回值	意义
	围。
RME_ERR_CAP_FROZEN （不太可能返回该值）	Cap_Pgtbl_Parent 或 Cap_Pgtbl_Child 的一级/二级查找的权能被冻结。
RME_ERR_CAP_TYPE	Cap_Pgtbl_Parent 或 Cap_Pgtbl_Child 不是页目录权能。
RME_ERR_PGT_ADDR	Pos 超出了父页目录的页表项数目。
	子页目录的总大小大于父页目录的一个页的大小。
	在开启了物理地址等于虚拟地址的检查时，映射的物理地址和目标虚拟地址有冲突。
RME_ERR_PGT_MAP	尝试构造，由于硬件原因失败。具体的失败原因与硬件有关，可能是硬件不支持此种映射。

在启动时，需要构造多少次页目录，就调用本函数多少次。在通常启动过程中，只需要一个处理器完成这个功能即可，因此在整个系统中该函数只会被调用一组。

7.8.2.7 Adding Pages to Page Directories at Boot-time

该函数用来在已经构建好的页目录中添加页，并且这一操作无视页目录是否允许添加操作。这些页在启动后会构成所有的用户地址可访问空间。这个函数是新增加物理内存页到系统中的唯一机会，未来用户地址可访问的内存空间只能从这些页中产生。并且，这些页在被映射时，还要求提供一个属性，在以后的页映射操作中，该物理内存页不可能拥有更多的属性。

Table 7-40 Parameters Needed for Adding Pages to Page Directories at Boot-time

原型	ret_t _RME_Pgtbl_Boot_Add(struct RME_Cap_Captbl* Captbl, cid_t Cap_Pgtbl, ptr_t Paddr, ptr_t Pos, ptr_t Flags)	
参数名称	类型	描述
Captbl	...	类型为 struct RME_Cap_Captbl*，是一个指向上级权能表权能的指针。所有的权能号都是针对这个权能表而言的。
Paddr	ptr_t	物理内存地址。
Pos	ptr_t	一个该页目录中要被填充的目标页表项位置。该页表项必须是空白的。
Flags	ptr_t	页表项的属性。这个属性限制了页表项的特性。

该函数的返回值可能如下：

Table 7-41 Possible Return Values for Adding Pages to Page Directories at Boot-time

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Pgtbl 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN （不太可能返回该值）	Cap_Pgtbl 的一级/二级查找的权能被冻结。
RME_ERR_CAP_TYPE	Cap_Pgtbl 不是页目录权能。
RME_ERR_PGT_ADDR	Pos 超出了页目录的页表项数目。
	在开启了物理地址等于虚拟地址的检查时，映射的物理地址和目标虚拟地址不同。
RME_ERR_PGT_MAP	尝试映射，由于硬件原因失败。具体的失败原因与硬件有关。

在启动时，需要添加多少个物理内存页，就调用本函数多少次。在通常启动过程中，只需要一个处理器完成这个功能即可，因此在整个系统中该函数只会被调用一组。同时，在这一过程中不要求每个物理内存页只能映射一次。如果映射了多次，那么这多个映射将会同时存在，并且都是合法的。

7.8.2.8 Creating the First Process at Boot-time

该函数用来在启动时创建第一个进程，并将这个指向进程的权能放入指定的权能表内。该函数不需要内核内存权能。

Table 7-42 Parameters Needed for Creating the First Process at Boot-time

原型	ret_t _RME_Proc_Boot_Crt(struct RME_Cap_Captbl* Captbl, cid_t Cap_Captbl_Crt, cid_t Cap_Proc, cid_t Cap_Captbl, cid_t Cap_Pgtbl, ptr_t Vaddr)	
参数名称	类型	描述
Captbl	...	类型为 struct RME_Cap_Captbl*，是一个指向上级权能表权能的指针。所有的权能号都是针对这个权能表而言的。
Cap_Captbl_Crt	cid_t	一个对应于必须拥有 RME_CAPTBL_FLAG_CRT 属性的权能表权能的权能号，该权能号对应的权能指向要接受此新创建的进程权能的权能表。该权能号可以是一级或者二级查找编码。
Cap_Proc	cid_t	一个对应于接受该新创建的进程权能的权能表的某位置的权能号。该权能号对应的权能必须是空白的。该权能号只能是一级查找编码。
Cap_Captbl	cid_t	一个对应于必须拥有 RME_CAPTBL_FLAG_PROC_CRT 属性的权能表权能的权能号，该权能号对应的权能指向要给新创建的进程使用的权能表。该权能号可以是一级或者二级查找编码。

Cap_Pgtbl	cid_t	一个对应于必须拥有 RME_PGTBL_FLAG_PROC_CRT 属性的页表权能的权能号，该权能号对应的权能指向要给新创建的进程使用的页表[1]。该权能号可以是一级或者二级查找编码。
Vaddr	ptr_t	新创建的进程内核对象要使用的内核空间起始虚拟地址。

该函数的返回值可能如下：

Table 7-43 Possible Return Values for Creating the First Process at Boot-time

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Captbl_Crt 的一级/二级查找超出了范围。
	Cap_Captbl 的一级/二级查找超出了范围。
	Cap_Pgtbl 的一级/二级查找超出了范围。
	Cap_Proc 的一级查找超出了范围。
RME_ERR_CAP_FROZEN（不太可能返回该值）	Cap_Captbl_Crt 的一级/二级查找的权能已经被冻结。
	Cap_Captbl 的一级/二级查找权能已经被冻结。
	Cap_Pgtbl 的一级/二级查找权能已经被冻结。
	Cap_Proc 被冻结，或者其它核正在该处创建权能。
RME_ERR_CAP_TYPE	Cap_Captbl_Crt 或 Cap_Captbl 不是权能表权能。
	Cap_Pgtbl 不是页表权能。
RME_ERR_CAP_FLAG	Cap_Captbl_Crt 无 RME_CAPTBL_FLAG_CRT 属性。
	Cap_Captbl 无 RME_CAPTBL_FLAG_PROC_CRT 属性。
	Cap_Pgtbl 无 RME_PGTBL_FLAG_PROC_CRT 属性。
RME_ERR_CAP_EXIST	Cap_Proc 不是空白权能。
RME_ERR_CAP_KOTBL	分配内核内存失败。
RME_ERR_CAP_REFCNT（不太可能返回该值）	Cap_Captbl 或 Cap_Pgtbl 的引用计数超过了系统允许的最大范围。

该函数在整个系统启动时只要由一个核调用一次即可。

[1] 顶层页目录

7.8.2.9 Creating Kernel Function Capability at Boot-time

该函数用来在系统启动时创建内核功能调用权能，并将这个内核功能调用权能放入指定的权能表内。内核功能调用权能只能在内核启动时完成创建，此后新产生的内核功能调用权能都是由此权能传递得到的。

Table 7-44 Parameters Needed for Creating Kernel Function Capability at Boot-time

原型	ret_t _RME_Kern_Boot_Crt(struct RME_Cap_Captbl* Captbl, cid_t Cap_Captbl, cid_t Cap_Kern)	
参数名称	类型	描述
Captbl	...	类型为 struct RME_Cap_Captbl*，是一个指向上级权能表权能的指针。所有的权能号都是针对这个权能表而言的。
Cap_Captbl	cid_t	一个对应于必须拥有 RME_CAPTBL_FLAG_CRT 属性的权能表权能的权能号，该权能号对应的权能指向要接受此新创建的内核功能调用权能的权能表。该权能号可以是一级或者二级查找编码。
Cap_Kern	cid_t	一个对应于接受该新创建的内核功能调用权能的权能表的某位置的权能号。该权能号对应的权能必须是空白的。该权能号只能是一级查找编码。

该函数的返回值可能如下：

Table 7-45 Possible Return Values for Creating Kernel Function Capability at Boot-time

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Captbl 的一级/二级查找超出了范围。
	Cap_Kern 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl 的一级/二级查找的权能已经被冻结。
（不太可能返回该值）	Cap_Kern 被冻结，或者其它核正在该处创建权能。
RME_ERR_CAP_TYPE	Cap_Captbl 不是权能表权能。
RME_ERR_CAP_FLAG	Cap_Captbl 无 RME_CAPTBL_FLAG_CRT 属性。
RME_ERR_CAP_EXIST	Cap_Kern 不是空白权能。

该函数在整个系统启动时只要由一个核调用一次即可。

7.8.2.10 Creating Kernel Memory Capability at Boot-time

该函数用来在系统启动时创建内核内存权能，并将这个内核内存权能放入指定的权能表内。内核内存权能只能在内核启动时完成创建，此后新产生的内核内存权能都是由此权能传递得到的。

Table 7-46 Parameters Needed for Creating Kernel Memory Capability at Boot-time

原型	ret_t _RME_Kmem_Boot_Crt(struct RME_Cap_Captbl* Captbl, cid_t Cap_Captbl, cid_t Cap_Kmem, ptr_t Start, ptr_t End, ptr_t Flags)	
参数名称	类型	描述
Captbl	...	类型为 struct RME_Cap_Captbl*，是一个指向上级权能表权能的指针。所有的权能号都是针对这个权能表而言的。
Cap_Captbl	cid_t	一个对应于必须拥有 RME_CAPTBL_FLAG_CRT 属性的权能表权能的权能号，该权能号对应的权能指向要接受此新创建的内存权能的权能表。该权能号可以是一级或者二级查找编码。
Cap_Kmem	cid_t	一个对应于接受该新创建的内核内存权能的权能表的某位置的权能号。该权能号对应的权能必须是空白的。该权能号只能是一级查找编码。
Start	ptr_t	内核内存的起始虚拟地址。该地址在传入时会被自动对齐到内核内存登记表的粒度。
End	ptr_t	内核内存的终止虚拟地址。该地址在传入时会被自动对齐到内核内存登记表的粒度-1。
Flags	ptr_t	该内核内存权能的标志位，指明允许创建哪些内核对象在这段内存上。该值不能为 0，否则内核会直接崩溃。

该函数的返回值可能如下：

Table 7-47 Possible Return Values for Creating Kernel Memory Capability at Boot-time

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Captbl 的一级/二级查找超出了范围。
	Cap_Kmem 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl 的一级/二级查找的权能已经被冻结。
（不太可能返回该值）	Cap_Kmem 被冻结，或者其它核正在该处创建权能。
RME_ERR_CAP_TYPE	Cap_Captbl 不是权能表权能。
RME_ERR_CAP_FLAG	Cap_Captbl 无 RME_CAPTBL_FLAG_CRT 属性。
RME_ERR_CAP_EXIST	Cap_Kmem 不是空白权能。

通常而言该函数在整个系统启动时只要由一个核调用一次即可。如果系统的可用内核内存分成很多段，或者各个段有不同的性质，那么可能会有多个内核内存权能被创建。

7.8.2.11 Creating Kernel Signal Endpoints at Boot-time

该函数用来在启动时创建内核信号端点。内核信号端点用来处理中断，在中断向量中通过发送信号到内核信号端点来唤醒对应的用户态线程进行中断处理。由于任何一个内核信号端点在任何时刻只能有一个线程阻塞在它上面，因此需要创建的内核信号端点的数量为“中断向量-处理线程”对的数量。

内核信号端点只能在内核启动时完成创建，并且不可删除。此后新产生的内核信号端点都是由此权能传递得到的。该函数不需要内核内存权能。

Table 7-48 Parameters Needed for Creating Kernel Signal Endpoints at Boot-time

原型	ret_t _RME_Sig_Boot_Crt(struct RME_Cap_Captbl* Captbl, cid_t Cap_Captbl, cid_t Cap_Sig, ptr_t Vaddr)	
参数名称	类型	描述
Captbl	...	类型为 struct RME_Cap_Captbl*，是一个指向上级权能表权能的指针。所有的权能号都是针对这个权能表而言的。
Cap_Captbl	cid_t	一个对应于必须拥有 RME_CAPTBL_FLAG_CRT 属性的权能表权能的权能号，该权能号对应的权能指向要接受此新创建的内核信号端点权能的权能表。该权能号可以是一级或者二级查找编码。
Cap_Sig	cid_t	一个对应于接受该新创建的内核信号端点权能的权能表的某位置的权能号。该权能号对应的权能必须是空白的。该权能号只能是一级查找编码。
Vaddr	ptr_t	新创建的内核信号端点内核对象要使用的内核空间起始虚拟地址。

该函数的返回值可能如下：

Table 7-49 Possible Return Values for Creating Kernel Signal Endpoints at Boot-time

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Captbl 的一级/二级查找超出了范围。
	Cap_Sig 的一级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Captbl 的一级/二级查找的权能已经被冻结。
（不太可能返回该值）	Cap_Sig 被冻结，或者其它核正在该处创建权能。
RME_ERR_CAP_TYPE	Cap_Captbl 不是权能表权能。
RME_ERR_CAP_FLAG	Cap_Captbl 无 RME_CAPTBL_FLAG_CRT 属性。
RME_ERR_CAP_EXIST	Cap_Sig 不是空白权能。
RME_ERR_CAP_KOTBL	分配内核内存失败。

该函数的调用方法有两种模式。在第一种模式下，由一个处理器核创建所有的内核信号端点。在第二种模式下，由各个处理器分开创建各自需要使用的内核信号端点。第二种模式快一些，但是程序相对更加复杂。通常，使用第一种方法就足够了。

在多核系统下如果使用第一种模式，那么需要在完成这一步之后，给其他处理器发送 IPI，让其他处理器各自都执行接下来的函数，完成各自的初始化。如果使用第二种模式，那么在执行这一步之前就要分开。这一步之后的所有步骤都需要各自处理器分开执行一次。

对于每个核至少要创建两个信号端点：第一个信号端点是用来接收定时器信号的，指向它的指针应当被赋给该 CPU 的本地存储的 Tick_Sig 成员；第二个端点是用来在默认情况下接收所有的其他外设中断的，指向它的指针应当被赋给该 CPU 的本地存储的 Int_Sig 成员。

7.8.2.12 Creating Initial Threads at Boot-time

该函数用来在系统启动时创建初始线程，也即 Init 线程。Init 线程一经创建就会被绑定到某处理器，并且拥有无限的时间片。Init 进程中，每个处理器核都拥有一个 Init 线程。Init 线程不可被杀死，不能被从该处理器解除绑定，而且不能被在任何一个信号端点上被阻塞，但其优先级是可以更改的。Init 线程的优先级上限由系统指定为 RME_MAX_PREEMPT_PRIO-1。Init 线程的 TID 总是 0。

与创建线程的系统调用不同，这个函数允许通过 CPU_Local 参数来指定该线程被绑定到何处理器，这是为了在多处理器体系中方便启动处理器核创建所有的内核对象，这也是推荐的做法。该函数不需要内核内存权能。

Table 7-50 Parameters Needed for Creating Initial Threads at Boot-time

原型	ret_t _RME_Thd_Boot_Crt(struct RME_Cap_Captbl* Captbl, cid_t Cap_Captbl, cid_t Cap_Thd, cid_t Cap_Proc, ptr_t Vaddr, ptr_t Prio, struct RME_CPU_Local* CPU_Local)	
参数名称	类型	描述
Captbl	...	类型为 struct RME_Cap_Captbl*，是一个指向上级权能表权能的指针。所有的权能号都是针对这个权能表而言的。
Cap_Captbl	cid_t	一个对应于必须拥有 RME_CAPTBL_FLAG_CRT 属性的权能表权能的权能号，该权能号对应的权能指向要接受此新创建的初始线程权能的权能表。该权能号可以是一级或者二级查找编码。
Cap_Thd	cid_t	一个对应于接受该新创建的线程权能的权能表的某位置的权能号。该权能号对应的权能必须是空白的。该权能号只能是一级查找编码。
Cap_Proc	cid_t	一个对应于必须拥有 RME_PROC_FLAG_THD 属性的进程权能的权能号，该权能号对应的权能指向包含新创建的线程的进程。该权能号可以是一级或者二级查找编码。

Vaddr	ptr_t	新创建的初始线程内核对象要使用的内核空间起始虚拟地址。
Prio	ptr_t	初始线程的的抢占优先级。在 RME 中线程的优先级从 0 开始计算，值越大优先级越高。这个值不能超过系统允许的最大值。
CPU_Local	...	类型为 struct RME_CPU_Local*，是一个指向要将该线程绑定到的 CPU 的 CPU 本地存储的指针。

该函数的返回值可能如下：

Table 7-51 Possible Return Values for Creating Initial Threads at Boot-time

返回值	意义
非负值	操作成功，返回线程标识符（TID）。
RME_ERR_CAP_RANGE	Cap_Captbl 的一级/二级查找超出了范围。
	Cap_Proc 的一级/二级查找超出了范围。
	Cap_Thd 的一级查找超出了范围。
RME_ERR_CAP_FROZEN（不太可能返回该值）	Cap_Captbl 的一级/二级查找的权能已经被冻结。
	Cap_Proc 的一级/二级查找权能已经被冻结。
	Cap_Thd 被冻结，或者其它核正在该处创建权能。
RME_ERR_CAP_TYPE	Cap_Captbl 不是权能表权能。
	Cap_Proc 不是进程权能。
RME_ERR_CAP_FLAG	Cap_Captbl 无 RME_CAPTBL_FLAG_CRT 属性。
	Cap_Proc 无 RME_PROC_FLAG_THD 属性。
RME_ERR_CAP_EXIST	Cap_Thd 不是空白权能。
RME_ERR_CAP_KOTBL	分配内核内存失败。
RME_ERR_PTH_PRIO	指定的初始线程优先级超过了 RME_MAX_PREEMPT_PRIO-1。

该函数需要每个处理器调用一次，在该处理器上创建绑定到自身的 Init 线程。当然，也可以由一个处理器为所有的其他处理器创建 Init 线程。

7.8.2.13 Set Current Page Table, Enable Interrupts and Enter User Level

各处理器依次调用一次__RME_Pgtbl_Set、__RME_Enable_Int 和__RME_Enter_User_Mode，进入用户态开始执行。关于__RME_Pgtbl_Set 的相关信息，请参见 7.11；关于__RME_Enable_Int 和__RME_Enter_User_Mode 的相关信息，请参见 7.4。

7.9 Porting of Register Set Related Functions

RME 中，和寄存器组有关的函数有以下 10 个。这 10 个函数都是非常短小的，仅涉及寄存器上下文。这些函数的实现往往和架构相关结构体有关系，和用户态库使用这些寄存器的方法也有关系。

7.9.1 Implementation of __RME_Get_Syscall_Param

该函数用于从寄存器组中提取系统调用的参数。

Table 7-52 Implementation of __RME_Get_Syscall_Param

原型	void __RME_Get_Syscall_Param(struct RME_Reg_Struct* Reg, ptr_t* Svc, ptr_t* Capid, ptr_t* Param)
意义	提取系统调用的参数，并放入分别的各个返回值。
返回值	无。
参数	struct RME_Reg_Struct* Reg 指向寄存器组的指针。
	ptr_t* Svc 该参数用于输出，输出半字长的系统调用号（N）。
	ptr_t* Capid 该参数用于输出，输出半字长的权能号（C）。
	ptr_t* Param 该参数用于输出，输出三个字长的参数（P1-P3）。

7.9.2 Implementation of __RME_Set_Syscall_Retval

该函数用于向寄存器组中存入系统调用的返回值。

Table 7-53 Implementation of __RME_Set_Syscall_Retval

原型	void __RME_Set_Syscall_Retval(struct RME_Reg_Struct* Reg, ret_t Retval)
意义	将系统调用的返回值存入寄存器组。
返回值	无。
参数	struct RME_Reg_Struct* Reg 该参数用于输出，是指向寄存器组的指针。
	ret_t Retval 系统调用返回的返回值。

7.9.3 Implementation of __RME_Thd_Reg_Init

该函数用于初始化线程或迁移调用的寄存器组。在线程设置执行属性和迁移调用被调用时，该函数都会被调用。

Table 7-54 Implementation of __RME_Thd_Reg_Init

原型	void __RME_Thd_Reg_Init(ptr_t Entry, ptr_t Stack, ptr_t Param, struct RME_Reg_Struct* Reg)
意义	使用入口地址，栈地址和参数初始化线程或迁移调用寄存器组。
返回值	无。
参数	ptr_t Entry 线程的入口地址。
	ptr_t Stack 线程栈的地址。
	ptr_t Param 要赋给线程的参数。
	struct RME_Reg_Struct* Reg 该参数用于输出，是指向该线程寄存器组结构体的指针。

7.9.4 Implementation of __RME_Thd_Reg_Copy

该函数用于复制线程的寄存器组。有时候需要用汇编实现这个函数以提高效率。

Table 7-55 Implementation of __RME_Thd_Reg_Copy

原型	void __RME_Thd_Reg_Copy(struct RME_Reg_Struct* Dst, struct RME_Reg_Struct* Src)
意义	将一个寄存器组数据结构复制到另一个。
返回值	无。
参数	struct RME_Reg_Struct* Dst 该参数用于输出，是指向目标寄存器组数据结构的指针。
	struct RME_Reg_Struct* Reg 指向源寄存器组数据结构的指针。

7.9.5 Implementation of __RME_Thd_Cop_Init

该函数仅被用于在设置线程执行属性时初始化线程的协处理器寄存器组。在某些系统上，某些协处理器也需要被初始化，但是这在绝大多数系统上都是用不到的。

Table 7-56 Implementation of __RME_Thd_Cop_Init

原型	void __RME_Thd_Cop_Init(struct RME_Reg_Struct* Reg, struct RME_Cop_Struct* Cop_Reg)
意义	初始化线程协处理器寄存器组。
返回值	无。
参数	struct RME_Reg_Struct* Reg 指向寄存器组数据结构的指针。这个参数是用来辅助协处理器寄存器初始化用的。
	struct RME_Cop_Struct* Cop_Reg 该参数用于输出，是指向协处理器寄存器组的指针。

7.9.6 Implementation of __RME_Thd_Cop_Save

该函数用于保存线程的协处理器寄存器组。有时候需要用汇编实现这个函数以提高效率。

Table 7-57 Implementation of __RME_Thd_Cop_Save

原型	void __RME_Thd_Cop_Save(struct RME_Reg_Struct* Reg, struct RME_Cop_Struct* Cop_Reg)
意义	保存线程的协处理器寄存器组。
返回值	无。
参数	struct RME_Reg_Struct* Reg 指向寄存器组数据结构的指针。这个参数是用来辅助判断是否需要保存协处理器寄存器组用的。对于某些处理器，协处理器是否被使用会体现在程序状态字或某个特殊寄存器中，此时即可通过该字判断是否需要保存协处理器状态。
	struct RME_Cop_Struct* Cop_Reg 指向协处理器寄存器组的指针。

7.9.7 Implementation of __RME_Thd_Cop_Restore

该函数用于恢复线程的协处理器寄存器组。有时候需要用汇编实现这个函数以提高效率。

Table 7-58 Implementation of __RME_Thd_Cop_Restore

原型	void __RME_Thd_Cop_Restore(struct RME_Reg_Struct* Reg, struct RME_Cop_Struct* Cop_Reg)
意义	恢复线程的协处理器寄存器组。
返回值	无。
参数	struct RME_Reg_Struct* Reg

	指向寄存器组数据结构的指针。这个参数是用来辅助判断是否需要恢复协处理器寄存器组用的。对于某些处理器，协处理器是否被使用会体现在程序状态字或某个特殊寄存器中，此时即可通过该字判断是否需要恢复协处理器状态。
	struct RME_Cop_Struct* Cop_Reg 该参数用于输出，是指向协处理器寄存器组的指针。

需要特别注意的是，协处理器寄存器组有时可以被当作一个传输能力很强的隐蔽通道使用。因此，在那些注重信息安全的实现中，如果检测到当前线程没有使用协处理器寄存器组，那么应当使用无意义的字符填充协处理器寄存器组；也可以无视线程是否使用了协处理器寄存器组，总是保存和恢复协处理器寄存器组。

7.9.8 Implementation of __RME_Inv_Reg_Save

该函数用于保存必要的寄存器到线程迁移调用结构体中以方便返回。只需要保存那些对恢复程序执行流必要的寄存器就可以了[1]。

Table 7-59 Implementation of __RME_Inv_Reg_Save

原型	void __RME_Inv_Reg_Save(struct RME_Iret_Struct* Ret, struct RME_Reg_Struct* Reg)
意义	保存必要的寄存器到线程迁移调用结构体中。
返回值	无。
参数	struct RME_Iret_Struct* Ret 该参数用于输出，是指向必要寄存器结构体的指针。
	struct RME_Reg_Struct* Reg 指向寄存器组的指针。

7.9.9 Implementation of __RME_Inv_Reg_Restore

该函数用于从线程迁移调用结构体恢复必要的寄存器以返回。只需要恢复那些对恢复程序执行流必要的寄存器就可以了[2]。

Table 7-60 Implementation of __RME_Inv_Reg_Restore

原型	void __RME_Inv_Reg_Restore(struct RME_Reg_Struct* Reg, struct RME_Iret_Struct* Ret)

[1] 比如对于 x86-64，要保存 SP 和 IP；对于 Cortex-M，则要保存 LR 和 SP

[2] 比如对于 x86-64，要恢复 SP 和 IP；对于 Cortex-M，则要恢复 LR 和 SP

意义	从线程迁移调用结构体中恢复必要的寄存器。
返回值	无。
参数	struct RME_Reg_Struct* Reg 该参数用于输出，是指向寄存器组的指针。
	struct RME_Iret_Struct* Ret 指向必要寄存器结构体的指针。

7.9.10 Implementation of __RME_Set_Inv_Retval

该函数用于向寄存器组中存入线程迁移调用的返回值。

Table 7-61 Implementation of __RME_Set_Inv_Retval

原型	void __RME_Set_Inv_Retval(struct RME_Reg_Struct* Reg, ret_t Retval)
意义	将线程迁移调用的返回值存入寄存器组。
返回值	无。
参数	struct RME_Reg_Struct* Reg 该参数用于输出，是指向寄存器组的指针。
	ret_t Retval 线程迁移调用返回的返回值。

7.10 Porting of Kernel Functions

内核功能调用函数是一组由用户实现的、可以在操作系统内核态运行的一系列函数。这些函数是在编译时确定的。这些函数的描述如下：

Table 7-62 Implementation of Kernel Functions

原型	ptr_t __User_Func(struct RME_Reg_Struct* Reg, ptr_t Sub_ID, ptr_t Param1, ptr_t Param2)
意义	实现一个用户定义的内核态操作。
返回值	ptr_t 如果失败，必须返回负值；如果成功，必须返回非负值。此外，如果该函数成功，由该函数负责设置其返回值到寄存器组。
参数	struct RME_Reg_Struct* Reg 该参数可用于输入或输出，是指向寄存器组的指针。
	ptr_t Sub_ID 子功能号。

ptr_t Param1	该函数的第一个参数。
ptr_t Param2	该函数的第二个参数。

这是一个接受两个用户自定义参数，完成一些操作，然后返回的内核态函数。通常而言这些函数被用于实现一些处理器特定的功能[1]。这些函数的实现都应该短小精悍，并且应当保证能在一定时限之内完成，否则调用这些函数的实时性就没有保证。

接下来介绍几个最常见的内核功能的实现思路以供参考。

7.10.1 Tickless Kernel Implementation

无节拍内核通常要求系统具备一个只能在内核态下进行设置的高精度定时器，并且由该高精度定时器产生系统的调度器时间中断。具体的实现随着各个处理器是非常不同的，但是实现的思路是大同小异的。

在无节拍内核中，时钟中断向量不使用 RME 提供的周期性时钟中断处理函数，而是仅在该向量中对一个内核信号端点进行发送操作，并且同步增加 RME_Timestamp 的值。收到该端点信号的调度器工作在系统的最高优先级上，并且由它决定系统的调度情况。

无节拍内核的最长无节拍时间上限可以根据系统的要求灵活实现。需要注意的是，无节拍时间的上限不能太高，否则 RME_Timestamp 会有很久得不到更新，这样反而会影响权能的创建、冻结、删除、移除等操作。从工程实际出发，推荐的最长上限为 200ms 以内。

7.10.2 High Precision Timer System Implementation

高精度定时器的实现和无节拍内核的实现是类似的，只需要设计几个内核功能调用，并且赋予他们操作定时器的功能即可。定时器产生的中断可以直接通过内核信号端点传递到对应的目标线程，也可以由另外一个管理线程负责处理，然后再把定时器中断传递给其他线程。

7.10.3 Inter-Processor-Interrupt Implementation

由于 RME 中，从一个 CPU 发出的异步信号无法直接被传送到另外一个 CPU，从而唤醒其上的线程，因此需要一个内核功能调用来实现处理器间中断，并且提示另一个核上的某个线程需要唤醒某其他线程。

7.10.4 Cache Maintenance Operation Implementation

处理器的缓存操作一般也是特权指令。因此，可以把这些操作分别用内核功能调用实现。

7.11 Porting of Page Table Related Functions

[1] 比如某些内建于 CPU 的外设、特殊协处理器指令或者其他必须在内核态实现的 I/O 操作

RME 中，和页表相关的函数有以下 11 个。这些函数的实现和处理器架构紧密相关，而且在多核环境下还要负责检查并行操作的冲突。这些函数的安全性和可靠性会极大地影响系统的安全性和可靠性，因此是系统移植中最重要的一环。接下来我们分别解释这些函数的功能和移植注意事项。

7.11.1 Implementation of __RME_Pgtbl_Set

该函数负责设置处理器当前使用的页表。该函数传入的是一个虚拟地址；在该函数中往往需要先进行虚拟地址到物理地址的转换，然后再将物理地址赋给处理器的相应寄存器。

Table 7-63 Implementation of __RME_Pgtbl_Set

原型	void __RME_Pgtbl_Set(ptr_t Pgtbl)
意义	设置处理器使用该页表。
返回值	无。
参数	ptr_t Pgtbl 指向能被处理器硬件直接识别的页表数据结构的内核虚拟地址。

对于 MMU 架构，该函数会将顶层页目录指针寄存器[1]指向顶层页目录，这一操作也会同时刷新 TLB 缓存。对于 MPU 架构，该函数会将顶层页表的元数据复制进 MPU 的相关寄存器中完成保护区域设置。由于 MPU 的寄存器相对较多，因此可考虑用汇编实现该函数，从而达到快速设置页表的效果。

7.11.2 Implementation of __RME_Pgtbl_Kmem_Init

该函数负责在系统启动时建立初始的内核页表。

Table 7-64 Implementation of __RME_Pgtbl_Kmem_Init

原型	ptr_t __RME_Pgtbl_Kmem_Init(void)
意义	建立内核初始页表。
返回值	ptr_t 成功返回 0，失败返回 RME_ERR_PGT_OPFAIL（-1）。
参数	无。

该函数建立的内核映射一经成立，就不会被用户变更，而且这些物理地址将在系统存在期间永续地被作为内核内存来使用。这个内核页表[2]将会被映射进每一个进程的顶层页目录，并且其特权属性将会被定义为内核级别。在初始内核页表中应当包括两个部分，一个部分是内核所占虚拟空间，另一个部分则是内核虚拟机使用的内存的空间。关于该种映射进行的时间，请参看有关__RME_Pgtbl_Init 的部分。

[1] 如 x86-64 中的 CR3

[2] 或者这些内核页目录

在 MPU 环境下，这个函数一般直接返回成功就可以了。因为一般情况下，在特权模式下的处理器可以访问所有的内存空间，无需往页表和页表元数据中加入关于内核地址的条目。

7.11.3 Implementation of __RME_Pgtbl_Check

该函数负责检查用来创建页目录的各个参数是否能够被底层架构支持。

Table 7-65 Implementation of __RME_Pgtbl_Check

原型	ptr_t __RME_Pgtbl_Check(ptr_t Start_Addr, ptr_t Top_Flag, ptr_t Size_Order, ptr_t Num_Order, ptr_t Vaddr)
意义	检查传入的页目录创建参数是否能够被底层硬件支持。
返回值	ptr_t 成功（硬件支持）返回 0，失败（硬件不支持）返回 RME_ERR_PGT_OPFAIL（-1）。
参数	ptr_t Start_Addr 页目录映射起始虚拟地址。该参数仅在 MPU 环境中有效。
	ptr_t Top_Flag 页目录是否为顶层页目录。1 为顶层，0 则不为顶层。
	ptr_t Size_Order 页目录的页表项大小级数。
	ptr_t Num_Order 页目录的页表项数量级数。
	ptr_t Vaddr 页目录内核对象自身位于的内核虚拟地址。

这个函数会在创建页表的内核调用之前被调用，用来确认该种页表能够被创建，从而先在分配内核内存之前检查页表参数的有效性。该函数需要按照处理器硬件对页表的要求严格编写，使它只能对处理器支持的页表形式返回 0，对于其他的组合都要返回 RME_ERR_PGT_OPFAIL（-1）。

7.11.4 Implementation of __RME_Pgtbl_Init

该函数负责初始化一个刚刚创建的页目录。

Table 7-66 Implementation of __RME_Pgtbl_Init

原型	ptr_t __RME_Pgtbl_Init(struct RME_Cap_Pgtbl* Pgtbl_Op)
意义	初始化刚刚创建的页目录。
返回值	ptr_t

	成功返回 0，失败返回 RME_ERR_PGT_OPFAIL（-1）。
参数	struct RME_Cap_Pgtbl* Pgtbl_Op 指向该页目录的，含有该页目录的所有信息的页目录权能。

这个函数会在页目录创建时被调用，也是页目录创建的关键函数。该函数要把页目录初始化成可用的形式，比如将其所有的空隙全部初始化成空表项等。此外，如果是在 MMU 环境下创建顶层页目录，还需要把所有的由__RME_Pgtbl_Kmem_Init 创建的内核表项全部都映射到该页目录中去。由于之前由__RME_Pgtbl_Check 检查过页目录的参数，因此本函数可以略过这些检查。

7.11.5 Implementation of __RME_Pgtbl_Del_Check

该函数负责检查一个页目录能否被安全删除。

Table 7-67 Implementation of __RME_Pgtbl_Del_Check

原型	ptr_t __RME_Pgtbl_Del_Check(struct RME_Cap_Pgtbl* Pgtbl_Op)
意义	检查一个页目录能否被安全删除。
返回值	ptr_t 成功（可以删除）返回 0，失败（不能删除）返回 RME_ERR_PGT_OPFAIL（-1）。
参数	struct RME_Cap_Pgtbl* Pgtbl_Op 指向该页目录的，含有该页目录的所有信息的页目录权能。

这个函数是删除页目录操作中必备的检查函数。在这个函数中，我们需要检查该级页目录有没有被上一级页目录引用。如果有的话，那么不能直接删除该页目录。此外，还需要检查，这个页目录中是否含有下一级页目录的引用。如果有，那么这一级页目录也不能够被直接删除。

如果本函数仅检查了其中的一项，或者两项都没有检查而直接返回成功，那么删除页目录操作的正确性就必须由用户库保证。用户必须保证在删除一个页目录时不会出现该页目录被引用或者该页目录仍然含有引用的状况。如果在该状况下，用户库也不检查这些项目，那么内核的数据完整性就会遭到破坏。

7.11.6 Implementation of __RME_Pgtbl_Page_Map

该函数负责映射一个页到一个页目录内。如果这种映射由于传入的参数不正确[1]不能被完成，应当返回错误。

Table 7-68 Implementation of __RME_Pgtbl_Page_Map

原型	ptr_t __RME_Pgtbl_Page_Map(struct RME_Cap_Pgtbl* Pgtbl_Op,

[1] 比如位号超标、物理地址对齐不符合要求、有不支持的标志位、或者该位置已经有映射

	ptr_t Paddr, ptr_t Pos, ptr_t Flags)
意义	映射一个页到页目录内部。
返回值	ptr_t 成功返回 0，失败返回 RME_ERR_PGT_OPFAIL（-1）。
参数	struct RME_Cap_Pgtbl* Pgtbl_Op 指向该页目录的，含有该页目录的所有信息的页目录权能。
	ptr_t Paddr 需要被映射的物理页框地址。
	ptr_t Pos 需要将该页映射到的页目录表项位号。
	ptr_t Flags 该页的 RME 标准页标志。

在上表中，“页目录表项位号”指的是被映射的页在页目录中的槽位号。比如一个页目录的每一项都代表了 4kB 大小的一个页框，那么 12kB 处就是其第 3 个槽位的起始点[1]。“RME 标准页标志” 是 RME 系统使用的抽象页标志，不是具体页表中使用的那些页标志，具体请参见第三章描述。该函数需要将这些页标志转换为处理器能直接识别的页表项的页标志，然后再写入页表。对于那些不支持部分页标志的处理器，那些不被支持的页标志可以直接被忽略。比如，对于那些硬件更新 TLB 的 MMU 架构，“静态（RME_PGTBL_STATIC）”页标志就可以不实现。

在多核环境下，本函数需要保证两个 CPU 不会同时向一个位置处同时映射两个页。如果发生了这种情况，本函数可以使用读-改-写[2]原子操作，保证多核环境下这样的冲突不会发生。在 MPU 环境下，该函数还要负责更新 MPU 的顶层页表元数据，加入该页的映射。

7.11.7 Implementation of __RME_Pgtbl_Page_Unmap

该函数负责解除页目录内一个页的映射。如果该操作由于传入的参数不正确[3]，那么应当返回错误。

Table 7-69 Implementation of __RME_Pgtbl_Page_Unmap

原型	ptr_t __RME_Pgtbl_Page_Unmap(struct RME_Cap_Pgtbl* Pgtbl_Op, ptr_t Pos)
意义	从一个页目录中解除一个页的映射。
返回值	ptr_t 成功返回 0，失败返回 RME_ERR_PGT_OPFAIL（-1）。

[1] 槽位号从 0 开始计算

[2] 也即比较交换，CAS

[3] 比如位号超标或者位号的位置没有页存在

参数	struct RME_Cap_Pgtbl* Pgtbl_Op 指向该页目录的，含有该页目录的所有信息的页目录权能。
	ptr_t Pos 需要解除映射的页目录表项位号。

这个函数是上面函数的逆操作，只要解除该页映射就可以了。在多核环境下，也需要保证当两个 CPU 同时试图解除映射时，冲突不会发生。在 MMU 环境下，该函数还要负责使用 TLB 刷新指令，刷新整个 TLB 缓存，或者也可以在确知该页映射的位置的状况下使用 TLB 单条刷新操作[1]。不刷新缓存或者是定时使用特殊内核功能调用刷新缓存也是可以的，此时需要在用户态进行页面映射安定化处理[2]。在 MPU 环境下，则需要负责更新 MPU 的顶层元数据，去掉该页的映射。

7.11.8 Implementation of __RME_Pgtbl_Pgdir_Map

该函数负责映射一个子页目录到父页目录内。如果该操作由于传入的参数不正确[3]，那么应当返回错误。

Table 7-70 Implementation of __RME_Pgtbl_Pgdir_Map

原型	ptr_t __RME_Pgtbl_Pgdir_Map(struct RME_Cap_Pgtbl* Pgtbl_Parent, ptr_t Pos, struct RME_Cap_Pgtbl* Pgtbl_Child, ptr_t Flags)
意义	映射一个子页目录到父页目录内部。
返回值	ptr_t 成功返回 0，失败返回 RME_ERR_PGT_OPFAIL（-1）。
参数	struct RME_Cap_Pgtbl* Pgtbl_Parent 指向父页目录的，含有父页目录的所有信息的页目录权能。
	ptr_t Pos 需要将该子页目录映射到的父页目录表项位号。
	struct RME_Cap_Pgtbl* Pgtbl_Child 指向子页目录的，含有子页目录的所有信息的页目录权能。
	ptr_t Flags 该子页目录的 RME 标准页标志。该页标志决定了子页目录及以下各层页目录的访问权限限制。

[1] 如 x86-64 的 INVLTB 等

[2] 也即被除去映射的页面要等待一会才会真的从 TLB 中消失

[3] 比如位号超标、物理地址对齐不符合要求、虚拟地址的关系不正确、该位置已经有映射、传入的标志位不正确或者在 MPU 环境下某些特殊约束不满足

对于那些不允许设置页目录属性的架构或者基于 MPU 的架构，该值无效。

该函数在 MMU 系统下和 MPU 系统下往往有不同的表现。在 MMU 系统下，这种映射不需要检查起始虚拟地址是否合规，但是需要子页目录包含的地址范围正好是父页目录的一个槽位的大小。在 MPU 系统下，由于可以使用压缩页表，因此子页目录包含的地址范围可以比父页目录的一个槽位小，但是需要保证其起始虚拟地址是合规的。

在 MPU 系统下，由于 MPU 的某些固有属性[1]，因此要求父页目录必须具备[2]顶层页目录，要求子页目录必须自己不是顶层页目录，也不具备顶层页目录。此外，在映射完成后，如果子页目录中含有已映射的页，那么需要更新顶层页目录处包含的 MPU 元数据，添加这些页的映射。

7.11.9 Implementation of __RME_Pgtbl_Pgdir_Unmap

该函数负责解除父页目录内一个子页目录的映射。如果该操作由于传入的参数不正确[3]，那么应当返回错误。

Table 7-71 Implementation of __RME_Pgtbl_Pgdir_Unmap

原型	ptr_t __RME_Pgtbl_Pgdir_Unmap(struct RME_Cap_Pgtbl* Pgtbl_Op, ptr_t Pos)
意义	解除父页目录内一个子页目录的映射。
返回值	ptr_t 成功返回 0，失败返回 RME_ERR_PGT_OPFAIL（-1）。
参数	struct RME_Cap_Pgtbl* Pgtbl_Op 指向父页目录的，含有父页目录的所有信息的页目录权能。
	ptr_t Pos 需要解除映射的子页目录表项位号。

这个函数是上面函数的逆操作，只要解除子页目录映射就可以了。在多核环境下，也需要保证当两个 CPU 同时试图解除映射时，冲突不会发生。在 MMU 环境下，该函数还要负责使用 TLB 刷新指令，刷新整个 TLB 缓存。不刷新缓存而使用上面提到的页面映射安定化处理也是可以的。在 MPU 环境下，则需要负责更新 MPU 的顶层元数据，去掉子页目录中含有的页的映射。

7.11.10 Implementation of __RME_Pgtbl_Lookup

该函数负责查找一个页目录内的某个页的信息。

[1] 请参见 3.2.5

[2] 或者自身就是

[3] 比如位号超标、位号的位置没有子页目录存在，或者在 MPU 环境下某些特殊约束不满足

Table 7-72 Implementation of __RME_Pgtbl_Lookup

原型	ptr_t __RME_Pgtbl_Lookup(struct RME_Cap_Pgtbl* Pgtbl_Op, ptr_t Pos, ptr_t* Paddr, ptr_t* Flags)
意义	查找一个页目录内某个位号上的页的信息并且返回之。
返回值	ptr_t 成功[1]返回 0，失败[2]返回 RME_ERR_PGT_OPFAIL（-1）。
参数	struct RME_Cap_Pgtbl* Pgtbl_Op 指向该页目录的，含有该页目录的所有信息的页目录权能。
	ptr_t Pos 需要查找的页目录表项位号。
	ptr_t* Paddr 该参数用于输出，是指向该页的物理页框地址的指针。
	ptr_t* Flags 该参数用于输出，是指向该页的 RME 标准页标志的指针。

该函数只要查找该页上对应的信息并且将其输出[3]即可。对于页标志，要注意把处理器可识别的页标志转换为 RME 的标准页标志再输出。此外，两个输出参数都应该实现为可选项，当只需要查找其中一项时，另外一个参数传入 0（NULL）即可，此时只查询其中一种信息。

7.11.11 Implementation of __RME_Pgtbl_Walk

该函数负责查找整个页表[4]中一个虚拟地址是否被映射以及其信息。该函数只应该接受从顶层页目录发起的页表查找，如果试图从其他页目录开始页表查找，那么都应该返回错误。

Table 7-73 Implementation of __RME_Pgtbl_Walk

原型	ptr_t __RME_Pgtbl_Walk(struct RME_Cap_Pgtbl* Pgtbl_Op, ptr_t Vaddr, ptr_t* Pgtbl, ptr_t* Map_Vaddr, ptr_t* Paddr, ptr_t* Size_Order, ptr_t* Num_Order, ptr_t* Flags)
意义	查找页表[5]中一个虚拟地址是否被映射以及其信息，并且返回之。
返回值	ptr_t

[1] 找到该页

[2] 未找到或该位置上映射的表项为页目录

[3] 写入指针所指的变量内

[4] 页目录树

[5] 页目录树

	成功[1]返回 0，失败[2]返回 RME_ERR_PGT_OPFAIL（-1）。
参数	struct RME_Cap_Pgtbl* Pgtbl_Op 指向该页目录的，含有该页目录的所有信息的页目录权能。该页目录必须是顶层的。
	ptr_t Vaddr 需要查询的虚拟地址。
	ptr_t* Pgtbl 该参数用于输出，是指向该虚拟地址所在的页目录内核对象的存放虚拟地址的指针。
	ptr_t* Map_Vaddr 该参数用于输出，是指向该虚拟地址所在的页框的映射起始虚拟地址的指针。
	ptr_t* Paddr 该参数用于输出，是指向该虚拟地址所在的页框的映射起始物理地址的指针。
	ptr_t* Size_Order 该参数用于输出，是指向该虚拟地址所在的页目录的大小级数的指针。
	ptr_t* Num_Order 该参数用于输出，是指向该虚拟地址所在的页目录的数量级数的指针。
	ptr_t* Flags 该参数用于输出，是指向该虚拟地址所属页的 RME 标准页标志的指针。

该函数需要根据传入的虚拟地址查找（可能是压缩的）页表树，确定所传入的虚拟地址是否在该页表中，如果存在的话还要确定其所在的页目录内核对象本身的虚拟地址和它在这个页目录中的哪个槽位。该函数只允许查找用户页；查找内存页的信息是不允许的，必须返回错误。

对于页标志，要注意把处理器可识别的页标志转换为 RME 的标准页标志再输出。对于那些允许在页目录上设置访问控制标志的架构而言，各级页目录的标志位也应该考虑在内。此外，六个输出参数都应该实现为可选项，当只需要查找其中几项时，其他各项参数传入 0 即可，此时只查询其中几种信息。

7.12 Writing Interrupt Vectors

除了系统调用中断、时钟中断和错误处理中断之外，RME 中还有两种中断。第一种中断是透明中断，这种中断函数的编写方法和普通的无操作系统下程序的编写方法是一样的，不需要按照 RME 的中断保存方式来压栈寄存器保存上下文，而且可以任意嵌套。因此，这种中断函数的中断响应会很快，而且内容的自由度也很大。但是，该种中断不能调用任何的内核函数，最多只能读取或写入 IO，或者修改某个内

[1] 找到该页

[2] 未找到该页

存地址的变量。因此，该种中断主要适合编写那些要求快速响应的或时序严格的设备的内核态驱动程序。典型的此类设备是 1-Wire 的各种传感器[1]。

第二种中断是可感知中断。这种中断的进入和退出需要按照 RME 中断保存方式来压栈寄存器，保存线程上下文，并且不允许嵌套。该种中断可以调用一些特定的内核函数，向某个用户线程发送一些信号。此类中断适合那些需要把信号发送给应用程序并由它们来处理该设备的数据的用户态驱动程序。此外，此类中断还可以进行上下文切换。在下面的两节中，我们主要介绍第二种中断的特性，因为第一种中断和常见的裸机程序的中断区别不大。

无论是何种中断向量，它们都是这个系统非常重要的一部分。一个系统的安全性高度依赖于其中断向量的实现的安全性。对于这些中断的优先级和可嵌套性的要求是，透明中断之间可以互相嵌套，并且其优先级必须高于可感知中断；可感知中断不可互相嵌套，其优先级必须高于系统调用中断和错误处理中断。

7.12.1 Entering and Exiting of Interrupt Vectors

（可感知）中断向量的进入和退出和系统中断向量的进入和退出是一样的，都需要按照寄存器结构体的顺序进行压栈和弹栈，并且在调用以 C 语言编写的中断处理函数时需要传入寄存器组作为参数。以 C 语言编写的中断函数的原型均如下：

Table 7-74 Prototype of C Interrupt Routines

原型	void _User_Handler(struct RME_Reg_Struct* Reg)
意义	执行可感知中断处理。
返回值	无。
参数	struct RME_Reg_Struct* Reg 在进入阶段被压栈的处理器寄存器组。

7.12.2 Callable Kernel Functions in Interrupt Vectors

在（可感知）中断向量中，有一些特定的函数可以调用，来发送信号给用户态处理线程，使其就绪，或者执行其他操作。这些操作的函数列表如下：

7.12.2.1 Sending to Kernel Endpoint

该函数用来向某个内核端点发送信号。这是最重要的函数，一般用于可感知中断向量的信号外传。该函数可以在一个中断向量中调用多次，如果有多个信号端点需要发送的话。

Table 7-75 Parameters Needed for Sending to Kernel Endpoint

原型	ret_t _RME_Kern_Snd(struct RME_Reg_Struct* Reg, struct RME_Sig_Struct* Sig_Struct)

[1] 如 DS18B20，PGA300，DS2432，SHT-XX 等等

参数名称	类型	描述
Reg	...	类型为 struct RME_Reg_Struct*，是一个指向寄存器组的指针。该参数是从中断处理函数传入的。
Sig_Struct	...	类型为 struct RME_Sig_Struct*，是一个直接指向内核信号端点对象的一个指针。调用本函数会向这个内核信号端点发送信号。

该函数的返回值可能如下：

Table 7-76 Possible Return Values for Sending to Kernel Endpoint

返回值	意义
0	操作成功。
RME_ERR_SIV_FULL	该信号端点的信号计数已满，不能再向其继续发送。这是很罕见的，因为在 32 位系统中信号计数的上限为 2^{32}-1，64 位系统中则为 2^{64}-1，依此类推。

本函数和常规发送函数的不同是，如果在这一发送过程中有更高优先级的线程被唤醒，我们仅将该线程就绪而并不立即进行上下文切换。这样做的理由是，在一个中断向量中我们可能向多个内核端点发送信号，而如果被唤醒的线程的优先级一个比一个高，推迟切换到中断向量退出是一个更好的办法。关于中断向量退出时需要调用的切换函数，请参见 7.12.2.2。

7.12.2.2 Performing Context Switch upon Interrupt Exit

该函数用来在中断向量退出时进行最终的线程切换。在任何调用了_RME_Kern_Snd 函数的中断向量的结尾，该函数都必须被调用。

Table 7-77 Parameters Needed for Performing Context Switch upon Interrupt Exit

原型	void _RME_Kern_High(struct RME_Reg_Struct* Reg, ptr_t CPUID)	
参数名称	类型	描述
Reg	...	类型为 struct RME_Reg_Struct*，是一个指向寄存器组的指针。该参数是从中断处理函数传入的。
CPUID	ptr_t	当前 CPU 的 CPUID。

该函数无返回值。

7.12.2.3 Increasing Value of RME_Timestamp

该函数会增加 RME_Timestamp 的值若干个时间片，主要用来在无节拍内核中实现系统时间计时器的更新。需要注意的是，在无节拍内核中，只需要一个核去更新该值即可。

Table 7-78 Parameters Needed for Increasing Value of RME_Timestamp

原型	ptr_t _RME_Timestamp_Inc(cnt_t Value)	
参数名称	类型	描述
Value	cnt_t	要增加的值。这个值必须大于 0，否则内核会崩溃。

该函数会返回更新之前的 RME_Timestamp 值。

7.13 Explanations for Other Functions

在编写底层驱动和调试内核代码的过程中，有几个常用的助手函数可以使用。内核提供这些函数，这样就尽可能地实现了与编译器自带 C 运行时库的脱钩。这些函数的定义都位于 kernel.h，在需要使用时包含 kernel.h 即可。这些函数的列表如下：

7.13.1 Variable Clearing

该函数用来在内核中清零一片区域。该函数实质上等价于 C 语言运行时库的 memset 函数填充 0 时的特殊情况。

Table 7-79 Parameters Needed for Variable Clearing

原型	void _RME_Clear(void* Addr, ptr_t Size)	
参数名称	类型	描述
Addr	void*	需要清零区域的起始地址。
Size	ptr_t	需要清零区域的字节数。

7.13.2 Comparing Memory Segments

该函数用来比较两段内存是否相同。该函数实质上等价于 C 语言运行时库的 memcmp。

Table 7-80 Parameters Needed for Comparing Memory Segments

原型	ret_t _RME_Memcmp(const void* Ptr1, const void* Ptr2, ptr_t Num)	
参数名称	类型	描述
Ptr1	const void*	指向参与比较的第一段内存的指针。
Ptr2	const void*	指向参与比较的第二段内存的指针。
Num	ptr_t	要比较内存的长度，单位是字节。

如果两段内存在指定的长度范围内完全相同，会返回 0；如果不相同则会返回一个非 0 值。

7.13.3 Replicating Memory Segments

该函数用来复制一段内存的内容到另一区域。该函数实质上等价于 C 语言运行时库的 memcpy。两段内存区域不能重叠，否则该函数的行为是未定义的。

Table 7-81 Parameters Needed for Replicating Memory Segments

原型	void _RME_Memcpy(void* Dst, void* Src, ptr_t Num)	
参数名称	类型	描述
Dst	void*	复制的目标地址。
Src	void*	复制的源地址。
Num	ptr_t	要复制的内存的长度，单位是字节。

需要注意的是，7.13.1-7.13.3 列出的三个函数都是它们功能的逐字节实现，并且没有考虑任何优化，因此不要在大段内存操作中使用它们。这是为了最大的编译器和架构兼容性（某些架构对于按字操作有对齐等特殊要求；又或者需要使用特殊指令才能高效操作；又或者其编译器内建的高速实现会使用 FPU 寄存器。这三种情况在内核中都必须被尽力避免）。RME 的架构无关部分没有使用这三个函数中的任何一个；在硬件抽象层层中也应尽量避免用大段的内存操作。如果一定要用到大段内存操作，那么可以考虑自行编写，或者使用编译器提供的版本。无论如何，使用到的操作一定不能运用 FPU 寄存器，或者造成内存访问不对齐错误，这一点在使用编译器提供的库函数时应多加注意。

7.13.4 Printing Signed Integers

该函数用来按十（10）进制打印一个有符号整数，主要用于内核调试。打印是阻塞的，直到打印完成为止函数才返回。打印是包含符号位的。

Table 7-82 Parameters Needed for Printing Signed Integers

原型	cnt_t RME_Print_Int(cnt_t Int)	
参数名称	类型	描述
Int	cnt_t	需要打印的有符号整数。

该函数的返回值是成功打印的字符串的长度。

7.13.5 Printing Unsigned Integers

该函数用来按十六（16）进制打印一个无符号整数，主要用于内核调试。打印是阻塞的，直到打印完成为止函数才返回。打印是不包含“0x”前缀的，并且十六进制中的 A-F 均为大写。

Table 7-83 Parameters Needed for Printing Unsigned Integers

原型	cnt_t RME_Print_Uint(ptr_t Uint)	
参数名称	类型	描述
Uint	ptr_t	需要打印的无符号整数。

该函数的返回值是成功打印的字符串的长度。

7.13.6 Printing Strings

该函数用来打印一个字符串，主要用于内核调试。打印是阻塞的，直到打印完成为止函数才返回。

Table 7-84 Parameters Needed for Printing Strings

原型	cnt_t RME_Print_String(s8_t* String)	
参数名称	类型	描述
String	s8_t*	需要打印的字符串。

该函数的返回值是成功打印的字符串的长度。这个长度不包括字符串的“\0”终结标志。

7.14 Bibliography

无

Chapter 8 Kernel Function Implementation Specifications

8.1 Introduction to Kernel Function Implementation Specifications

处理器的底层功能在 RME 中是使用内核调用实现的。这些底层功能往往五花八门，而 RME 的内核调用必须支持这些底层功能。如果每种架构都采用自己的实现方案，那么必然会造成各个移植的接口语义互不兼容，极大地增加维护工作量，并且导致生态环境碎片化。为了避免这种现象，RME 规定了一系列常用内核调用的实现方法。凡是在本章所述范围内的内核调用，其实现接口必须与本章所述一致。对于那些本章未作规定而处理器或系统提供了的额外功能，其实现是自由的，但是推荐与本章保持一致的风格。在下面的各个小节中，我们将分别介绍各个内核调用类别以及它们的实现规范。

RME 的内核调用可以携带一个半字长的主功能号，一个半字长的子功能号和两个单字长的参数。但是，RME 并不禁止实现者使用处理器的寄存器组传递额外的参数。在某些扩展标准中，允许传递额外的参数，并且也允许输出额外的参数。这些允许的例外情况会在具体章节进行说明。如果不加特殊说明，所有内核调用的系统调用号 N 都是 RME_SVC_KERN，都要求传入一个有效[1]的内核调用权能作为其 C 位置上的参数，并且返回的参数类型都是 ret_t。在后面的章节中，我们把第六章已经列出的与内核调用权能相关的那些错误返回值返回值略去而不加介绍。

有些内核调用需要传入某些额外的权能[2]。此时，我们不对这些权能进行操作标志属性检查而默认可以进行操作。

8.2 Page Table Kernel Functions (“P” Standard Extension)

页表是操作系统进行内存管理的唯一参照，也是内核移植中最重要的数据结构之一。RME 被设计为支持几乎所有的内存访问控制方法，包括了内存保护单元、软件填充的内存管理单元和硬件填充的内存管理单元三类。此外，页表管理还要支持多核操作，以及对快表或寄存器缓存的刷新。因此，页表相关功能的实现是内核移植中的一个重点，也是一个难点。RME 规定的页表相关内核调用一共有如下 5 个：

Table 8-1 Overview of Page Table Related Kernel Functions

主功能号	调用名称	功能
0	RME_KERN_PGTBL_CACHE_CLR	清除当前 CPU 上某页表的全部缓存。
1	RME_KERN_PGTBL_LINE_CLR	清除当前 CPU 上某页表的缓存的某一行。
2	RME_KERN_PGTBL_ASID_SET	设置某个页表的 ASID。
3	RME_KERN_PGTBL_TLB_LOCK	锁定 TLB 内的某页。
4	RME_KERN_PGTBL_ENTRY_MOD	查询或修改某页属性。

[1] 允许操作范围包括所传入的主功能号

[2] 比如页目录权能和线程权能等等

8.2.1 Four Recommended Implementations of Page Tables

在 RME 中，随着底层硬件的不同，所采取的页表设计策略也可以不同。我们把所有的内存管理硬件分成四类，如下表所述：

Table 8-2 Overview of Four Recommended Implementations of Page Tables

分类	路径压缩	缓存更新	缓存	构造/析构限制
单核下的内存保护单元	是	变动时更新	元数据	无子页目录，甚至无页
多核下的内存保护单元	是	缺页时更新	元数据	无限制
软件填充的内存管理单元	否	缺页时更新	元数据	无限制
硬件填充的内存管理单元	否	硬件更新	快表	无限制

8.2.1.1 Single-core MPU

此类页表的实现表现为支持路径压缩[1]，缓存由储存在顶层页目录处的元数据组成。每一级页目录都包含一个指向顶层页目录的指针，并在映射、解除映射、构造和析构时都会直接将改动同步到顶层页目录内部包含的元数据，因此内存保护单元寄存器内部的数据总是与页表的实际内涵一致，无需进行任何额外的缓存维护工作。由于每一级页目录仅能记载一个指向顶层页目录的指针，此种实现不允许页表之间共享页目录。由于在构造和析构时要维护缓存元数据，此种实现在构造和析构时都要求子页目录中没有二级子页目录[2]，或者甚至也不能含有页[3]。由于不允许多个页表共享页目录，因此此种页表也没有必要支持页目录属性。

此种实现允许映射的静态页的数量不超过“内存保护单元可用区域的数量-1”[4]，对于动态页的数量则不做限制。在静态页上不会发生任何的缺页中断，在动态页上仅会在替换时发生中断。动态页的替换算法实现是由移植者自行决定的。EDI 推荐使用的算法是近期最少使用（Least Recently Used，LRU）算法。

ARMv7-M 的页表是此类实现的一个典型例子。此类实现不需要任何的页表内核调用，因为改动总是和缓存元数据状态一致。这使得该实现能够获得非常好的实时性。

8.2.1.2 Multi-core MPU

此类页表的实现表现为支持路径压缩[5]，缓存由储存在顶层页目录处的元数据组成。每一个 CPU 的缓存都是独立的，并且它们会在缺页中断发生时独立更新。这种实现允许不同 CPU 的缓存包含不同的条目，有利于在各个 CPU 之间分担工作，而且也不会造成各个 CPU 之间互相干扰。在映射、解除映射、

[1] 因此在构造时需要检查起始地址的合法性

[2] 当一个子页目录对应一个区域时，比如 ARMv7-M

[3] 当一个子页目录可能对应多个区域时，比如 ARMv8-M

[4] 因为要预留至少一个区域出来给动态页替换用

[5] 因此在构造时需要检查起始地址的合法性

构造和析构时，不会将改动同步到缓存元数据中。缓存元数据的条目添加和替换是在缺页中断发生时进行的；对于缓存元数据的清除操作，则需要使用内核调用进行。一旦进行了解除映射或者析构，所有核上都必须调用相应操作来重置与该页目录有关的缓存。如果有几个页表共享该页目录，那么在所有核上这些页表的缓存都必须被重置。

和 8.2.1.1 节所述的结构相比，此种设计虽然在实时性上有所损失，但是页表之间可以共享页目录，而且在构造和析构时也没有任何限制。对于实时性的损失也可以用如下方法进行弥补：静态页一经更新到缓存元数据中就不会消失，直到静态页的数量达到上限，而又有新的静态页需要被映射为止；而动态页则只能替换动态页而永远不会替换静态页。这样，只要一个 CPU 同时访问的静态页的数量不超过静态页的上限[1]，除去第一次访问以外，对静态页的后续访问都是实时的。页的替换算法也是由移植者决定的。

此种实现是否支持页目录属性是由移植者的实现决定的。如果用户决定实现页目录属性，那么在 __RME_Pgtbl_Walk 函数中返回的属性值（Flags）必须是各级页目录的属性值的共同子集，并且 __RME_Pgtbl_Pgdir_Map 函数中也要支持页目录属性。EDI 不推荐实现页目录属性支持，因为通常而言很少用到而无必要。

TriCore 的页表是此类实现的一个典型例子。此类实现的实时性也较好，仅次于 8.2.1.1 所述的实现。在此类页表实现中，可以实现的内核功能调用如下：

Table 8-3 Kernel Functions Needed on a Multi-core MPU architecture

名称	功能类别	实现必要性
RME_KERN_PGTBL_CACHE_CLR	清除缓存	必要
RME_KERN_PGTBL_LINE_CLR	清除缓存	可选

8.2.1.3 Software-loaded MMU

此类页表的实现和 8.2.1.2 所描述的实现是完全一样的，唯一的区别是此时路径压缩不被支持，创建页目录时起始地址一项也可以留空。关于具体的描述请参见 8.2.1.2。

TMS320C66X 的页表是此类实现的一个典型例子。此类实现的实时性和 8.2.1.2 描述的实现是完全一致的。在此实现中要求实现的内核功能调用也请参见 8.2.1.2 所述。

8.2.1.4 Hardware-loaded MMU

此类页表的实现表现为不支持路径压缩，缓存是硬件实现的快表（Trans Look-aside Buffer，TLB），而且缓存更新是由硬件进行的。在映射和构造时，无需特别考量，在解除映射和析构时则要强制刷新所有 CPU 的 TLB[2]。页表之间可以任意共享页目录。

[1] 这个上限也是“内存保护单元可用区域的数量-1”

[2] 如果这些 CPU 的 TLB 可能含有过期条目的话

在实时性方面，由于 TLB 的更新完全由硬件控制，因此需要更多的内核功能调用来辅助进行 TLB 的锁定和地址空间标识符（Address Space IDentifier，ASID）的相关操作来确保实时性。

此种实现一般要支持页目录属性，因为相关的硬件支持往往存在，而且此类架构一般都用于相当复杂的系统，具有对页目录属性的现实需求。

x86-64 的页表是此类实现的一个典型例子。此类实现的实时性的不确定程度比较大。如果实现了 ASID 支持和 TLB 锁定，那么可以获得非常好的实时性；如果没有实现任何形式的 TLB 锁定，或者处理器不支持 ASID，那么实时性就较差。在此类页表实现中，可以实现的内核功能调用如下：

Table 8-4 Kernel Functions Needed on a hardware-assisted MMU architecture

名称	功能类别	实现必要性
RME_KERN_PGTBL_CACHE_CLR	清除缓存	必要
RME_KERN_PGTBL_LINE_CLR	清除缓存	可选
RME_KERN_PGTBL_ASID_SET	设置属性	可选
RME_KERN_PGTBL_TLB_LOCK	锁定快表	可选
RME_KERN_PGTBL_ENTRY_MOD	查询或修改属性	可选

如果架构支持更多形式的页表相关操作，那么增加额外的内核调用实现它们也是允许的。

8.2.2 Flushing the CPU-local Cache of a Page Table

该操作应当被实现为清除某页表在当前 CPU 上的全部缓存。在 8.2.1.1 所述的情况下，该操作不需要实现；在 8.2.1.2 和 8.2.1.3 所述的情况下，该操作必须实现，而且应当清空当前 CPU 上的该页表的顶层的缓存元数据；在 8.2.1.4 所述的情况下，该操作也必须实现。如果处理器支持 ASID，那么应该清除 TLB 中该页表对应的 ASID 的所有条目[1]；如果该处理器不支持 ASID 而且当前使用的是本页表，那么直接刷新 TLB 即可；如果该处理器不支持 ASID 而且当前使用的不是本页表，那么无需进行任何操作直接返回成功即可。

Table 8-5 Parameters Needed for Flushing the CPU-local Cache of a Page Table

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_PGTBL_CACHE_CLR。
Cap_Pgtbl	cid_t	Sub_ID	一个对应于要清除缓存的顶层页目录权能的权能号。该权能号可以是一级或者二级查找编码。

该操作的返回值建议实现为如下：

[1] 包括锁定的 TLB 条目

Table 8-6 Recommended Ret. Impl. for Flushing the CPU-local Cache of a Page Table

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Pgtbl 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Pgtbl 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Pgtbl 不是顶层页目录权能，或为空白权能。

8.2.3 Invalidating a Line of the CPU-local Cache of a Page Table

该操作应当被实现为清除某页表在当前 CPU 上的一个特定条目。在 8.2.1.1 所述的情况下，该操作不需要实现；在 8.2.1.2 和 8.2.1.3 所述的情况下，该操作可选实现，如果实现则应当清除当前 CPU 上的该页表的顶层的缓存元数据中的某一条；在 8.2.1.4 所述的情况下，该操作也可选实现。如果处理器支持 ASID，那么应该清除 TLB 中该页表对应的 ASID 的某一条[1]；如果该处理器不支持 ASID 而且当前使用的是本页表，那么清除 TLB 中的某一条即可；如果该处理器不支持 ASID 而且当前使用的不是本页表，那么无需进行任何操作，那么无需进行任何操作直接返回成功即可。

Table 8-7 Parameters Needed for Inv. a Line of the CPU-local Cache of a Page Table

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_PGTBL_CACHE_CLR。
Cap_Pgtbl	cid_t	Sub_ID	一个对应于要清除缓存中某一行的顶层页目录权能的权能号。该权能号可以是一级或者二级查找编码。
Vaddr	ptr_t	Param1	要清理 TLB 的虚拟地址。

该操作的返回值建议实现为如下：

Table 8-8 Recommended Ret. Impl. for Inv. a Line of the CPU-local Cache of a Page Table

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Pgtbl 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Pgtbl 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Pgtbl 不是顶层页目录权能，或为空白权能。

[1] 包括锁定的 TLB 条目，如果它被选中的话

8.2.4 Setting the ASID of a Page Table

该操作应当被实现为设置某个页表的 ASID。页表的 ASID 的存储位置问题由实现者自行解决。在 8.2.1.1，8.2.1.2 和 8.2.1.3 所述的情况下，该操作不需要实现；在 8.2.1.4 所述的情况下，如果处理器支持 ASID，那么该操作可选实现，并应该设定该页表的 ASID；如果该处理器不支持 ASID，那么该操作不需要实现。

Table 8-9 Parameters Needed for Setting the ASID of a Page Table

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_PGTBL_ASID_SET。
Cap_Pgtbl	cid_t	Sub_ID	一个对应于要设置 ASID 的顶层页目录权能的权能号。该权能号可以是一级或者二级查找编码。
ASID	ptr_t	Param1	要设置为该页表 ASID 的值。

该操作的返回值建议实现为如下：

Table 8-10 Recommended Ret. Impl. for Setting the ASID of a Page Table

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Pgtbl 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Pgtbl 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Pgtbl 不是顶层页目录权能，或为空白权能。

8.2.5 Locking Down a Page into the TLB

该操作应当被实现为锁定某一页到 TLB 中。在 8.2.1.1，8.2.1.2 和 8.2.1.3 所述的情况下，该操作不需要实现；在 8.2.1.4 所述的情况下，该操作可选实现，并应该将指定的虚拟地址锁定到 TLB 中。如果处理器不支持 ASID，那么此锁定页面在切换地址空间时是否被清除是由架构和实现者决定的。

Table 8-11 Parameters Needed for Locking Down a Page into the TLB

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_PGTBL_TLB_LOCK。
Cap_Pgtbl	cid_t	Sub_ID	一个对应于要设置 ASID 的顶层页目录权能的权能号。该权能号可以是一级或者二级查找编码。
Vaddr	ptr_t	Param1	要锁定到 TLB 的虚拟地址。

该操作的返回值建议实现为如下：

Table 8-12 Recommended Ret. Impl. for Locking Down a Page into the TLB

返回值	意义
0	操作成功。
RME_ERR_CAP_RANGE	Cap_Pgtbl 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Pgtbl 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Pgtbl 不是顶层页目录权能，或为空白权能。
RME_ERR_PGT_ADDR	指定的地址在所给页表中查询不到，无法进行 TLB 锁定。
RME_ERR_PGT_MAP	TLB 容量已满，无法继续锁定新页面。

8.2.6 Get ting or Setting the Page Attributes of a Page

该操作应该被实现为对某页面属性的查找或修改。最常见的页面属性是“脏”（Dirty）[1]属性，这一属性用来判断该页是否被写过；某些处理器可能具备其他可查询或者修改的页面属性。需要注意的是，本函数不能被用来修改 RME 在页表操作接口中规定的任何标准页属性，而只能用来读取它们；对其他架构特有属性的读写则是都被允许的。在 8.2.1.1，8.2.1.2 和 8.2.1.3 所述的情况下，该操作可选实现，如果实现则不需要具备修改属性的功能；在 8.2.1.4 所述的情况下，那么该操作可选实现，如果实现则应该实现读写功能。

Table 8-13 Parameters Needed for Getting or Setting the Page Attributes of a Page

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_PGTBL_ENTRY_MOD。
Cap_Pgtbl	cid_t	Sub_ID	一个对应于要查找页面属性的顶层页目录权能的权能号。该权能号可以是一级或者二级查找编码。
Vaddr	ptr_t	Param1	要查询或修改属性的虚拟地址。
Operation	ptr_t	Param2	要查询或修改的属性。此参数的意义由实现者决定。

该操作的返回值建议实现为如下：

Table 8-14 Recommended Ret. Impl. for Getting or Setting the Page Attributes of a Page

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。

[1] 又称“被修改”（Modified）

返回值	意义
RME_ERR_CAP_RANGE	Cap_Pgtbl 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Pgtbl 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Pgtbl 不是顶层页目录权能，或为空白权能。
RME_ERR_PGT_ADDR	指定的地址在所给页表中查询不到，无法进行属性修改或查询。

8.3 Interrupt Controller Kernel Functions (“I” Standard Extension)

中断是系统响应外部事件的重要方法，它负责接收外部异步信号并且将该信号传递给系统的其他部分，是实时系统的重要组成部分。RME 采用内核信号端点的办法来管理各个中断，每个中断都会被表征成向某个内核信号端点的发送。因此，我们需要知道哪个中断应当被发送到哪个内核端点；如果系统中有多个 CPU，我们还要知道硬件的中断要发送给哪个 CPU。RME 规定的中断控制器相关内核调用一共有如下 3 个，它们都是可选实现的：

Table 8-15 Overview of Interrupt Controller Kernel Functions

主功能号	调用名称	功能
10	RME_KERN_INT_LOCAL_MOD	获取或设置某本地中断源的状态。
11	RME_KERN_INT_GLOBAL_MOD	获取或设置某全局中断源的状态。
12	RME_KERN_INT_LOCAL_TRIG	触发某个 CPU 的某个本地中断源。

8.3.1 Getting or Setting the State of a Local Interrupt Source

该操作应该被设计为获取或者设置当前 CPU 的某个本地中断源状态。常见的本地中断源状态包括本地中断源的开关，抢占和非抢占优先级以及其对应的内核信号端点。该内核调用一般会修改 CPU 的本地中断控制器[1]的设置。

Table 8-16 Parameters Needed for Getting or Setting the State of a Local Int. Source

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_INT_LOCAL_MOD。
Int_Num	ptr_t	Sub_ID	要查询或修改的本地中断源的编号。
Operation	ptr_t	Param1	要执行的操作。此参数的意义由实现者决定。
Param	ptr_t	Param2	额外参数。此参数的意义由实现者决定。

该操作的返回值建议实现为如下：

[1] 如 x86-64 的 LAPIC

Table 8-17 Recommended Ret. Impl. for Getting or Setting the State of a Local Int. Source

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.3.2 Getting or Setting the State of a Global Interrupt Source

该操作应该被设计为获取或者设置全局中断源的状态。常见的全局中断源状态包括全局中断源的开关，要连接到的 CPU，以及要对应的该 CPU 本地中断源。该内核调用一般会修改系统的全局中断控制器[1]的设置。

Table 8-18 Parameters Needed for Getting or Setting the State of a Global Int. Source

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_INT_LOCAL_MOD。
Int_Num	ptr_t	Sub_ID	要查询或修改的全局中断源的编号。
Operation	ptr_t	Param1	要执行的操作。此参数的意义由实现者决定。
Param	ptr_t	Param2	额外参数。此参数的意义由实现者决定。

该操作的返回值建议实现为如下：

Table 8-19 Recommended Ret. Impl. for Getting or Setting the State of a Global Int. Source

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.3.3 Triggering the Local Interrupt Source of a CPU

该操作应该被设计为触发某个 CPU 的某个本地中断源状态。常见的用途包括触发本地软中断或是向其他处理器发送处理器间中断（Inter-Processor Interrupt，IPI）。该内核调用一般会通过 CPU 的本地中断控制器[2]触发该中断，有时也有可能通过全局中断控制器[3]触发该中断。该操作可以实现的功能包括触发一个 CPU 的一个中断，触发一个 CPU 的几个中断，触发一组 CPU 的一个中断和触发一组 CPU 的几个中断；具体要实现何种功能组合由使用者决定。

---

[1] 如 x86-64 的 IOAPIC

[2] 如 x86-64 的 LAPIC

[3] 如 x86-64 的 IOAPIC

Table 8-20 Parameters Needed for Triggering the Local Interrupt Source of a CPU

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_INT_LOCAL_TRIG。
CPUID	ptr_t	Sub_ID	要触发的 CPU 的编号。
Int_Num	ptr_t	Param1	要触发的中断号。此参数的意义由实现者决定。
Param	ptr_t	Param2	额外参数。此参数的意义由实现者决定。

该操作的返回值建议实现为如下：

Table 8-21 Recommended Ret. Impl. for Triggering the Local Interrupt Source of a CPU

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.4 Cache and Prefetching Kernel Functions (“C” Standard Extension)

缓存和预取子系统是系统性能的重要保障。缓存可以利用程序的局部性加速执行，而预取则能通过提前发起内存操作来掩盖内存延迟。基于如下的理由，我们有必要实现缓存和预取相关的内核调用：

1. DMA 等硬件往往和 CPU 没有缓存一致性，要手动进行缓存一致性操作。
2. 自修改代码会使指令预取和指令缓存发生失效，在某些架构上要手动清空预取和缓存。
3. 某些架构的地址空间可缓存性不由页表控制，而是由特殊的寄存器控制，需要额外配置。
4. 某些架构的缓存大小可变，并且允许将一部分缓存用作便笺存储器（Scratchpad Memory）[1]，需要额外配置。
5. 某些架构的缓存的组织可变[2]，需要额外配置。
6. 某些多核架构允许某些内存区域上各处理器的缓存不一致，需要额外配置。

RME 规定的缓存与预取相关内核调用一共如下 7 个，它们都是可选实现的：

Table 8-22 Overview of Cache and Prefetching Kernel Functions

主功能号	调用名称	功能
20	RME_KERN_CACHE_ENABLE	使能缓存。
21	RME_KERN_CACHE_DISABLE	除能缓存。
22	RME_KERN_CACHE_CONFIG	配置缓存。

[1] 又称紧耦合存储器（Tightly Coupled Memory，TCM）

[2] 路、组或者缓存行大小

主功能号	调用名称	功能
23	RME_KERN_CACHE_INVALIDATE	缓存强制失效。
24	RME_KERN_CACHE_LOCK	锁定缓存。
25	RME_KERN_PRFTH_ENABLE	使能预取。
26	RME_KERN_PRFTH_DISABLE	除能预取。

8.4.1 Enabling Cache

该调用使能 CPU 的某个缓存。通常而言，这仅仅需要操作缓存控制器的一个寄存器。被操作的缓存可能是一级缓存（L1），二级缓存（L2），以此类推。缓存又可以进一步细分成指令缓存和数据缓存。在某些可缓存性不受页表控制的处理器上，地址空间的可缓存性也可以由此内核调用配置。

Table 8-23 Parameters Needed for Enabling Cache

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_CACHE_ENABLE。
Cache_ID	ptr_t	Sub_ID	要使能的缓存的代号。此参数的意义由实现者决定。
Param1	ptr_t	Param1	额外参数。此参数的意义由实现者决定。
Param2	ptr_t	Param2	额外参数。此参数的意义由实现者决定。

该操作的返回值建议实现为如下：

Table 8-24 Recommended Ret. Impl. for Enabling Cache

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.4.2 Disabling Cache

该调用除能 CPU 的某个缓存。在除能缓存之前，是否把被修改缓存行的内容全部刷新到内存是由实现者决定的；EDI 推荐把所有修改的缓存行刷新到内存再关闭缓存。在某些可缓存性不受页表控制的处理器上，地址空间的可缓存性也可以由此内核调用配置。

Table 8-25 Parameters Needed for Disabling Cache

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_CACHE_DISABLE。
Cache_ID	ptr_t	Sub_ID	要除能的缓存的代号。此参数的意义由实现者决定。
Param1	ptr_t	Param1	额外参数。此参数的意义由实现者决定。

参数名称	类型	位置	描述
Param2	ptr_t	Param2	额外参数。此参数的意义由实现者决定。

该操作的返回值建议实现为如下：

Table 8-26 Recommended Ret. Impl. for Disabling Cache

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.4.3 Configuring Cache

某些处理器支持缓存组织的动态配置。这可能包括允许将一部分缓存用作 TCM，或者允许修改缓存的组织结构，或者允许将通用缓存配置成专门的指令缓存或数据缓存，或者允许各个 CPU 之间的缓存不一致，或者将某些缓存行预留给某个 CPU。这些操作是由本内核调用负责的。

Table 8-27 Parameters Needed for Configuring Cache

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_CACHE_CONFIG。
Cache_ID	ptr_t	Sub_ID	要配置的缓存的代号。此参数的意义由实现者决定。
Param1	ptr_t	Param1	额外参数。此参数的意义由实现者决定。
Param2	ptr_t	Param2	额外参数。此参数的意义由实现者决定。

该操作的返回值建议实现为如下：

Table 8-28 Recommended Ret. Impl. for Configuring Cache

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.4.4 Invalidating Cache

在很多场合，我们需要使得一条特定的缓存行或者缓存的一部分失效。这是为了将缓存中的数据及时刷新到内存中，共某些外设使用。典型的此类外设包括 DMA，网卡或者显卡等等。此类操作是由本内核调用负责的。

Table 8-29 Parameters Needed for Invalidating Cache

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_CACHE_INVALIDATE。
Cache_ID	ptr_t	Sub_ID	要强制失效的缓存的代号。此参数的意义由实现者决定。
Param1	ptr_t	Param1	额外参数。此参数的意义由实现者决定。
Param2	ptr_t	Param2	额外参数。此参数的意义由实现者决定。

该操作的返回值建议实现为如下：

Table 8-30 Recommended Ret. Impl. for Invalidating Cache

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.4.5 Locking Down Cache

在某些场合，我们希望某些数据永远存留在缓存之内，永远不会被替换，此时就需要锁定缓存。锁定缓存有很多种可能的实现方案：锁定缓存的某一行，锁定缓存的某一组，锁定缓存的某一路，或者锁定缓存的某一区域等等。此类操作是由本内核调用负责的。

Table 8-31 Parameters Needed for Locking Down Cache

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_CACHE_LOCK。
Cache_ID	ptr_t	Sub_ID	要锁定的缓存的代号。此参数的意义由实现者决定。
Param1	ptr_t	Param1	额外参数。此参数的意义由实现者决定。
Param2	ptr_t	Param2	额外参数。此参数的意义由实现者决定。

该操作的返回值建议实现为如下：

Table 8-32 Recommended Ret. Impl. for Locking Down Cache

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.4.6 Enabling Prefetcher

该操作使能 CPU 的预取功能。通常而言，这只需要配置预取引擎控制器的一个寄存器。在某些处理器上，地址空间的可预取性也可以由此内核调用配置。

Table 8-33 Parameters Needed for Enabling Prefetcher

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_PRFTH_ENABLE。
Prefetch_ID	ptr_t	Sub_ID	要使能的预取引擎的代号。此参数的意义由实现者决定。
Param1	ptr_t	Param1	额外参数。此参数的意义由实现者决定。
Param2	ptr_t	Param2	额外参数。此参数的意义由实现者决定。

该操作的返回值建议实现为如下：

Table 8-34 Recommended Ret. Impl. for Enabling Prefetcher

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.4.7 Disabling Prefetcher

该操作除能 CPU 的预取功能。通常而言，这只需要配置预取引擎控制器的一个寄存器。在某些处理器上，地址空间的可预取性也可以由此内核调用配置。

Table 8-35 Parameters Needed for Disabling Prefetcher

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_PRFTH_DISABLE。
Prefetch_ID	ptr_t	Sub_ID	要使能的预取引擎的代号。此参数的意义由实现者决定。
Param1	ptr_t	Param1	额外参数。此参数的意义由实现者决定。
Param2	ptr_t	Param2	额外参数。此参数的意义由实现者决定。

该操作的返回值建议实现为如下：

Table 8-36 Recommended Ret. Impl. for Disabling Prefetcher

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.5 Hardware Hotplug Kernel Functions ("H" Standard Extension)

硬件热插拔是系统可用性和可扩展性很重要的一部分。热插拔允许系统在连续运转的前提下动态地增加或减少计算资源[1]，这使服务器级系统和大型机的规模灵活性大大提高；在较为复杂的嵌入式应用中，也可以动态增加板卡来增前运算能力，或者动态减少板卡来进行维修和系统维护。RME 规定的硬件热插拔相关内核调用一共有如下 3 个，它们都是可选实现的：

Table 8-37 Overview of Hardware Hotplug Kernel Functions

主功能号	调用名称	功能
30	RME_KERN_HPNP_PCPU_MOD	获取或设置某物理处理器封装的状态。
31	RME_KERN_HPNP_LCPU_MOD	获取或设置某逻辑处理器的状态。
32	RME_KERN_HPNP_PMEM_MOD	获取或设置某物理内存封装的状态。

8.5.1 Getting or Setting the State of a Physical CPU Package

该操作获取或设置某个物理处理器封装的状态。"物理处理器"指一个处理器芯片封装，或者一块含有多个处理器芯片封装的处理器板卡。常见的需要获取或设置的状态包括电源通断、插入检测等等。

Table 8-38 Parameters Needed for Getting or Setting the State of a Phys. CPU Pkg.

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_HPNP_PCPU_MOD。
PCPUID	ptr_t	Sub_ID	要获取或设置状态的物理处理器的代号。
Param1	ptr_t	Param1	额外参数。此参数的意义由实现者决定。
Param2	ptr_t	Param2	额外参数。此参数的意义由实现者决定。

该操作的返回值建议实现为如下：

Table 8-39 Recommended Ret. Impl. for Getting or Setting the State of a Phys. CPU Pkg.

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.5.2 Getting or Setting the State of a Logical CPU

[1] 主要指 CPU 和内存

该操作获取或设置某个逻辑处理器的状态。“逻辑处理器”指软件能识别的相互独立的 CPU 核，每个核在任何时间仅可运行一个线程。需要注意的是，一个逻辑处理器未必总是对应于一个物理核心或者封装。多核处理器可能会在一个封装之内整合多个物理处理器，而超线程技术则会将一个物理处理器模拟成多个逻辑处理器使用。常见的需要获取或设置的状态包括处理器初始化、关闭等等。

Table 8-40 Parameters Needed for Getting or Setting the State of a Logical CPU

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_HPNP_LCPU_MOD。
LCPUID	ptr_t	Sub_ID	要获取或设置状态的逻辑处理器的代号。
Param1	ptr_t	Param1	额外参数。此参数的意义由实现者决定。
Param2	ptr_t	Param2	额外参数。此参数的意义由实现者决定。

该操作的返回值建议实现为如下：

Table 8-41 Recommended Ret. Impl. for Getting or Setting the State of a Logical CPU

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.5.3 Getting or Setting the State of a Physical Memory Package

该操作获取或设置某个物理内存封装的状态。“物理内存封装”指一个物理内存颗粒，或者一个包含多个物理内存颗粒的内存模组。常见的需要获取或设置的状态包括电源通断、插入检测等等。

Table 8-42 Parameters Needed for Getting or Setting the State of a Phys. Mem. Pkg.

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_HPNP_PMEM_MOD。
PMEMID	ptr_t	Sub_ID	要获取或设置状态的物理内存封装的代号。
Param1	ptr_t	Param1	额外参数。此参数的意义由实现者决定。
Param2	ptr_t	Param2	额外参数。此参数的意义由实现者决定。

该操作的返回值建议实现为如下：

Table 8-43 Recommended Ret. Impl. for Getting or Setting the State of a Phys. Mem. Pkg.

返回值	意义

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.6 Clock and Power Kernel Functions (“F” Standard Extension)

系统的时钟频率和电压直接决定了系统的性能和功耗，并且对电池寿命和散热余量等方面都有重要影响。RME 规定的时钟与电压相关内核调用一共有如下 4 个，它们都是可选实现的：

Table 8-44 Overview of Clock and Power Kernel Functions

主功能号	调用名称	功能
40	RME_KERN_VOLTAGE_MOD	获取或设置电压参数。
41	RME_KERN_FREQ_MOD	获取或设置频率参数。
42	RME_KERN_POWER_MOD	获取或设置电源状态。
43	RME_KERN_SAFETY_MOD	获取或设置安全保护状态。

8.6.1 Getting or Setting Voltage Parameters

本操作获取或者设置 CPU、内存和其他外设的各路供电电压值。电压的单位可以是毫伏（mV）也可以是微伏（uV）。

Table 8-45 Parameters Needed for Getting or Setting Voltage Parameters

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_VOLTAGE_MOD。
PDOMID	ptr_t	Sub_ID	要获取或设置电压的电源域的代号。
Operation	ptr_t	Param1	操作选项，决定了要进行的操作。
Voltage	ptr_t	Param2	要设置的电压值。此参数的单位由实现者决定。

该操作的返回值建议实现为如下：

Table 8-46 Recommended Ret. Impl. for Getting or Setting Voltage Parameters

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.6.2 Getting or Setting Frequency Parameters

本操作获取或者设置 CPU、内存和其他外设的各个时钟域的工作频率。频率的单位可以是毫赫兹（mHz），赫兹（Hz），千赫兹（kHz），兆赫兹（MHz）或者吉赫兹（GHz）。

Table 8-47 Parameters Needed for Getting or Setting Frequency Parameters

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_FREQ_MOD。
FDOMID	ptr_t	Sub_ID	要获取或设置时钟频率的时钟域的代号。
Operation	ptr_t	Param1	操作选项，决定了要进行的操作。
Frequency	ptr_t	Param2	要设置的频率值。此参数的单位由实现者决定。

该操作的返回值建议实现为如下：

Table 8-48 Recommended Ret. Impl. for Getting or Setting Frequency Parameters

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.6.3 Getting or Setting Power States

本操作获取或者设置 CPU、内存和其他外设的电源状态。电源状态通常包括电源域的开关和时钟域的开关。在支持动态电压频率调整（Dynamic Voltage Frequency Scaling，DVFS）的处理器上，本操作也负责设置该功能的状态。

Table 8-49 Parameters Needed for Getting or Setting Power States

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_POWER_MOD。
PDOMID	ptr_t	Sub_ID	要获取或设置电源状态的电源域的代号。
Operation	ptr_t	Param1	操作选项，决定了要进行的操作。

该操作的返回值建议实现为如下：

Table 8-50 Recommended Ret. Impl. for Getting or Setting Power States

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.6.4 Getting or Setting Safety Protection States

本操作获取或者设置电源和频率安全锁的状态。当安全锁未被打开时，某些电压和频率是不允许被 8.6.1-8.6.3 所述的内核调用设置的，因为这有可能会造成系统崩溃，甚至会对硬件造成永久损坏[1]。当安全锁打开时，这些限制会被解除，允许设置系统到更极端的状态。

Table 8-51 Parameters Needed for Getting or Setting Safety Protection States

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_SAFETY_MOD。
DOMID	ptr_t	Sub_ID	要获取或设置安全保护状态的电源域或时钟域的代号。
Operation	ptr_t	Param1	操作选项，决定了要进行的操作。

该操作的返回值建议实现为如下：

Table 8-52 Recommended Ret. Impl. for Getting or Setting Safety Protection States

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.7 System Monitoring Kernel Functions (“M” Standard Extension)

系统监视器是识别系统功能、确认系统状态和调校系统性能必不可少的。RME 规定的系统监视内核调用一共有如下 6 个，它们都是可选实现的：

Table 8-53 Overview of System Monitoring Kernel Functions

主功能号	调用名称	功能
50	RME_KERN_PERF_CPU_FUNC	获取 CPU 功能特性。
51	RME_KERN_PERF_MON_MOD	获取或设置性能监视器配置。
52	RME_KERN_PERF_CNT_MOD	获取或设置计次性能计数器的数值。
53	RME_KERN_PERF_CYCLE_MOD	获取或设置周期性能计数器的数值。
54	RME_KERN_PERF_DATA_MOD	获取或设置数据性能计数器的数值。
55	RME_KERN_PERF_PHYS_MOD	获取或设置物理性能计数器的数值。
56	RME_KERN_PERF_CUMUL_MOD	获取或设置累积性能计数器的数值。

[1] 尤其是电压调整

在实现某些性能计数器的读操作时，由于这些计数器的长度可能超出 RME 系统调用返回值的允许长度[1]，因此需要实现者自行使用其他寄存器来返回计数器的其余部分。

8.7.1 Getting CPU Functionality Support Statuses

同一个架构的 CPU 可能因为种种原因而支持不同的功能子集。本调用可以允许用户查询 CPU 的功能以确认其对软件的兼容性。

Table 8-54 Parameters Needed for Getting CPU Functionality Support Statuses

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_PERF_CPU_FUNC。
FUNCID	ptr_t	Sub_ID	要获取的功能寄存器的代号。

该操作的返回值建议实现为如下：

Table 8-55 Recommended Ret. Impl. for Getting CPU Functionality Support Statuses

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.7.2 Getting or Setting Performance Monitor Configurations

通常而言，每个 CPU 都有一些性能计数器来记录系统中的各种事件。系统中的事件数量往往是非常多的，而 CPU 的性能计数器数量有限，因此只能选择一部分事件记录下来。本调用允许配置性能计数器的状态并且指定各个计数器的事件来源。

Table 8-56 Parameters Needed for Getting or Setting Perf. Monitor Configurations

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_PERF_MON_MOD。
PERFID	ptr_t	Sub_ID	要获取或设置配置的性能计数器的代号。
Event_ID	ptr_t	Param1	要设置给该计数器的事件代号。
Operation	ptr_t	Param2	操作选项，决定了要进行的操作。

该操作的返回值建议实现为如下：

[1] 一个机器字长

Table 8-57 Recommended Ret. Impl. for Getting or Setting Perf. Monitor Configurations

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.7.3 Getting or Setting Counting Monitor Values

本操作获取或者设置计次性能计数器的数值。计次性能计数器负责记录系统中某些事件发生的次数。常见的计次事件有：

1. 缓存替换的次数
2. 缓存命中或不命中的次数
3. 上级缓存不命中并在下级缓存中命中或未命中的次数
4. 上级缓存不命中并在下级缓存中命中时，该命中涉及或不涉及缓存一致性操作的次数
5. NUMA 结构中，本地或异地内存访问次数
6. 处理器收到的总中断[1]的次数
7. 处理器正确预测或不正确预测的分支指令数量
8. 处理器成功执行或不成功执行的原子操作数量，以及被串行化的原子操作数量
9. 处理器获取或者执行完毕的总指令[2]的数量

Table 8-58 Parameters Needed for Getting or Setting Counting Monitor Values

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_PERF_CNT_MOD。
CNTID	ptr_t	Sub_ID	要获取或设置数值的计次性能计数器的代号。
Operation	ptr_t	Param1	操作选项，决定了要进行的操作。
Value	ptr_t	Param2	要设置给计数器的值。某些计数器是只读的，此时不允许设置操作而只允许读取操作。

该操作的返回值建议实现为如下：

Table 8-59 Recommended Ret. Impl. for Getting or Setting Counting Monitor Values

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。

[1] 或某一类中断

[2] 或某一类，比如分支，访存，I/O，乘除法等

返回值	意义
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.7.4 Getting or Setting Cycle Monitor Values

本操作获取或者设置周期性能计数器的数值。周期性能计数器负责记录系统中某些事件产生的影响持续的时钟周期数。常见的周期计量事件有：

1. 处理器上电以来经过的周期数
2. 由于缓存不命中导致流水线停滞的周期数
3. 由于快表不命中导致流水线停滞的周期数
4. 由于分支预测失败，指令延时槽或其他原因导致的处理器不分发新指令的周期数

Table 8-60 Parameters Needed for Getting or Setting Cycle Monitor Values

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_PERF_CYCLE_MOD。
CYCLEID	ptr_t	Sub_ID	要获取或设置配置的周期性能计数器的代号。
Operation	ptr_t	Param1	操作选项，决定了要进行的操作。
Value	ptr_t	Param2	要设置给计数器的值。某些计数器是只读的，此时不允许设置操作而只允许读取操作。

该操作的返回值建议实现为如下：

Table 8-61 Recommended Ret. Impl. for Getting or Setting Cycle Monitor Values

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.7.5 Getting or Setting Data Monitor Values

本操作获取或者设置数据性能计数器的数值。数据性能计数器负责记录系统中某些事件带来的累计数据交换量。常见的数据计量事件有：

1. 系统中某条总线上流过的总数据量[1]
2. 系统中由某些操作或者某些指令造成的总数据量

Table 8-62 Parameters Needed for Getting or Setting Data Monitor Values

[1] 或符合某个条件，如大小，传输方向，数据性质等的数据量

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_PERF_DATA_MOD。
DATAID	ptr_t	Sub_ID	要获取或设置配置的数据性能计数器的代号。
Operation	ptr_t	Param1	操作选项，决定了要进行的操作。
Value	ptr_t	Param2	要设置给计数器的值。某些计数器是只读的，此时不允许设置操作而只允许读取操作。

该操作的返回值建议实现为如下：

Table 8-63 Recommended Ret. Impl. for Getting or Setting Data Monitor Values

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.7.6 Getting or Setting Physical Monitor Values

本操作获取或者设置物理性能计数器的数值。物理性能计数器负责记录系统中某些物理量的值，或者其相关信息。常见的物理量有：

1. 电路某部分的温度或环境温度
2. 散热系统的风扇转速
3. 某个时钟域的电流大小
4. 某个时钟域或者电源域消耗的总电量

Table 8-64 Parameters Needed for Getting or Setting Physical Monitor Values

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_PERF_PHYS_MOD。
PHYSID	ptr_t	Sub_ID	要获取或设置配置的物理性能计数器的代号。
Operation	ptr_t	Param1	操作选项，决定了要进行的操作。
Value	ptr_t	Param2	要设置给计数器的值。某些计数器是只读的，此时不允许设置操作而只允许读取操作。

该操作的返回值建议实现为如下：

Table 8-65 Recommended Ret. Impl. for Getting or Setting Physical Monitor Values

返回值	意义

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.7.7 Getting or Setting Cumulative Monitor Values

本操作获取或者设置累积性能计数器的数值。累计性能计数器负责记录系统中某些累积性时间量的值，或者其相关信息。常见的累积量有：

1. 系统总上电时间或总上电次数
2. 系统本次上电的连续工作时间
3. 系统剩余的使用年限或使用寿命
4. 系统的健康状态和损伤管理状态

Table 8-66 Parameters Needed for Getting or Setting Cumulative Monitor Values

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_PERF_CUMUL_MOD。
CUMULID	ptr_t	Sub_ID	要获取或设置配置的累积性能计数器的代号。
Operation	ptr_t	Param1	操作选项，决定了要进行的操作。
Value	ptr_t	Param2	要设置给计数器的值。某些计数器是只读的，此时不允许设置操作而只允许读取操作。

该操作的返回值建议实现为如下：

Table 8-67 Recommended Ret. Impl. for Getting or Setting Cumulative Monitor Values

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.8 Full Virtualization Kernel Functions (“V” Standard Extension)

本标准尚未冻结。

8.9 Secure Monitor Kernel Functions (“S” Standard Extension)

本标准尚未冻结。

8.10 Debug and Trace Kernel Functions (“D” Standard Extension)

调试与跟踪是程序除错和分析的基本手段。RME 规定的调试与跟踪内核调用一共有如下 5 个，它们都是可选实现的：

Table 8-68 Overview of Debug and Trace Kernel Functions

主功能号	调用名称	功能
80	RME_KERN_DEBUG_PRINT	调试打印。
81	RME_KERN_DEBUG_REG_MOD	获取或设置某线程的寄存器组。
82	RME_KERN_DEBUG_INV_MOD	获取或设置某线程的迁移调用寄存器组。
83	RME_KERN_DEBUG_MODE_MOD	获取或设置调试引擎模式。
84	RME_KERN_DEBUG_IBP_MOD	获取或设置指令调试断点状态。
85	RME_KERN_DEBUG_DBP_MOD	获取或设置数据调试断点状态。

8.10.1 Debug Printing

调试打印是最基本的程序分析工具。为此，RME 提供了一个专门的内核调用来实现控制台字符打印。通常而言，这个调用会打印一个字符到串口，并且在打印完成之前不会返回。此调用也可以被实现为打印多个字符；在这种状况下，多余的字符参数需要实现者自行使用空闲的寄存器传入。

Table 8-69 Parameters Needed for Debug Printing

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_DEBUG_PRINT。
Char	char	Sub_ID	要打印的字符。

该操作的返回值建议实现为如下：

Table 8-70 Recommended Ret. Impl. for Debug Printing

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.10.2 Getting or Setting the Register Set of a Thread

读取和修改某个线程的寄存器组对于分析程序的执行状况是至关重要的，因此 RME 提供了一个内核调用来负责此类操作。本调用也可以用来改变程序的执行流，或者修改其协处理器状态。本内核调用负责获取或者设置线程的当前寄存器组。需要注意的是，RME 的内核调用只默认实现了两个参数和一个

返回值，而线程的寄存器组往往非常庞大。为了提高读写效率，用户应当使用其他寄存器自行实现更多的参数和返回值传递。

Table 8-71 Parameters Needed for Getting or Setting the Reg. Set of a Thread

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_DEBUG_REG_MOD。
Cap_Thd	cid_t	Sub_ID	一个对应于要获取或设置寄存器组内容的线程权能的权能号。该权能号可以是一级或者二级查找编码。
Operation	ptr_t	Param1	操作选项，决定了要进行的操作。
Reg	ptr_t	Param2	要设置给该寄存器的值。

该操作的返回值建议实现为如下：

Table 8-72 Recommended Ret. Impl. for Getting or Setting the Reg. Set of a Thread

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
RME_ERR_CAP_RANGE	Cap_Thd 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Thd 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Thd 不是线程权能权能，或为空白权能。
RME_ERR_PTH_INVSTATE	指定的寄存器不存在或者不允许读写，或者试图写入的值非法。

8.10.3 Getting or Setting The Invocation Register Set of a Thread

读取和修改迁移调用中保存的寄存器组可以使线程在从迁移调用返回时不返回到原来迁移调用发起的地方，而是返回到其他地址继续执行。这在实现某些功能如准虚拟化时是很有用处的，因此 RME 为此功能专门保留了一个内核调用。虽然迁移调用保存的寄存器组仅仅是线程寄存器组的一个小子集，但为了提高读写效率，用户应当使用其他寄存器自行实现更多的参数和返回值传递。

Table 8-73 Parameters Needed for Getting or Setting The Inv. Reg. Set of a Thread

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_DEBUG_INV_MOD。
Cap_Thd	cid_t	Sub_ID	一个对应于要获取或设置寄存器组内容的线程权能的权能号。该权能号可以是一级或者二级查找编码。
Operation	ptr_t	Param1	操作选项，决定了要进行的操作。
Reg	ptr_t	Param2	要设置给该寄存器的值。

该操作的返回值建议实现为如下：

Table 8-74 Recommended Ret. Impl. for Getting or Setting The Inv. Reg. Set of a Thread

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
RME_ERR_CAP_RANGE	Cap_Thd 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Thd 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Thd 不是线程权能权能，或为空白权能。
RME_ERR_SIV_EMPTY	指定的线程不在迁移调用中，无法读写其迁移调用寄存器组。
RME_ERR_PTH_INVSTATE	指定的寄存器不存在或者不允许读写，或者试图写入的值非法。

8.10.4 Getting or Setting Debug Engine Mode

处理器的调试引擎模组往往包括一些寄存器，通过它们可以设置处理器的调试模式以及状态。本内核调用负责获取或设置这些寄存器。

Table 8-75 Parameters Needed for Getting or Setting Debug Engine Mode

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_DEBUG_MODE_MOD。
Operation	ptr_t	Sub_ID	操作选项，决定了要进行的操作。
Param1	ptr_t	Param1	额外参数。此参数的意义由实现者决定。
Param2	ptr_t	Param2	额外参数。此参数的意义由实现者决定。

该操作的返回值建议实现为如下：

Table 8-76 Recommended Ret. Impl. for Getting or Setting Debug Engine Mode

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.10.5 Getting or Setting Instruction Breakpoint Statuses

某些处理器可能支持硬件指令断点。当处理器执行到硬件指令断点对应的指令地址时，会产生一个调试中断，调试器可以通过截获这个中断来得知程序已经运行至此。本内核调用负责驱动相关硬件来获

取和设置硬件指令断点及其状态。某些处理器的断点功能可能支持 ASID，此时需要额外传入顶级页目录权能。

Table 8-77 Parameters Needed for Getting or Setting Instruction Breakpoint Statuses

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_DEBUG_IBP_MOD。
Cap_Pgtbl	cid_t	Sub_ID	该指令断点的 ASID。
Operation	ptr_t	Param1	操作选项，决定了要进行的操作。
Address	ptr_t	Param2	要打指令断点的地址。

该操作的返回值建议实现为如下：

Table 8-78 Recommended Ret. Impl. for Getting or Setting Instruction Breakpoint Statuses

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
RME_ERR_CAP_RANGE	Cap_Pgtbl 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Pgtbl 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Pgtbl 不是顶层页目录权能，或为空白权能。
其他负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.10.6 Getting or Setting Data Breakpoint Statuses

某些处理器可能支持硬件数据断点。当处理器对某个数据地址写入或读取时，会产生一个调试中断，调试器可以通过截获这个中断来得知程序正在访问该内存中的某个变量。本内核调用负责驱动相关硬件来获取和设置硬件数据断点及其状态。某些处理器的断点功能可能支持 ASID，此时需要额外传入顶级页目录权能。某些处理器的断点功能可能支持 ASID，此时需要额外传入顶级页目录权能。

Table 8-79 Parameters Needed for Getting or Setting Data Breakpoint Statuses

参数名称	类型	位置	描述
Func_Num	ptr_t	Func_ID	必须为 RME_KERN_DEBUG_DBP_MOD。
Cap_Pgtbl	cid_t	Sub_ID	该数据断点的 ASID。
Operation	ptr_t	Param1	操作选项，决定了要进行的操作。
Address	ptr_t	Param2	要打数据断点的地址。

该操作的返回值建议实现为如下：

Table 8-80 Recommended Ret. Impl. for Getting or Setting Data Breakpoint Statuses

返回值	意义
非负值	操作成功，返回值的意义由实现者决定。
RME_ERR_CAP_RANGE	Cap_Pgtbl 的一级/二级查找超出了范围。
RME_ERR_CAP_FROZEN	Cap_Pgtbl 的一级/二级查找的权能已经被冻结。
RME_ERR_CAP_TYPE	Cap_Pgtbl 不是顶层页目录权能，或为空白权能。
其他负值	该操作因硬件原因无法完成，返回值的意义由实现者决定。

8.11 Bibliography

无

Chapter 9 Appendix

9.1 Supporting Special Functionality in RME

RME 可以支持诸多某些其他操作系统提供的特殊功能，诸如 CPU 热插拔、内存热插拔、隔离内核、准虚拟化等等。下面简述它们的实现思路。

9.1.1 CPU Hotplug

CPU 热插拔分为两个功能，一个是热插，也即插入新的 CPU，增加 CPU 的数量；另外一个是热拔，也即从插槽上拔出 CPU，减少 CPU 的数量。RME 对该功能的支持依赖于底层硬件平台提供的硬件级别支持，并且都要用定制的内核功能调用完成。

对于热插，可以在检测到处理器插入后，初始化该处理器并且创建应有的 Init 线程，然后即可使用这些处理器核。

对于热拔则是相反的，需要停止相应处理器的活动，并且其他处理器核不应该再向该处理器发送 IPI。然后，我们才能把这个 CPU 移除。

9.1.2 Memory Hotplug

内存热插拔也分为两部分，一部分是增加内存，另一部分是减少内存。RME 对该功能的支持依赖于底层硬件平台提供的硬件级别支持，并且也都要用定制的内核功能调用完成。

对于增加内存的情况，如果是增加用户态内存，只要将这些内存对应的物理页框加入到某个用户页表中即可。如果需要增加内核内存，那么需要先暂停其他处理器的运行，然后向内核页表中增加新的页面并修改内核页表以反映这一更改，最后再恢复其他处理器的运行。在完成所有操作后还要做一个 TLB 刷新操作。

对于减少内存的情况，如果是减少用户态内存，需要用户态库确定现在这些物理页面没有被映射。此时，可以直接除去这些页面的映射并拔出该内存条。如果是减少内核内存，那么需要首先暂停其他处理器的运行，然后将要拔出的内存上的数据拷贝到空白的物理页面上，并修改内核页表以反映这一更改，最后再恢复其他处理器的运行。在完成所有操作后也要做一个 TLB 刷新操作。

9.1.3 Separation Kernel

由于 RME 是一个微内核操作系统，因此，就像 Barrelfish 那样[1]，可以很方便地在一台物理机器上运行 RME 的多个实例。每个实例可以管理一个或多个 CPU 核，然后在用户态通过多个操作系统共享内存或者网络实现通信即可。这种方式不要求 CPU 间的缓存是同步的，也不要求各个 CPU 含有的功能是一样的，甚至不要求各个 CPU 的指令集和架构是一样的。如果在此种系统中用到了多种架构的处理器，那么就需要针对它们分别移植 RME 和用户态库。

比如，如果存在如下图所示的 SoC 或单板机[1]，那么是可能在全部的核上运行 RME 的，然后通过不同的子系统间共享内存来完成信息传递。

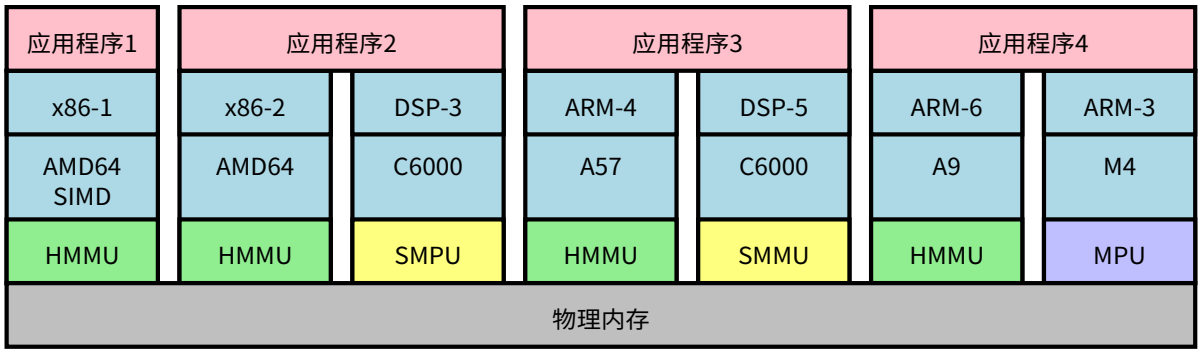

Figure 9-1 Typical Implementation Based on Heterogeneous Multi-core Architectures

上图所示为不同架构共同运行的一个例子。在不同的 NUMA 节点上各运行一个 RME 的实例也是可行的，如下图所示：

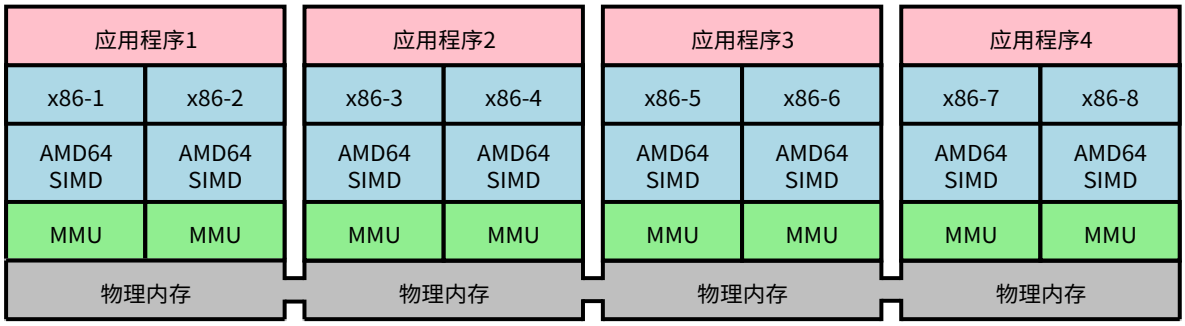

Figure 9-2 Typical Separation Kernel Implementation Based on NUMA Architectures

9.2 Afterwords

9.2.1 Non-scalable Portions of RME in Multi-core Environments

在 RME 中，并非所有的操作都可以互不影响地执行。典型的不能这样执行的操作是多个 CPU 试图同时向一块内存中创建内核对象。下面列出不能并行执行的操作，并且说明其原因。

9.2.1.1 Update of RME_Timestamp

RME_Timestamp 的更新是由一个处理器完成的。在多核处理器上，需要其它核同步这个变量到自己的缓存行。因此，这个操作是不能很好并行化的。但是这并不会对系统的并行度有很大影响，因为大型系统的时钟中断频率可以被配置的较低。

[1] HMMU 指硬件填充的 MMU，SMMU 指软件填充的 MMU

9.2.1.2 Multiple CPUs Try to Create Kernel Object at the Same Location

由于对内核对象的创建都要写入内核对象登记表，因此当多个 CPU 试图竞争地写入表的同一个位置的时候，就会发生大量的缓存行更新，而使得这个操作不能很好地并行化。

9.2.1.3 Multiple CPUs Try to Send to the Same Signal Endpoint Simultaneously

由于这是一个原子比较交换操作，可能需要锁总线，因此多核同时累加实际上是串行完成的。因此，这种情况不能很好地并行化。当然，如果系统具备可以并行进行累加的硬件[1]，这个问题不存在。

9.2.2 Restrictions of RME on 32-bit Processors

RME 在 32 位处理器的情况下，受限于处理器字长长度，对某些功能的实现有所制约。被制约的两个功能是线程创建和安定时间的计算。

9.2.2.1 Thread Creation

RME 中每个线程在绑定时都对应一个 TID。该 TID 计数变量的长度是一个机器字长，而 RME 中因为其他原因还要占用掉该变量的两个位。因此，在 32 位系统上一共仅有 2^{30}-1 个不同的 TID[2]。如果要创建多于这个数量的线程，那么就意味着在绑定时某些线程的 TID 必然要重合。如果用户态软件设计的时候没有考虑这一情况，那么就会造成麻烦。

不过，在一般的 32 位系统中这不是个问题。32 位系统主要都是嵌入式系统，因此很少会有这个数量的线程。在这个前提下，用户态总是可以找一个没有使用的 TID 分配给新的线程。

9.2.2.2 Calculation of Quiescence Period

系统中所有的安定时间都是通过与 RME_Timestamp 的值相比较而计算得出的。该值随着每个时钟嘀嗒自增。RME_Timestamp 的长度总是一个机器字长，因此会存在溢出回滚的问题。比如，在该值为 0x00000000 的时候，有一个操作发生[3]，需要 10 个时间片的安定时间，那么等待到 RME_Timestamp 的值超过 0x0000000B 或更高的时候，就可以对该权能进行下一步操作了。但是，如果我们经过很长一段时间没有做下一步操作，而是等待到该计数器计时到尽头返回 0 值时再进行下一步操作，我们会发现该位置又进入不安定的状态。此时该位置早已安定，这种不安定是变量溢出导致的假象。

因此，在 32 位多核系统下，如果发现经过了安定时间以后对象不安定，那么可以隔一个安定时间以后立即再试一次，直到成功为止。在 64 位系统下由于 RME_Timestamp 永不溢出，因此该问题是不会发生的。

9.2.3 Known Potential Covert Channels in RME

---

[1] 常见的现代架构一般都具备此类硬件

[2] 在 64 位系统中该值为 2^{62}-1，因此不是问题

[3] 比如权能冻结

在 RME 中有一些已知的的潜在隐蔽通道。这些隐蔽通道包括两种，如下列出。

9.2.3.1 Covert Storage Channels

RME 中存在的隐蔽存储通道主要发生在共享权能、内核对象和权能表的两个进程之间。对于共享权能或内核对象的情况，两个进程可以通过对该权能或内核对象的操作试探和改变它们的状态，从而完成信息传递。

此外，在具备协处理器的系统中，如果协处理器上下文保存和恢复的部分实现不当，可能导致协处理器寄存器组被当做一个带宽很大的隐蔽存储通道使用。内核功能调用实现不当也存在同样的问题，在设计上也需要慎重处置。

9.2.3.2 Covert Timing Channels

在 RME 中，时间片和调度是由用户管理的，因此如果用户的调度算法或时间片分配算法编写不当，会导致大量隐蔽定时通道的出现。为了减少这种通道的带宽，可以考虑禁用处理器的用户态高精度计时器指令[1]。关于此部分，在此不详细说明。

9.2.4 Memory Consistency Model

RME 大量地使用了无锁数据结构，因此内存一致性模型显得至关重要。内存一致性模型总的而言可以分为两类：强顺序一致性模型和弱一致性模型。在强顺序一致性模型中，多核处理器上的所有内存访问有一个全局顺序，也即在所有的处理器看来，其他处理器对内存的操作顺序都总是一致的；在弱一致性模型中，一个处理器观察到的其他处理器的操作可能发生乱序，而且如果两个处理器都同时观察第三个处理器，观察到的乱序也可能是不一样的。

在一个弱一致性模型的处理器中，可能发生如下四种乱序：

读-读乱序，也即两个相邻的读操作之间没有一定顺序。

读-写乱序，也即一个相邻的读-写对之间没有一定顺序。

写-读乱序，也即一个相邻的写-读对之间没有一定顺序。

写-写乱序，也即两个相邻的写操作之间没有一定顺序。

实际上，在两种一致性模型之间是一个连续的光谱。某些处理器可能对写-读是乱序的，但对其他情况则是顺序的[2]。

乱序可能导致某些同步标志失效。通常而言，一个共享内存区域都有一个同步标志把守，同步标志的值代表了共享内存区当前的状态。我们先检查并且尝试占有该同步标志，然后再在共享内存区域中进行操作；当操作完成后，我们先把工作收尾，然后再释放该标志位。当乱序发生时，共享内存区域内部的操作可能在占有该同步标志之前就发生了，而收尾工作则有可能拖到同步标志被释放之后。这都破坏了临界区域的性质。因此，在获取标志之后应当加上一个内存屏障，保证对共享内存的操作必须在标志

[1] 比如 x86-64 的 RDTSC

[2] 比如 x86-64

被获取后才开始；在收尾工作之后应当加上一个内存屏障，保证同步标志的释放在收尾工作结束后才进行。虽然 RME 并不使用上面所述的锁来进行内核同步，但是内存一致性仍然会影响内核的同步操作。

为了消除内存重排序对内核的影响，RME 提供了两个宏 RME_READ_ACQUIRE() 和 RME_WRITE_RELEASE()来进行内存操作同步。并且，比较交换原子操作都要求在最后附加一个完全内存屏障。关于具体的实现指导，请参见 7.3.2 和 7.7 所述。

9.2.5 Factors That are Known to Hamper Real-time Performance in RME

在 RME 中已知的影响实时性的主要因素有两个，分别是权能表和页表。

9.2.5.1 Influence of Capability Table Related Operations on Real-time Performance

权能表对实时性的影响主要表现在权能表的创建和删除上。由于 RME 并没有采取内核抢占点的实现，因此权能表在创建和删除时会一次性连续访问大量的内存。如果创建的权能表的大小比较大，而内核中 RME_CAPTBL_LIMIT[1]又未加限制而不能阻止创建时，就会发生内核短暂失去响应的情况。要避免这种情况，需要设置 RME_CAPTBL_LIMIT 以限制权能表的最大大小，或仅允许受信任的用户态根权能管理器在不需要响应实时中断的处理器核上创建和删除权能表。

9.2.5.2 Influence of Page Table Related Operations on Real-time Performance

对于 8.2.1.1 所述的页表实现，动态页可能会产生缺页中断[2]，从而无法保证对动态页数据访问的实时性；对于 8.2.1.2 和 8.2.1.3 所述的页表实现，如果映射的静态页的数量超过了硬件支持的区域数量，那么甚至在静态页上也会引起缺页中断。在这种状况下一个解决办法是不要映射过多的静态页。

对于 8.2.1.4 所述的页表实现，由于快表由硬件填充，因而具有更大的不确定性。如果需要在硬件填充的内存管理单元上获取实时性能，推荐使用支持快表锁定的处理器，这样就可以将实时进程的页面锁定在快表中[3]；如果处理器不支持快表锁定，在映射页面时可以采用快表着色算法，尽量使得实时进程的快表条目不会被非实时进程挤占，或者使得系统中映射的总页面数量不超过快表的大小[4]；如果处理器连快表着色算法都无法实现，还可以着色数据缓存而使实时进程的页表的一部分永存于数据缓存中，一旦发生快表不命中，可以使查找速度大大加快[5]。

9.2.6 Deviations to the C language standard and MISRA coding conventions in RME

[1] 见 7.3.2

[2] 但静态页永远不会

[3] 这是推荐的做法

[4] 这是次好的做法

[5] 这是最坏的做法

9.2.6.1 C Undefined Behavior: Type-punning Pointer-aliasing Exists and is not Strict Aliasing Rule Compliant

类型双关（Type-Punning）指针别名（Pointer-Aliasing）是一种广泛存在于 C 语言[1]实际应用中的用法，它允许两个不同类型的指针引用同一块内存。这在网络协议栈中是非常常见的，因为我们往往需要组装或者拆解网络封包，变换其字节序，等等。在操作系统中这也很常见，尤其是将无符号整数指针与结构体指针来回转换以方便对对象的操作。不幸的是，C 语言的严格别名机制（Strict Aliasing Rule）并不允许两个不同类型的指针引用同一块内存，除非是兼容类型或者从其他类型转换到 char[2]，而且也没有通用的办法绕开这一点。通常而言有三种方法可以避免类型双关，它们分别是：

1. 使用 union 来进行不同类别之间的转换。不幸的是这在 C 中是被定义行为而在 C++中不是，而 RME 的代码由于可能和使用 C++写成的某些库直接链接[2]而需要作为 C++被编译，因此仍然有可能是未定义行为。此外，此种方法对长度在编译时不确定的数组对象是束手无策的，除非牺牲潜在的性能来对它们进行逐一转换。最后，这样使用 union 不符合 union 的设计精神，属于对语言规则的滥用。

2. 使用 char*来进行操作。不幸的是现存编译器对这一点的优化非常糟糕；某些相当优秀的编译器甚至也会把这些操作编译成逐字节访问的指令，对于低质量嵌入式编译器则更是如此。因此，这是一种不好的解决方案。

3. 使用 memcpy 函数在两个不兼容的指针间拷贝内容，操作完后再拷贝回去，寄希望于编译器能够探测这种行为并且优化掉重复的 memcpy。这巧妙地利用了 memcpy 的参数为 char*，而 char*总是可以和任何指针别名的特点。它带来的直接问题是，某些低质量编译器可能不够智能，有时不会把 memcpy 函数优化掉，而导致生成的代码中实际调用了两次 memcpy，从而导致巨大的性能损失。而且一旦 memcpy 没有被优化掉，这可能会导致在函数内部声明的局部拷贝缓冲区被分配在栈上，大大增加栈消耗而导致堆栈溢出。最后，此类对 memcpy 的使用是不符合 memcpy 的设计意图的，属于对语言漏洞的滥用；C 标准对此种做法不持鼓励态度。

有鉴于以上三种方法的问题，RME 并不使用这些方法，而是直接使用类型双关指针别名，而在编译时关掉相关的优化[3]。

9.2.6.2 C Undefined Behavior: The Length of a Byte is Always 8 Bits and the Size of a Char is Always a Byte

C 语言并未定义 char 的长度，它仅仅表示 char 必须等于系统中基本字符集的单个字符的宽度[2]，理论上讲一个 char 类型可以是 32 或 16 个二进制位[4]。现时的所有主流编译器和架构上 char 的长度都总是 8 个二进制位，因此我们在代码中也作此假设以避免不必要的“char 类型长度”配置宏。

---

[1] 指 C90，下同

[2] 如果不希望使用 extern "C"的话

[3] 如 GCC 的-fno-strict-aliasing 选项

[4] 早期的某些嵌入式编译器会这样做

9.2.6.3 C Undefined Behavior: The Actual Representation of a Pointer is Always a Machine Understandable Address

C 语言并未定义指针在内存中应该如何表示。理论上讲，指针变量的实际表示可以是任何东西，只要在解引用时它返回的值是按照 C 语言语义指向的变量的值就可以了[2]。这使得在严格的 C 标准下将指针对齐到某个内存粒度是不可能的，因为无法对其使用掩码来掩蔽掉它的某些位；也无法利用指针的某些不使用的位来存储其他信息。因此，我们假设指针的实际表示总是机器可直接识别的二进制地址，这一点在所有 RME 支持或计划支持的架构上都是成立的。

9.2.6.4 C Undefined Behavior: The Length of a Pointer is Always Smaller than or Equal to the Length of a Machine Word

C 语言并未规定指针的长度。指针的长度可以是任何值，只要按照 C 语言的语义，它能够正常工作即可。完全有可能出现指针的长度比一个机器字长长从而无法放置在长度等于一个机器字长的无符号整数值内部的情况[2]。这种情况现时仅仅出现在几乎所有的 8 位单片机和部分 16 位单片机上，而 RME 从设计之初就不打算支持这些架构，因此我们可以假设指针的长度总是小于或等于一个机器字长。实际上，在 RME 支持或计划支持的所有架构上，指针的长度都总是等于一个机器字长。此外，我们不使用 C 库提供的 uintptr_t 等类型，因为这些类型只有 C99 才有，而且甚至是可选实现的。

9.2.6.5 C Undefined Behavior: Coercing a Pointer to an Unsigned Integer Machine Word Always Keep Its Actual Representation

C 语言并未规定指针在转换到无符号整数后，其实际表示应如何变化。它只规定了，如果先把一个指针转换成一个空间足够大的无符号整数后再转换回去，来回转换后得到的指针在和原指针做比较操作时应当视为相等[1][2]。如果指针在转换到无符号整数后的表示会变化，那么我们会遇到与 9.2.6.3 所述类似的问题。因此，我们假设强制转换指针到一个机器字长的无符号整数值总是不改变它的实际表示，这一点在所有 RME 支持或计划支持的架构上都是成立的。

9.2.6.6 C Undefined Behavior: The Length of External Identifiers are Expanded to 31 Characters

C90 规定外部（extern）变量名最少只有前 6 个字符是有效的。这对任何操作系统来说都太少了。因此，RME 把这个限制放宽到了与内部变量名限制相同的 31 个字符。所有的现代编译器均支持这一点。

9.2.6.7 Deviations to the MISRA-C coding convention

RME 对于 MISRA-C:2012[3]中的所有三类建议都进行了严格检查，并且所有的背离都被批准。

RME 对于 MISRA-C:2012[3]中所述的强制（Mandatory）类别没有任何背离发生。

RME 对于 MISRA-C:2012[3]中所述的要求（Required）类别的背离如下：

[1] 这甚至不禁止转换前和来回转换后得到的表示有差异，只要这两个值在做指针比较时总是按照相等处理即可

Table 9-1 RME Deviations to the Required Section of MISRA-C:2012

条目	解释
1.3	某些如前所述的未定义行为对于实现操作系统这样的底层软件而言是必需的。
5.1	RME 扩展了外部标识符的有效长度到 31 位。这对于操作系统的软件工程而言是必需的。
11.3	类型双关指针别名对于操作系统这样的软件而言是必需的。RME 事实上仅在结构体和子结构体处使用此类双关指针别名。
11.9	RME 不使用 NULL 而使用 RME_NULL。这是一个非常细微的不一致。
14.3	RME 在检查配置是否正确的流程中大量使用了编译时就能确定结果的布尔表达式。这些布尔表达式都被直接送入 RME_ASSERT 宏，确保 RME 被正确配置。这提高了软件的质量。

RME 对于 MISRA-C:2012[3]中所述建议（Advisory）类别的背离如下：

Table 9-2 RME Deviations to the Advisory Section of MISRA-C:2012

条目	解释
2.5	在硬件抽象层，尤其是固件库中，某些宏是作为配置选项出现的。它们可能在当前配置中没有使用，但必须保留以适应未来的软件配置调整，有利于维护工程的一致性和提高代码质量。这些宏都有很好的注释和文档可供参考。
2.7	由于 RME 的硬件抽象层要对所有的受支持硬件定义统一的接口，因此不免会有一些硬件抽象层函数的参数对于某些架构而言是多余的。
11.4	转换一个无符号数到指针对于操作系统这样的软件而言是必需的。
12.4	在 32 位系统上 RME_Timestamp 可能会在自增时溢出并重新从 0 开始。这是正确的实现。在 9.2.2.2 中有关于它的详细的解释。
15.4	RME 在内核内存登记表相关代码中有部分循环会有两个 break 都能将其打断，并且在硬件抽象层的页表查询函数中也会有这样的实现。其他形式的实现可能会使实现复杂度增加而不利于工程维护。在出现此种情况的循环中，注释都详细标注了这样做的理由及各个分支的退出条件，并且各个 break 语句均从上到下依次并列。
15.5	基于与 15.4 同样的理由，RME 的一个函数可能会有多个出口。在这些代码处都进行了良好注释，并且其他可选实现会违反其他的要求或建议，并且破坏工程的整体抽象。
20.5	为了使每个函数和变量的声明都只进行一次（包括其 extern 形式），系统性地防止出现隐式声明错误来提高软件质量，RME 使用了独创的分段包含式头文件。该格式要求使用#undef。#undef 仅用于该格式，而且该格式在整个系统中都被良好遵循，也有对应文档进行详细说明。

9.3 Glossary

Table 9-3 Glossary

Chinese terms	English translation
权能	Capability
组件	Component
协程	Coroutine
守护进程	Daemon
（信号）端点	Endpoint (Signal)
可扩展/可扩展性	Expandable/Expandability
（线程迁移）调用	Invocation (Thread Migration)
页目录	Page Directory
页表项	Page Entry
页表	Page Table
优先级	Priority
进程	Process
安定	Quiescence
可伸缩/可伸缩性	Scalable/Scalability
调度器	Scheduler
信号	Signal
线程	Thread

9.4 Bibliography

[1] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, et al., "The multikernel: a new OS architecture for scalable multicore systems," in Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009, pp. 29-44.

[2] ISO, International Standards Organisation, "Programming Languages—C, International standard, ISO/IEC 9899: 1990 (E)", 1990.

[3] M. I. S. R. Association, MISRA C 2012: Guidelines for the Use of the C Language in Critical Systems: March 2013: Motor Industry Research Association, 2013.