突越・中生・真核（七阶）

Mutate・Mesozoa・Eukaryon(R.VII)

M7M1(Eukaryon) R2T1

Microkernel General-Purpose RTOS (Rev.2 Typ. 1)

Technical Manual

System Features

1.Minimal Yet Efficient

- Less than 5000 lines of kernel code
- Only necessary features are implemented in the kernel level
- Most advanced features are implemented in the user level

2.Broad Architecture Support

- Easy to port across different architectures with minimal assembly
- Supports symmetric or asymmetric multi-processor architectures
- Supports cache-coherent or noncoherent UMA or NUMA systems

3.Comprehensive Capability Management

- Capability-based 3rd generation microkernel
- Access control over individual operations on each over kernel object
- Comprehensive capability granting & revoking mechanisms

4.Flexible Memory Management

- Genuine user-level and kernel-level separation
- Memory access control imposed by page tables or path-compressed tries
- Supports page table to microcontroller MPU metadata conversion
- User-level memory management

5.Advanced Thread and Process Model

- A process can contain multiple threads
- User-level hierachical scheduling and timeslice granting
- Can work as a para-virtualizing hypervisor to run other guest systems

6.Advanced Inter-Process Communication and Hardware Specific Support

- Synchronous communication implemented as synchronous invocations
- Asynchronous communication implemented a asynchronous signals
- Synchronous invocation have minimal overhead
- Asynchronous signals have maximum flexibility
- Interrupts vectored to user level as kernel signals
- Can call architecture-specific functions through kernel function calls

Contents

Chapter 1 Introduction

1.1 Preface

In modern embedded systems, as the needs for computation capability grows, the prevalence of multicore systems grows quickly, and the trend of asymmetric computation is also gaining popularity. In the meantime, as the amount of resource grows, the need for advanced memory management is also growing. However, for multi-core systems, the real-time guarantees are often hampered by race conditions, which calls for lockless kernels; for microcontrollers, their memory protection mechanism and memory layout is different from microprocessors, which calls for a unified programming paradigm to consolidate the different platforms.

In modern high-performance computing systems, the importance of light-weight virtualization is steadily growing. In high-end servers, virtualization facilitates centralized resource management and allocation, and enables new possibilities such as hot-migration; in high-performance embedded systems, virtualization makes it possible 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 required that the virtual machine should be real-time and highly efficient, while maintaining its scalability and expandability.

RME is a general-purpose scalable microkernel 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 mechanisms. RME is designed to run on microcontrollers that have 64kB ROM and 16kB RAM, and servers that have multiple sockets and hundreds of GBs of memory, while still being as efficient as other microkernels.

This manual provides a comprehensive description of RME's system calls from the user's perspective. For detailed information related to a specific architecture, please consult the corresponding manual for details. In this manual, we will review some concepts about operating system and real-time system before we introduce the APIs of RME.

1.1.1 Design Goal and Specs

The design goal of RME is to create the first commercially available open-source microkernel. This microkernel shall have the best flexibility, usability and scalability among all microkernels. As a 3rd-generation microkernel, security and reliability is also a design goal.

RME is a capability-based system. In the system, all operations are controlled by capabilities. All system calls must contain the corresponding capability to a kernel object; thus, the different subsystems of RME can manage different kinds of capabilities, and the managers are very loosely coupled.

1.1.2 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 (security and medical equipment), 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.

1.1.3.2 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 a 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 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 a thread and they can share the same thread stack.

1.1.3.5 Static Allocation

All resource allocations are done at compile time.

1.1.3.6 Semi-Static Allocation

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

1.1.3.7 Dynamic Allocation

At least a part of the resources can be allocated/freed at runtime.

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 Variable)

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 priority: ***seL4 (@2016 Data61/CSIRO)***.

Light-weight scheduling queues: ***RMProkaron (@EDI)***.

System calll interfaces: ***Linux (@The Linux Foundation/Linus Torvalds)***.

Separation kernel implementation: ***Barrelfish (Micro$oft/ETH Zurich)***.

All other references are listed in their respective chapters.

1.2 Forewords

Operating system is a kind of basic software that is responsible for CPU, memory and device management. For real-time operating systems, all operations of the system must be predictable and always meet its deadline. Generally speaking, there are two kinds of real-time systems: 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 the motor controller: the GUI part is soft real-time, an the motor control part is hard real-time.

Generally speaking, almost all real-time systems are embedded systems, which refers to the systems that are specially customized for one application. Embedded applications poses many restrictions on system performance, power, size and environment. Typical embedded systems include highly specialized systems such as programmable logic controllers, flight data computers and rocket computers; there are also systems that are relative general-purpose such as 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 no operating system altogether. With the advent of new microcontrollers and microprocessors, the complexity of embedded systems have greatly increased, which calls for new Real-Time Operating Systems (RTOSes).

The new-generation RTOS are supposed to be stronger than current RTOSes in both reliability, portability, flexibility, and should have inherent design considerations about multi-core support and parallelism. Considering all these requirements, microkernel design is the only choice. Microkernels implement a minimal set of primitives that can help move most operating system services to the user level; should any of these services fail, the fault can be contained within the server boundary and will not propagate to other modules. The different servers can be reboot independent of each other, and can have their own private copies for redundancy, which greatly boosts system reliability.

Additionally, a microkernel design made multi-core parallel design easier due to minimalism of the kernel, thus requiring less synchronization points. This also makes it possible to largely employ read-copy-update (RCU) techniques in the kernel. Most kernel objects and capabilities are aligned to the cache-line boundary, and cache-line contention is reduced as much as possible to avoid unnecessary cache-line sharing. This greatly boost 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 classify 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 system services and software abstraction layers found in common operating systems. 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() loop, in which it calls its task functions one by one. On each entrance of a task it will pick a state to run. All the tasks share the same stack, and the tasks are linked with the kernel statically. The tasks are cooperative and usually does not preempt each other. The tasks does not necessarily need to be reentrant.

The priority support of such operating system is implemented with hardware priorities of the systems, and the IRQ handlers will process everything other than handing the stuff 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 RMSimpron (ultra-light-weight coroutine library) 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 should be deployed on 16-bit and 32-bit machines at most times, and require a system tick timer. These systems do not have genuine kernel space and user space; however, it is possible to configure the MPU to protect some ranges of memory. The hardware abstraction layer of these systems include some simple assembly, which must be modified when porting to other architectures. The porting 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 MPU protection.

In basic hard-real-time systems, a task is always a thread, and can be reentrant. Threads have their own respective stacks. The application code can be either linked with the kernel or 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. Sometimes these systems also have primitive memory management support, which is 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 is required.

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

1.2.4 General-Purpose Operating Systems

General -purpose operating systems are systems that exhibit all natures of a typical operating systems but does not usually have good real-time performance. These systems usually run on 32-bit or 64-bit machines and require a system timer. They enforce separation of user-level and kernel-level, and they require hardware memory management devices such as memory management unit (MMU) or memory protection unit (MPU). They also require a complex customized linker script, and complex porting procedures must be followed when it is needed to run on a new architecture. The porting usually involves system timer, context/protection domain switch, interrupt management and coprocessor management.

Tasks present themselves as processes. One task can contain multiple threads. Due to the existence of virtual memory, there's 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 (e.g. SYSCALL/SYSRET).

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

These systems usually feature two-level memory management; the OS is responsible for low-level page management, and the language runtime libraries are responsible for runtime 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 variant of all the operating systems. The most distinct feature of such systems is the added real-time guarantee, and all kernel execution is completely predictable.

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

1.3 Performance and Specs of RTOSes and Components

There are hundreds of 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 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 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 acceptable. When in evaluation, the system load will also influence the results, thus the system load must remain the same when making measurements.

1.3.1 Kernel Size

Kernel size is a very important 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 fxed number. Usually these figures are tied closely to the detailed configuration. Therefore, to measure the performance, you should measure the kernel size under the minimal kernel configuration, common kernel configuration, and full-featured kernel configuration.[1]

Obtaining kernel size data is as simple as compiling the kernel with a compiler and then using a dedicated binary viewer (such as Objdump) 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 the RTOS system call. The worst execution time refers to the maximum length of execution time under the most unfavorable conditions. The worst case RTOS execution time (WCET) is usually achieved when the longest system call is made and a large number of cache misses and TLB misses occur. RTOSes generally disables interrupts in the execution of system calls; the worst execution time is usually the longest time in which the system disables interrupts, so the impact of WCET on the real-time properties is enormous.

The WCET can be divided into two categories: the first is the WCET of system calls, and the other is the WCET of inter-thread synchronization.

To get the first type of WCET, 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, note the 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 measurement among all measurements.

To get the second type of WCET, jot down the timestamp Ts at the sending side of the communication mechanism, then at the receiving end of the communication mechanism, read the timer get the timestamp Te. The cost measurement for reading the timer is similar to the first-type WCET. The resulting Te-Ts-(Tte-Tts) is the execution time. Repeat this on all communication mechanisms calls, then the WCET will be largest measurement 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 average execution time, and its standard deviation (and sometimes we also use the range) is called the jitter.

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

1.3.2.1 Intra-Process Thread Context Switch Time

The time cost of switching from one thread to another thread in the same process (address space). We use the following method to measure this. In the measurement, except for the method using Te-Ts, it is also possible to use the difference between two Ts divided by 2 (the same applies hereinafter).

There are two cases of thread switching, one case is to switch between threads with the same priority, the other is to wake up a high-priority thread low-priority thread.

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

Process 1 : Thread A	Process 1 : Thread B

Loop forever
{
> Read Ts;
Switch to thread B;
}
	Loop forever
	{
	Read Te;
	> Switch to thread A;
	}

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

Process 1 : Thread A	Process 1 : Thread B

Loop forever
{
> Read Ts;
Wake B up or switch to B;
}
	Loop forever
	{
	Read Te;
	> Sleep forever or switch to A;
	}

1.3.2.2 Intra-Process Thread Context Switch Time

The time cost of switching from one thread to another thread in a different process. The measurement methods and two possibilities are the same as stated above; the only difference is that the two threads are in two different processes.[2]

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

Process 1 : Thread A	Process 2 : Thread B

Loop forever
{
> Read Ts;
Switch to thread B;
}
	Loop forever
	{
	Read Te;
	> Switch to thread A;
	}

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

Process 1 : Thread A	Process 2 : Thread B

Loop forever
{
> Read Ts;
Wake B up or switch to B;
}
	Loop forever
	{
	Read Te;
	> Sleep forever or switch to A;
	}

1.3.2.3 Intra-Process Thread Synchronous Communication Time

The synchronous communication time 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 (RME and some variants of L4), this measurement is not necessary because there's no such need to use synchronous communication in the same process.[2]

Process 1 : Thread A	Process 1 : Thread B

Loop forever
{
> Read Ts;
Send to synchronous endpoint P;
}
	Loop forever
	{
	Read Te;
	> Receive from synchronous endpoint P;
	}

1.3.2.4 Inter-Process Thread Synchronous Communication Time

The synchronous communication time between different threads across different processes. The measurement procedure below suits traditional operating systems well. 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]

Process 1 : Thread A	Process 2 : Thread B

Loop forever
{
> Read Ts;
Send to synchronous endpoint P;
}
	Loop forever
	{
	Read Te;
	> Receive from synchronous endpoint P;
	}

For systems such as RME and L4 variants, the 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.

Process 1 : Thread A	Process 2 : Thread A

Loop forever
{
> Read Ts;
Call synchronous invocation function F;
}
	Function f
	{
	Read Te;
	Return;
	}

1.3.2.5 Intra-Thread Asynchronous Communication Time

The total time to send and receive asynchronous signals in the same thread. The asynchronous signals usually include RME's asynchronous endpoint, or Linux's semaphores & message queues & pipes, or RT-Thread's mailboxes. The measurement method is listed below.[2] In the final measurement result, Ti-Ts is the time cost of send operation, Te-Ti is the time cost of receive operation, and Te-Ti is the total cost.

Thread A
Loop forever { Read Ts; Send to asynchronous endpoint P; Read Ti;

Receive from asynchronous endpoint P; Read Te; }

1.3.2.6 Intra-Process Inter-Thread Asynchronous Communication Time

The time to send and receive asynchronous signal between different threads in the same process. The measurement method is listed 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]

Process 1 : Thread A	Process 1 : Thread B

Loop forever
{
> Read Ts;
Send to asynchronous endpoint P;
}
	Loop forever
	{
	Read Te;
	> Receive from asynchronous endpoint P;
	}

1.3.2.7 Inter-Process Asynchronous Communication Time

The time to send and receive asynchronous signal across processes. The measurement method is the same as stated in 1.3.2.6, with the only difference being that the two threads under test belong to different processes.[2]

Process 1 : Thread A	Process 2 : Thread B

Loop forever
{
> Read Ts;
Send to asynchronous endpoint P;
}
	Loop forever
	{
	Read Te;
	> Receive from asynchronous endpoint P;
	}

1.3.2.8 Page Table Operation Time

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 process's address space. Some MCU 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 time difference is the operation overhead.[1][2]

Thread A

Loop forever
{
Read Ts;
Perform operation;
Read Te;
}

1.3.3 Interrupt Response Time (IRT), Worst-Case IRT and Jitter

Interrupt response time refers to the time between the occurrence of an interrupt and the corresponding processing thread's wakeup. The worst-case interrupt response time (WCIRT) refers to the maximum length that an interrupt response time can reach under the most unfavorable conditions. The WCIRT is usually reached when a large number of cache misses and TLB misses occur during interrupt processing. Interrupt response time is the most important indicator of the RTOS, and it can even be said that everything of the RTOS should be designed around it. This measurement is the most direct reflection of the RTOS's real-time performance.

To obtain the WCIRT, a timer can be read in the first line of assembly of the interrupt vector (cannot wait until the C handler starts execution because register and stack maintenance are also part of the interrupt response time), resulting in a time stamp Ts; read the timer at the first line of code of the interrupt processing thread to get a timestamp Te. The measurement of the timer read cost is the same as above. The resulting Te-Ts-(Tte-Tts) is the interrupt response time. The worst-case interrupt response time is the one with the highest response time among all the interrupt-response tests.

The jitter of this 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 (and sometimes we also use range) is called interrupt response time jitter.

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

Kernel	Thread A

Hardware interrupt handler
{
> Read Ts;
Send to endpoint P from kernel;
}
	Loop forever
	{
	Read Te;
	> 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 operations' completion. The worst-case realistic interrupt response time 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 the realistic IRT, we need some extra hardware to support that kind of 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 the same FPGA to the CPU or motherboard's output pin. First of all, 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 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]:

FPGA (or oscilloscope)	System under test

Loop forever
{
> Send stimulus and start the timer;
Wait until receipt of the response;
Stop the timer;
}
	Loop forever
	{
	Receive signal from I/O;
	Minimal processing routine;
	Send response to I/O;
	}

1.3.5 Input/Output Perfomance

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 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 command's assertion and the data's arrival.

1.3.6 Virtualization Performance

For those operating systems that support virtualization (or paravirtualization) of other operating systems (such as RME), 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 index.

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, 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 repeated here.

1.4 RME System Call Interface

Making system call is a method to use the system's functionality. For RME, this is the only way to use its functionality. System calls are generally implemented by using software interrupts (i.e. SWI and SVC instructions of ARM). They can alternatively be implemented with dedicated instructions (SYSCALL/SYSRET of x86-64). 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 by shared memory or

register contents, 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 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 Call Convention

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 calling C conventions allow up to 4 registers without using stack to pass them. (i.e. MIPS and ARM architecture passes first 4 word-size arguments by registers and the arguments that follow by stack)

Non of RME's system calls use more than 4 registers; RME does not pass arguments in shared memory (Linux and some early variants of L4 will), 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 (the example is based on 32-bit machines; the same goes for 64-bit):

<table>
<tr><td colspan="8">[31 32-bit machine word 0]</td></tr>
<tr><td colspan="4">D1</td><td colspan="4">D0</td></tr>
<tr><td colspan="2">Q3</td><td colspan="2">Q2</td><td colspan="2">Q1</td><td colspan="2">Q0</td></tr>
<tr><td>O7</td><td>O6</td><td>O5</td><td>O4</td><td>O3</td><td>O2</td><td>O1</td><td>O0</td></tr>
</table>

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

System call name	Number	Explanation
RME_SVC_INV_RET	0	Return from an invocation
RME_SVC_INV_ACT	1	Activate the invocation
RME_SVC_SIG_SND	2	Send to a signal endpoint
RME_SVC_SIG_RCV	3	Receive from a signal endpoint
RME_SVC_KERN	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 into another
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 entry and stack of a thread
RME_SVC_THD_HYP_SET	27	Set hypervisor attributes of a thread
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 entry and stack of an invocation port

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 first register (P0)'s D0 field, 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

1.4.3.1 When creating kernel objects, the kernel virtual address must be aligned to the RME_KMEM_SLOT_ORDER order of 2 (See chapter 7 for details on this macro).

1.4.3.2 System call 0-8 will cause a potential context switch. For branch prediction optimization, their system call numbers are contiguous.
1.4.3.3 All system calls will return a non-negative value upon success, and a negative value upon failure.

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.

Chapter 2 Capability Table and Capability Management

2.1 The Concept of Capability

Capabilities are a kind of access permission control certificate originally introduced in multi-user computer systems[1]. It can be understood as an unforgeable token that uniquely points to some resource and carries the operations allowed on the resource. Unix descriptors are a kind of capability to some extent[2]; The access permission of Windows is also a kind of capability. In other words, capabilities are fat pointers to a certain system resource.

We guarantee the system's 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.

第三代微内核普遍采用权能的概念来管理资源。在 RME 中，所有的内核资源都是使用权能来管理的，并且权能全部位于内核空间中的权能表内。每个进程都对应一个权能表，在调用系统调用时，系统查找内核对象就是从该进程的权能表内查找的。每个权能都有一系列的操作标志位，如果某个权能拥有某个操作标志位，那么就可以通过这个权能对该内核对象做此种操作[3]。绝大多数系统，如 seL4 等[5]，都具有操作标志位。但是对于一些其他系统（如 Composite），某种内核对象的操作只有一个，因此不需要单独的操作标志位[4]。

除了权能之外，还有另外一种称为访问控制列表（Access Control List，ACL）[6]的方法。它是由 Lampson 在 1974 年提出的。也可以用来管理权限，但是其访问权限管理粒度较粗，而且表格很容易变得很大。它的优点是权能的授予和撤销相对简单（尤其是撤销）。

在 RME 中，使用权能的概念，可以方便地实现自主访问控制和强制访问控制，也能方便地构建起多级安全机制，并且根据最小特权的原则规划整个用户应用程序的设计。

2.2 权能表的操作和权能的状态

权能表是一种用来存放权能的内核对象。在 RME 中它是一个线性数组，每个数组位置的大小都是固定的 8 个机器字，可以用来存放一个权能。在权能内部，记载着内核资源的确切类型、在内核内存中的确切位置（一个指针）、权能的父权能、权能传递引用计数和权能的当前状态。此外，还有一个时间戳计数器用来在多核并行情况下用来保证权能的操作安定。

2.2.1 权能的类型

在 RME 中，权能一共有 8 种类型（不包括空白权能），如下表所示。关于相应类型权能的信息，请查看相应的章节作为参考。本章只讲述权能表权能相关的内容。

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。

每种权能都代表着对该种内核对象做操作的权力，也代表着该内核对象能实现的功能。需要注意的是，指向权能表的权能（简称权能表权能）是系统的元权能，因为该权能具有修改权能表的权力，能够决定权能表的内容。

2.2.2 权能的传递引用计数和父权能

权能的传递引用计数和父权能是用来跟踪权能传递关系的。权能可以被从一个权能表被传递给另外一个权能表，此时我们把源权能称为父权能，把目标权能称为子权能。子权能的父权能指针要指向父权能，同时父权能的引用计数要增加 1。在移除权能时，要求子权能先被移除，然后父权能才能被移除。指向一个内核对象的权能在初始创建时，其父权能设置为 NULL，其传递引用计数被设置为 0。[4]

对于那些在创建内核对象过程中创建的权能，我们把它们称为根权能；对于那些被权能传递操作创建的权能，我们把它们称作非根权能。根权能的标志是父权能指针被设置为 NULL。

2.2.3 权能的状态

权能的状态有四种：空白、创建中，有效和冻结。这四种状态可以通过系统调用相互转换。

要创建一个权能，调用相应的内核对象创建函数，使用权能表权能指明所被操作的权能表（由于创建操作需要向权能表内部增加权能，修改了权能表，因此需要权能表权能；同理，销毁内核对象也需要权能表权能。），指明需要创建的内核对象的内核虚拟内存地址和一些必要参数即可。此时，被指定的权能表的空白位会先被原子比较交换指令修改为“创建中”，待到创建成功，便会把状态修改为“有效”，否则仍然为“空白”并且返回相应的错误码。在创建权能时，如果要分配内核内存，那么还需要一个内核内存权能。关于内核内存权能，请参见下一章节的描述。

要删除/移除一个权能，首先要冻结这个位置。冻结操作是通过权能冻结系统调用进行的。此时，被制定的权能表的有效位会被修改为“冻结”。如果冻结不成功，那么会返回相应的错误码。如果冻结成功，等到权能操作安定之后，即可调用函数将其删除/移除。其中，删除操作会在移除权能的同时，删除相应的内核对象本身；而移除则仅仅移除权能而不删除内核对象。权能被移除后，就会变回“空白”状态。对于非根权能，只能使用权能移除操作；对于根权能则只能使用对应于该内核对象的删除操作。

关于权能的回收有两种实现。一种实现是 seL4 和 Fiasco.OC 等系统的实现，它们支持操作系统级权能回收的操作，也即只支持删除操作，而在删除时会递归遍历整棵权能传递树，回收所有的被传递的权能。RME 系统则采用另外一种实现，将权能移除和删除分成两个操作。操作系统不会做递归遍历回收的工作，这个工作必须由用户态完成。这样做的好处是不需要加入内核抢占点，但坏处则是用户必须跟踪每一次权能传递而自行负责权能回收。[4]

2.2.4 时间戳计数器与多核并行考量

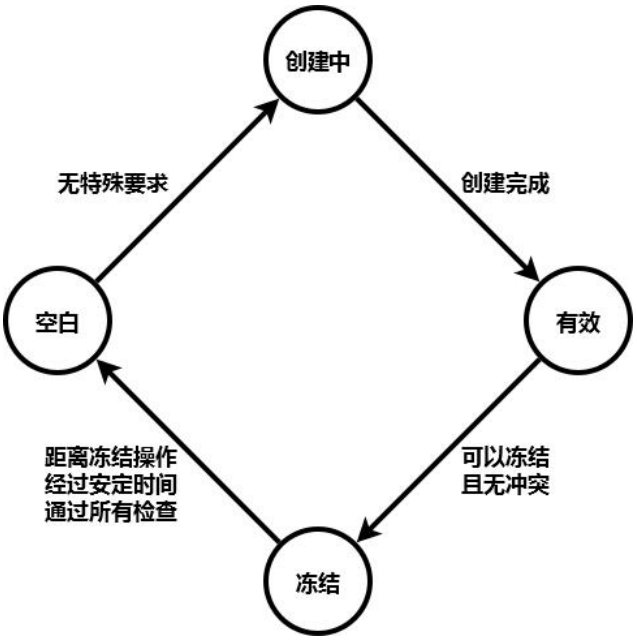
多核环境的执行绪是非常复杂的。内核有一些数据结构，不希望在这样的复杂操作中受到同时修改，它要求所有的操作都是原子的。因此，有两种解决方案：第一种解决方案是采用在每次操作都上锁的方法，第二种方法是在改变权能状态时采用原子操作。采用上锁的方案会引入额外的操作，速度较低，而且还会造成缓存行竞争。缓存行竞争指的是 CPU1 试图修改该缓存行，CPU2 则试图查询该缓存行的状态，从而导致 CPU2 的缓存频繁失效，等效于大大降低了内存的访问速度，导致频繁的权能操作效率严重降低。采用原子操作的方案则较少有这些问题，但是会带来额外的实现难度。在 RME 中，我们采用后一种方法来实现多核并行。

RME 内核采用了大量的比较交换原子操作、原子自加操作构成的读-改-写操作来修改权能。内核还使用一个时间戳计数器来确保没有内核操作的冲突，比如正在一个 CPU 上使用的内核对象在另一个 CPU 上遭到删除等。

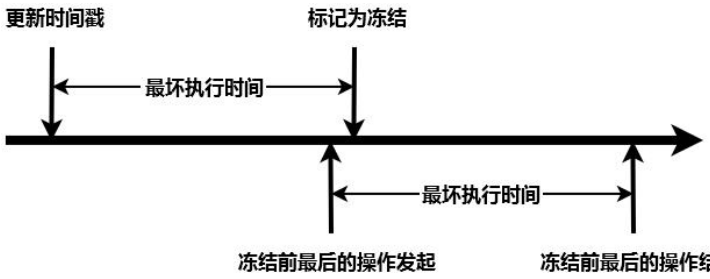
如果我们想要删除或移除权能，我们必须保证权能已经被冻结一段时间。我们把这段时间叫做权能安定时间。这段时间长度是可以在内核中被配置的。它要求被配置为至少比内核的最坏执行时间的两倍要长。这样，我们可以保证在删除或移除权能时，权能已经安定下来，也即所有 CPU 核上涉及到该权能的所有操作都已经完成，就不会有相互冲突的操作了。比如，CPU1 正在试图冻结权能 A，CPU2 则同时试图使用它。CPU2 已经完成了操作的合法性检查，正在进行该操作。每个权能内部的时间戳计数器就是为这个目的而存在的。它会记下上次删除操作或冻结操作开始进行的时间，保证在删除结束之前不会开始创建操作，也保证在完全冻结之前不会开始删除操作。

完整的权能状态转移图如下所示。在创建开始时，进行创建的 CPU 会对该空白槽位进行一个比较交换操作，标志着创建开始，该槽位被占用，直到创建完成为止才把该槽位标注成有效状态，从此时开始可以使用该权能；如果该权能没有被引用，在用毕后想要销毁，那么需要先将其冻结。冻结后，等待安定时间过去，就可以进行移除或删除了。

在删除或移除权能时，我们必须先完成所有的检查，确认该权能可以删除或移除，之后通过比较交换原子操作变更当前槽位的状态到空白来保证只有一个 CPU 核上的操作能够进行下去，最后再执行真正的后续操作，也即内核对象销毁（删除）或父权能解除引用（移除）。在 RME 中，删除或移除操作完成后，不需要再次等待一个安定时间，就可以立即开始新的创建操作。这样的原因是，RME 在修改槽位状态之前已经把后续操作所需的信息保存到了本地变量，在修改之后不再会访问该权能槽位，而是利用保存的信息进行后续操作。在 Composite[4]中，在修改后仍然有可能在后续过程中使用到该槽位上的信息，因此在这里还要插入一个安定时间，防止这个位置被新创建的权能覆盖。



安定时间要求至少是内核最坏执行时间两倍的原因如下：在调用权能冻结操作时，我们会先更新权能的时间戳，然后再通过原子操作标记该权能为被冻结。因此我们有如下的时间线：



从该时间线中可以可以看出，从更新时间戳到冻结前的最后操作结束最多可能经过两个内核最坏执行时间。在实际使用中，推荐将安定时间至少配置为内核最坏执行时间的 10 倍（高出一个数量级），因为在工程中对最坏执行时间的估算往往只有数量级是准确的。

2.2.5 权能表结构

权能表可以被组织成多级结构。组织的具体方法是，在权能表内部放入权能表权能，这样权能表就可以组织成一个基数树。在对内核对象做操作时，需要传递一个权能号，指定这个权能在当前进程的权能表内部的位置。权能号最多可以编码两级查找。我们把一级查找可以达到的范围称为主权能表，把只有二级查找才能到达的范围称为扩展权能表。在 32 位系统下，权能号是一个 16 位的值；在 64 位系统下，权能号是一个 32 位的值。在更高位的系统下，其权能号的编码方法如下：

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

可以看出，在 32 位系统下，单个权能表中最多可以有 2^7=128 个权能，在 64 位系统下这一个值则为 2^{15}=32768。在系统中这个值由宏 RME_CAPID_2L 代表。通常而言，建议的单个权能表内权能数目最好不超过 128（32 位系统下一个权能表占据一个 4kB 页），或者不超过 32768 个（64 位系统下一个权能表占据一个 1MB 页）。

与 seL4 等系统不同，RME 的权能表不支持基于基数树的无限级别查找。也即，扩展权能表内的权能表权能指向的权能表中的权能，不被认为在扩展权能表之内，无法通过扩展权能表被直接使用。如果要使用的话，必须通过权能传递调用传递到扩展权能表之内，才可以通过扩展权能表被调用。

2.3 权能表功能列表

与权能表有关的内核功能如下：

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
RME_SVC_CAPTBL_REM	系统调用	权能移除

权能表权能的操作标志如下：

标志	类型	用途
RME_CAPTBL_FLAG_CRT	位[0]	允许在该权能表中创建权能。

RME_CAPTBL_FLAG_DEL	位[1]	允许删除该权能表中的权能。
RME_CAPTBL_FLAG_FRZ	位[2]	允许冻结该权能表中的权能。
RME_CAPTBL_FLAG_ADD_SRC	位[3]	允许该权能表在权能传递作为来源表。
RME_CAPTBL_FLAG_ADD_DST	位[4]	允许该权能表在权能传递作为目标表。
RME_CAPTBL_FLAG_REM	位[5]	允许移除该权能表中的权能。
RME_CAPTBL_FLAG_PROC_CRT	位[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	位[7]	允许用该权能表替换某进程的权能表。

关于上表中的位[6]和位[7]，请参看后续进程管理有关章节。

2.3.1 创建权能表

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：

参数名称	类型	位置	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。
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。

该操作的返回值可能如下：

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 删除权能表

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：

参数名称	类型	位置	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：

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 权能传递

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：

参数名称	类型	位置	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看下章内核内存权能所述。

该操作的返回值可能如下：

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 权能冻结

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：

参数名称	类型	位置	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：

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_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 权能移除

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（对根权能应当使用删除操作），而且必须不被引用。如果一个权能被移除，那么在安定时间之后相应的空白位置才能够被重新使用。注意，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（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）。移除一个权能需要如下参数：

参数名称	类型	位置	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：

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。

本章参考文献

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

[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.

第三章 页表和内存管理

3.1 内存管理概述

内存管理指的是对物理内存进行的分配和回收操作。通常而言，要想保证内存管理的绝对安全，那么硬件上的支持就是必须的。RME 操作系统的内存管理支持 MPU 环境和 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：

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 页表的操作和结构

页表是由一系列页目录组成的多层基数树结构。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 的解决方法是一致的[1]。

3.2.1 内核内存和用户内存

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（静态或半静态分配）的，并且不可修改。这使得 RME 完全不需要内存动态类型机制（这种机制被 Composite、seL4 等系统采用[1][3]），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。

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，不再允许系统凭空地创造物理内存页框。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：

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 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。这样就限制了某些组件耗尽内核内存。

3.2.2 页目录的属性

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 映射起始地址

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格式（MMU 系统一般也不支持此格式）。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 顶层页目录标志

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 页目录大小级数

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 页目录数量级数

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 页目录的基本操作

在页目录上一共有六种基本操作，分别如下：

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。
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。若要获得更多信息，请参看下面两节的解释以理解具体差别。

3.2.4 内存管理单元下正常多级页表的实现

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，其第一级页目录是固定的 512 个槽位，每个槽位代表 2^{39} 字节；第二级页目录也是固定的 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。而且，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。

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（只要大小和数量级数合适），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。

一些常见的 MMU 的特性如下：

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
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	2 或 3 级	4kB，64kB，1MB，16MB	TLB 部分表项手动锁定

ARMv8-M（AArch64）	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

3.2.5 内存保护单元下路径压缩页表的实现

在内存保护单元下，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。为此，我们必须在页目录的线性表结构上附加一段额外数据，用来记录该页目录的映射起始地址、大小级数和数量级数。

3.MPU 不能进行虚拟地址转换。因此，我们使用的虚拟地址是等于物理地址的。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。

4.由于我们需要更新 MPU 元数据，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。如果我们对于每个页表维持一份单独的 MPU 元数据，我们就不可能把一个页目录映射到多个页表中，因为这会致使我们在修改底层页表时需要修改多个上层 MPU 元数据表示，而这在实时性上是不可接受的，在数据结构上实现也是很困难的。如果我们不这样做，我们就只能软件模拟 MMU 机制。这种方法的具体做法是，每次切换进程都清空 MPU 元数据，然后每一次对新页的内存访问都会触发使用内存保护错误中断向量。我们在中断向量中手动查找页表，然后来逐个添加 MPU 元数据。

一些常见的 MPU 的特性如下：

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	统一组织	128B-1GB	对齐到大小	8 个子区域
ARM V8-M	0 到 16	统一组织	128B-4GB	无	无
Tensilica L106	16 或 32	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
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 仅在顶层页表处放置 MPU 元数据

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。这种实现的最终效果如下：

操作	制约或缺点
激活进程	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（而且也不可能有子页目录）。在顶层 MPU 元数据中添加这个子页目录本身的 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（其子页目录必须从本页目录中提前析构）。在顶层页目录的 MPU 元数据中删除这个子页目录的 MPU 元数据。
内存消耗	仅在顶层页目录有 MPU 元数据。

这个实现是推荐的实现。这个实现最大限度地考虑了提高 MPU 架构下的效率，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。其更新 MPU 元数据的速度也是很快的。

3.2.5.2 使用软件模拟页表读取机制和快表

这种做法把 MPU 的区域寄存器组看作是快表（Trans-Lookaside Buffer），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 MPU 寄存器。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 寄存器组，这相当于 MMU 架构下的 TLB 刷新。

操作	制约或缺点
激活进程	无制约。
更换进程页表	清空 MPU 寄存器中的元数据，准备重建。
切换进程	清空 MPU 寄存器中的元数据，准备重建。
创建页目录	无制约。
删除页目录	无制约。
映射内存页	如果被操作的是当前进程，清空 MPU 寄存器中的元数据，准备重建。
移除内存页	如果被操作的是当前进程，清空 MPU 寄存器中的元数据，准备重建。
构造页目录	如果被操作的是当前进程，清空 MPU 寄存器中的元数据，准备重建。

析构页目录	如果被操作的是当前进程，清空 MPU 寄存器中的元数据，准备重建。
内存消耗	整个系统没有单独的 MPU 元数据，将 MPU 寄存器组用来模拟 TLB。

这个实现保留了和 MMU 系统最大的兼容性，也是 uCLinux 的实现方法。它对页表的构造顺序没有要求，而且多个页表可以以任意方式共享一部分。但是，由于每次进行进程切换（尤其是进程间通信）时都要清空整个 MPU 寄存器组，因此性能非常差，而且不可预测。

3.3 页表功能列表

与页表有关的内核功能如下：

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：

标志	类型	用途
RME_PGTBL_FLAG_ADD_SRC	位[0]	允许该页目录在页框传递作为来源目录。
RME_PGTBL_FLAG_ADD_DST	位[1]	允许该页目录在权能传递作为目标目录。
RME_PGTBL_FLAG_REM	位[2]	允许移除该页目录中的页框。
RME_PGTBL_FLAG_CON_CHILD	位[3]	允许该页目录在页表构造中作为子页目录。
RME_PGTBL_FLAG_CON_PARENT	位[4]	允许该页目录在页表构造中作为父页目录。
RME_PGTBL_FLAG_DES	位[5]	允许析构该页目录。
RME_PGTBL_FLAG_PROC_CRT	位[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	位[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]，请参看后续进程管理有关章节。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 创建页目录

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：

参数名称	类型	位置	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。
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。
Size_Order	ptr_t	P1.Q0	该页目录的大小级数（大小指每个页表项代表的内存页大小）。
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：

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 删除页目录

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，也即其全部页表项位置应该都是空白的。删除页目录需要以下几个参数：

参数名称	类型	位置	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。
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：

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 不是权能表权能。
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 映射内存页

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：

参数名称	类型	位置	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：

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 移除内存页

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：

参数名称	类型	位置	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：

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 构造页目录

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。构造页目录以下几个参数：

参数名称	类型	位置	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_PGTBL_CON。
Cap_Pgtbl_Parent	cid_t	P1	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。
Cap_Pgtbl_Child	cid_t	P3	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。

该操作的返回值可能如下：

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 或 Cap_Pgtbl_Child 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。

3.3.6 析构页目录

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：

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

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 内核内存功能列表

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：

标志	类型	用途
RME_KMEM_FLAG_CAPTBL	位[0]	允许在该段内核内存上创建权能表。
RME_KMEM_FLAG_PGTBL	位[1]	允许在该段内核内存上创建页目录。
RME_KMEM_FLAG_PROC	位[2]	允许在该段内核内存上创建进程。

RME_KMEM_FLAG_THD	位[3]	允许在该段内核内存上创建线程。
RME_KMEM_FLAG_SIG	位[4]	允许在该段内核内存上创建信号端点。
RME_KMEM_FLAG_INV	位[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：

参数名称	类型	位置	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（扩展的）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：

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

N:C（Ext_Flags）的具体意义如下：

位段范围	位段意义
高半字清零其最后六位	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	内核内存地址下限的低半字，对齐到 64Byte。
低半字的最后六位	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。上限不得等于或者低于下限，否则会返回错误。此外，如果传入的上限等于下限，那么也是不合法的。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。

本章参考文献

[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

第四章 进程和线程管理

4.1 进程和线程概述

4.1.1 进程概述

在 RME 中，一个进程指的是拥有一个独立地址空间（页表）和一个权能表的最小实体。页表决定了进程的地址空间，权能表则决定了线程的权限。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。各个进程的权能表都是相互独立的命名空间，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。这样组织的进程在被用作轻量级虚拟机时，可以提供近乎物理机的性能[4]。

4.1.2 线程概述

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，调度系统主要有四种设计策略，分别如下：

第一种是完全的调度器在内核内部。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内部。采用此类实现的典型系统包括 Linux，VxWorks。此类系统是非常传统的，在此不加叙述。

第二种是完全的调度器在内核外部，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，并且可以实现闲置窃取调度（Slack

Scheduling）。但是，这个方法的代价太高了，并且很少有实际用例。这是最原始的用户态调度设想，但是其额外开销过高而没有使用价值。

第三种是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，我们没办法在高优先级结束之后立即执行低优先级线程。此时，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我们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]。

第四种是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。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。在 RME 中我们实现了一个高效的内核队列维护器，将这种影响降低到最小。内核仍然要给线程发送调度器事件来配合用户态调度。

RME 系统不在内核态实现诸如线程本地存储等其他功能。这些功能会被用户态库实现。

每个线程都具有一个线程标识符（TID）。TID 的分配是递增的，并且不会被二次分配。在 32 位系统下，从系统上电开始，累计允许创建的最大线程数为 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（基本上是微控制器或低端微处理器），因此这个限制是没有实际问题的。在 64 位系统下这个限制是 2^{64-2}，实际上也是足够使用的。

在 RME 中，每个线程都有一个（抢占）优先级，优先级的数值越大，则线程的优先级越高（这与其他系统是相反的！）。优先级的数量由宏 RME_MAX_PREEMPT_PRIO 配置，其最大数量最高可以被配置为字长的一半，系统中的优先级为从 0 到 RME_MAX_PREEMPT_PRIO-1。比如 32 位系统中，优先级的最大数量为 2^{16}=65536，优先级为 0-65535。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到提高被操作线程的优先级上限以上。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]。

4.2 进程的操作和状态

4.2.1 进程的创建和删除

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。进程在 RME 中仅仅起到一个容器的作用，它不会被调度。进程在 RME 中没有独立的状态，销毁一个进程中所有的线程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（详见同步通信机制）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 更改进程的权能表或页表

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。动态更换会立即生效。

4.3 线程的操作和状态

4.3.1 线程操作总览

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：

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。

如果线程在执行过程中出现了一次访存错误，那么线程会转换到 RME_THD_FAULT 状态，并且向其父线程发送一个调度器事件。要解除这个状态，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。

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（详见异步通信机制）。

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，那么线程将仍然会维持在 RME_THD_FAULT 状态下。

4.3.2 线程的创建和删除

当创建线程时，需要指明线程所在的进程和它的线程迁移调用堆栈大小（详见同步通信机制）。其他的线程属性是通过其他内核调用设置的，在线程创建时这一点并不体现出来。

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。（详见同步通信机制）。

4.3.3 把线程绑定到某 CPU 和解除绑定

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和线程的父线程。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 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。

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。

与那些线程在创建时就被绑定到某处理器的系统，如 Composite 等[1]不同，RME 的线程可以动态更换绑定。

4.3.4 设置线程的执行属性

在完成线程绑定后，我们需要设置线程的入口和堆栈。这两个值在会被传递给线程的寄存器组，在线程第一次运行时，用户态库根据这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，这两个值都是虚拟地址。

4.3.5 设置线程的虚拟机属性

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。当下次切换到该线程时，将会从该处加载寄存器组。

4.3.6 线程分配时间片，修改优先级和运行

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（Init 线程的时间片为 RME_THD_INIT_TIME，是无限的），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。

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，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：

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。

几个宏的意义分别如下表所示：

宏名	意义
RME_THD_MAX_TIME	为处理器字长可表示的最大数值，在 32 位系统下为 0xFFFFFFFF。
RME_THD_INF_TIME	为 RME_THD_INIT_TIME-1，在 32 位系统下为 0xFFFFFFFE。
RME_THD_INIT_TIME	为 RME_THD_INIT_TIME>>1，在 32 位系统下为 0x7FFFFFFF。

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。第三种传递是回收传递，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。三种传递的规则如下表所示。

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

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

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。

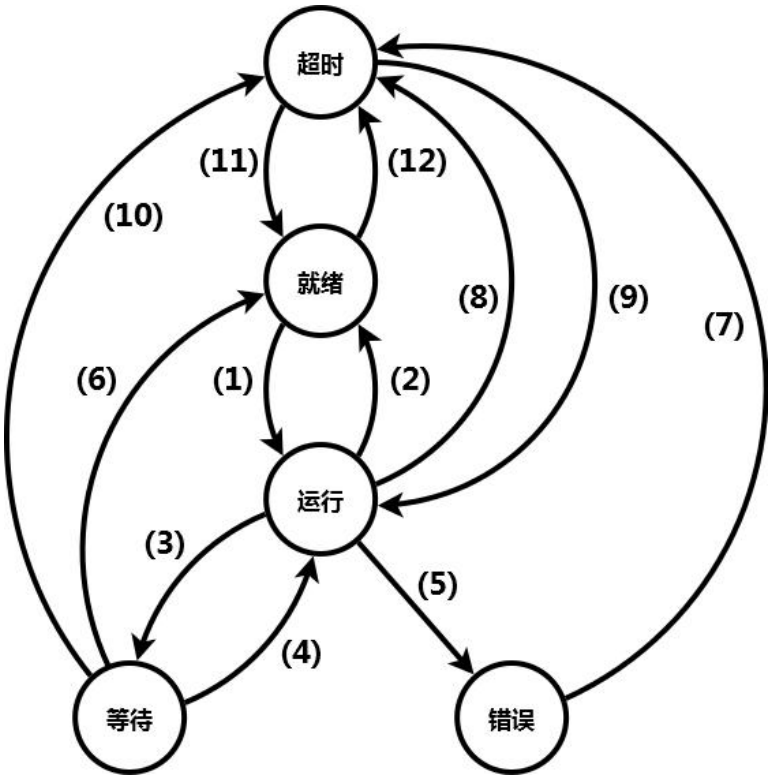
“S”代表源线程一定会超时。

“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 线程调度总览

最后，完善的线程状态转移图如下所示：



图中各个数字标号的意义如下所示：

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，因此有运行态转入超时态。
(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，因此进入运行状态。
(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，因此进入就绪状态。
(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，因此直接转入超时态。

4.4 进程功能列表

与进程有关的内核功能如下：

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（顶层页目录）

进程权能的操作标志如下：

标志	类型	用途
RME_PROC_FLAG_INV	位[0]	允许在该进程内创建线程迁移调用。
RME_PROC_FLAG_THD	位[1]	允许在该进程内创建线程。
RME_PROC_FLAG_CPT	位[2]	允许替换该进程的权能表。
RME_PROC_FLAG_PGT	位[3]	允许替换该进程的页表（顶层页目录）。

关于上表中的位[0]，请参看后续同步通信机制有关章节。

4.4.1 创建进程

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：

参数名称	类型	位置	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。
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（顶层页目录）。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。

该操作的返回值可能如下：

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 删除进程

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：

参数名称	类型	位置	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：

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（内部有线程或线程迁移调用）。

4.4.3 更改进程的权能表

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：

参数名称	类型	位置	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：

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 更改进程的页表

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：

参数名称	类型	位置	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（顶层页目录）的进程。该权能号可以是一级或者二级查找编码。
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（顶层页目录）。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：

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（顶层页目录）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 线程功能列表

与线程有关的内核功能如下：

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（入口和栈）
RME_SVC_THD_HYP_SET	系统调用	设置线程的虚拟机属性（寄存器保存位置）
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：

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

4.5.1 创建线程

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，然后返回其线程 ID。新创建的线程是没有绑定到任何 CPU，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：

参数名称	类型	位置	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。
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：

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_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。
	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 删除线程

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：

参数名称	类型	位置	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：

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。
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 设置线程执行属性

该操作会设置线程的执行属性，也即其入口和栈。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。设置线程执行属性需要以下几个参数：

参数名称	类型	位置	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_THD_EXEC_SET。
Cap_Thd	cid_t	P1	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	P2	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	P3	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。

该操作的返回值可能如下：

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 设置线程虚拟机属性

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（线程刚刚创建时），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：

参数名称	类型	位置	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：

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 将线程绑定到某 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。设置线程执行属性需要以下几个参数：

参数名称	类型	位置	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_THD_SCHED_BIND。
Cap_Thd	cid_t	P1	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	P2	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。
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：

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。
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。
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 更改线程优先级

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：

参数名称	类型	位置	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：

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_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 解除线程对某 CPU 的绑定

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。解除绑定需要如下几个参数：

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：

参数名称	类型	位置	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：

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（其子线程未全部解除绑定）。
RME_ERR_PTH_INVSTAT E	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 接收线程的调度器事件

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（处于倒数第二个二进制位）和线程标识符（其他后续位）两部分组成。接收一个线程的调度器事件需要如下参数：

参数名称	类型	位置	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，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：

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_INVSTAT E	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 传递运行时间片

该操作被用来在线程之间传递时间片。时间片的传递者（源线程）和接收者（目标线程）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：

参数名称	类型	位置	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：

返回值	意义
大于 0 的值	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_Sched 被绑定到了和调用本函数的 CPU 不同的 CPU。
RME_ERR_PTH_OVERFLOW	接收该时间片的线程的时间片已满（再接收的话就会超过或等于系统允许的最大值 RME_THD_MAX_TIME）。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 切换到某线程

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、绑定于同一处理器，而且必须处于就绪（RME_THD_READY）状态。如果试图切换到自身，那么也会成功，此时相当于不做操作。要进行线程切换，需要如下参数：

参数名称	类型	位置	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_THD_ARBITRARY。
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：

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 未被绑定或被绑定到了和其他 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。

本章参考文献

[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.

第五章 同步通信和异步通信机制

5.1 同步通信和异步通信概述

5.1.1 同步通信概述

操作系统中的同步通信机制是用来同步两个并行或并发进程的一种机制。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调用（或说是本地过程调用），其主要机制是使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 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。其表观结果好像是调用了一个进程内部运行的函数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。

线程迁移调用的设计参考了 Composite[1]。

5.1.2 异步通信概述

操作系统中的异步通信机制是用来协调操作系统中的生产者-消费者问题的。通常而言这样的机制有邮箱、消息队列等。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，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总是阻塞的。信号只携带数目信息（它只有一个计数器），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完成对这些信息的传递。

5.2 同步通信操作

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。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。

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（不管是在同一个 CPU 上还是在不同的 CPU 上），那么都会返回错误码。

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；如果线程在进行迁移调用的过程中发生了一个访存错误，那么它并不会直接停止执行，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。（在线程执行本身代码时错误才会试图终止该线程）。

5.3 异步通信操作

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（不管是在同一个 CPU 上还是在不同的 CPU 上），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情况，因为这很难协调。

需要注意的是，当一个线程已经阻塞在了多个 CPU 试图给同一个端点发送信号时，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。

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。

5.4 同步通信功能列表

与线程迁移调用有关的内核功能如下：

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（入口和栈）
RME_SVC_INV_ACT	系统调用	激活（进行）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：

标志	类型	用途
RME_INV_FLAG_SET	位[0]	允许设置该线程的执行属性。
RME_INV_FLAG_ACT	位[1]	允许在该线程在绑定操作中作为子线程。

注意，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。详见下文所述。

5.4.1 线程迁移调用创建

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：

参数名称	类型	位置	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。
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：

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 线程迁移调用删除

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：

参数名称	类型	位置	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：

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 线程迁移调用执行属性设置

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：

参数名称	类型	位置	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_INV_SET。
Cap_Inv	cid_t	P1	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。

该操作的返回值可能如下：

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 线程迁移调用激活

该操作会进行一个线程迁移调用。如果要进行线程迁移调用，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：

参数名称	类型	位置	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。
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：

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（其他线程正在进行调用）。
	两个 CPU 试图同时进行这个调用，失败的 CPU 返回该值。
RME_ERR_SIV_FULL	线程的迁移调用栈已经满了，不能再进行调用。

5.4.5 线程迁移调用返回

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，而后者则被放在寄存器中隐式传递。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：

参数名称	类型	位置	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：

返回值	意义
0	操作成功。如果成功，该迁移调用即停止执行，该返回值无意义。
RME_ERR_SIV_EMPTY	试图在没有线程迁移调用的情况下调用该函数。

5.5 异步通信功能列表

与信号端点有关的内核功能如下：

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：

标志	类型	用途
RME_SIG_FLAG_SND	位[0]	允许向该信号端点发送。
RME_SIG_FLAG_RCV	位[1]	允许从该信号端点接收。

5.5.1 信号端点创建

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：

参数名称	类型	位置	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。
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。
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：

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 信号端点删除

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

参数名称	类型	位置	描述
Svc_Num	ptr_t	N	必须为 RME_SVC_SIG_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_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：

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 向端点发送信号

该操作会向一个信号端点发送信号。向一个信号端点发送信号需要如下几个参数：

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

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，因为信号计数的上限为 2^{32}-1。

5.5.4 从端点接收信号

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：

参数名称	类型	位置	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。

该操作的返回值可能如下：

返回值	意义
非负值	操作成功。该值为接收完成后还剩余的信号的数量。
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 RME_SIG_FLAG_RCV 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。
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。

本章参考文献

[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.

第六章 内核功能调用机制和内核异步信号

6.1 内核调用机制概述

由于 RME 微内核的通用代码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。

内核调用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 内核异步信号概述

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 内核调用机制功能列表

与内核调用机制有关的内核功能如下：

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：

标志	类型	用途
所有位	位段	内核功能范围号。允许调用在这个范围内的内核功能。注意不要内核功能号和系统调用号相混淆。在填充时使用 RME_KERN_FLAG(HIGH,LOW)宏，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的范围。

6.3.1 内核调用机制初始创建

关于内核调用机制的初始创建，见下章所述。

6.3.2 内核调用激活

该操作会执行一个内核调用函数，并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：

参数名称	类型	位置	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	内核功能号。
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：

返回值	意义
非零值	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。

6.4 内核异步信号功能列表

6.4.1 内核信号端点初始创建

关于内核信号端点的初始创建，见下章所述。

6.4.2 从内核信号端点接收信号

接收信号的方法和调用和普通信号端点是一样的，请参见上章所述。

本章参考文献

无

第七章 移植 RME 到新架构

7.1 移植概述

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 的所有代码都用相对符合 MISRA C 规范的 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 移植前的检查工作

7.2.1 处理器

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，以克服软中断方式进行系统调用的开销，使使用一个全功能操作系统成为可能；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 64kB 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。RME 也不支持硬件堆栈机制，堆栈必须是由软件实现的（也即堆栈指针可以由用户修改，堆栈实现在内存中），而不能在处理器内部通过硬件实现（后者是 PIC 单片机等少数架构的典型实现方式）。

7.2.2 编译器

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、ewxxx、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，因此保证了最大限度的兼容性。

在使用编译器时，要注意关闭编译器的（激进的）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。某些优化有可能导致中断处理向量被整个优化掉，引起内核无法工作，也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 汇编器

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 语言调用。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 链接器

RME 要求链接器必须具备链接多个对象（.o）文件到一个中间对象文件（.o）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 调试器

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 RME 架构相关部分介绍

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。其对应的头文件在 Include/Platform/CortexM，其他架构以此类推。

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 Platform/Template 和对应的 Include/Platform/Template 下可以找到一套空的模板文件，可以从这里开始 RME 的移植工作。

7.3.1 类型定义

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

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 宏定义

其次，需要移植的是 RME 的宏定义。RME 的宏定义一共有如下几个：

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，此时用这个宏定义来处理它。 例子：#define EXTERN extern
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 __forceinline
RME_CPU_NUM	系统中的 CPU 数量。 例子：#define RME_CPU_NUM 2
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
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_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（比如 x86-64），这里填写内核允许的最小大小（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）。 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（单位为处理器字长，按 Byte 计算）对应的 2 的方次。比如，使用 32 位（4Byte 字长）处理器，内核内存分配的最小粒度为 64Byte，那么这个位置要填写的数字就是 $\log_2(64/4) = 4$。 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（仅在使用 MPU 时存在）的大小也为 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（仅在使用 MPU 时存在）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（如最新的 x86-64 可以有上百 TB 内存），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 （小内存） #define RME_KOTBL ((ptr_t*)0xFFFF800001000000)（大内存）

7.3.3 架构相关结构体

RME 的架构相关结构体一共有两个，分别如下：

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。

这两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 汇编底层函数

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：

函数名	意义
__RME_Disable_Int	禁止处理器中断。
__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 系统中断向量

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：

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 其他底层函数

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 内核调试打印函数

函数	意义
__RME_Putchar	打印一个字符到内核调试控制台。

7.3.6.2 原子操作与特殊操作函数

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。
__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 初始化、启动与 CPUID 函数

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。
__RME_Reboot	重新启动内核。
__RME_Shutdown	关闭处理器系统。
__RME_CPUID_Get	得到当前 CPU 的 CPUID。

7.3.6.4 寄存器组相关函数

函数	意义
__RME_Get_Syscall_Param	从寄存器组中得到系统调用参数。
__RME_Get_Inv_Retval	从寄存器组中得到线程迁移调用的返回值。
__RME_Set_Syscall_Retval	向寄存器组中设置系统调用的返回值。
__RME_Set_Inv_Retval	向寄存器组中设置线程迁移调用的返回值。
__RME_Thd_Reg_Init	初始化线程的寄存器组。
__RME_Thd_Reg_Read	读取某线程的寄存器组。
__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	初始化线程的浮点寄存器组。
__RME_Thd_Cop_Save	保存线程的浮点寄存器组。
__RME_Thd_Cop_Restore	恢复线程的浮点寄存器组。
__RME_Inv_Reg_Init	初始化线程迁移调用的寄存器组。
__RME_Inv_Cop_Init	初始化线程迁移调用的浮点寄存器组。

7.3.6.5 内核功能调用函数

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

7.3.6.6 页表相关函数

函数	意义
__RME_Pgtbl_Set	切换当前使用的页表（顶层页目录）。
__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	删除前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 类型定义、宏定义与汇编底层函数的移植

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。

类型	意义
s8	一个有符号八位整形。 例如：typedef char s8;
s16	一个有符号十六位整形。 例如：typedef short s16;
s32	一个有符号三十二位整形。 例如：typedef int s32;
s64	一个有符号六十四位整形。 例如：typedef long s64;
s128	一个有符号一百二十八位整形。 例如：typedef long long s128;
u8	一个无符号八位整形。 例如：typedef unsigned char u8;
u16	一个无符号十六位整形。 例如：typedef unsigned short u16;
u32	一个无符号三十二位整形。 例如：typedef unsigned int u32;
u64	一个有符号六十四位整形。

	例如：typedef unsigned long u64;
u128	一个有符号一百二十八位整形。 例如：typedef unsigned long long u128;

对于宏定义，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，依该表格说明填充这些宏定义即可。

接下来说明对于汇编底层函数的移植过程。

7.4.1 __RME_Disable_Int 的实现

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

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。

7.4.2 __RME_Enable_Int 的实现

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

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。

7.4.3 _RME_Kmain 的实现

函数原型	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。

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。

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，因此可以把这个页表创建在内核对象分配内存中，这样不占用额外的内存空间。如果这个工作没有被进行，那么这个工作需要由本函数正确实现。

7.4.4 __RME_Enter_User_Mode 的实现

函数原型	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。

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 即可。该函数将永远不会返回。

7.5 系统中断向量的移植

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，另一部分是系统错误处理中断向量本身。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 中断向量进入和退出，以及架构相关结构体部分

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 定时器中断向量

在定时器中断处理向量中，需要调用如下函数：

函数原型	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。

7.5.1.2 系统调用中断向量

在系统调用中断向量中，需要调用如下函数：

函数原型	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 系统错误处理中断向量

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下节。

函数原型	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 系统错误处理中断向量

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（比如未定义指令、访存错误等等），那么直接调用由系统提供好的如下函数即可：

函数原型	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。

在完成调用后，直接退出中断向量就可以了。

如果发生的是可恢复错误（比如页面交换、缺页中断或者 MPU 动态页的映射等），那么可以在进行完相应的处理工作之后，向该 CPU 上的错误处理信号端点 RME_Fault_Sig[CPUID]（关于此端点的信息请参见 7.8.2.8）发送信号后直接返回。如果恢复失败，那么也需要调用上述函数杀死该线程。

7.6 内核调试打印函数的移植

内核调试打印函数的底层接口只有一个函数，如下：

函数原型	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 原子操作函数与处理器特殊功能函数的移植

原子操作函数是用来在多核条件下实现无锁内核的。处理器特殊功能函数则能方便处理器的一些特定功能的使用。这些函数可以用 C 语言实现，也可以用汇编语言实现，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。

7.7.1 比较交换原子操作

函数原型	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。该操作会将*Old 和*Ptr 的值进行比对，如果 Old 和 Ptr 不相等，那么返回 0，并把*Ptr 的值赋给*Old；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 指向老值的指针
	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。

该函数完成一个基本的比较交换原子操作。在 x86 等架构上，这个架构有直接的指令（PREFIX LOCK CMPXCHG）支持，此时可以考虑以汇编或内联汇编实现该指令。在 ARM 等 RISC 架构上，也可以考虑使用排他性加载（LDREX）和排他性写回（STREX）指令来支持。具体的支持方法随各个处理器而有不同。

7.7.2 加载自增原子操作

函数原型	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。

该函数完成一个基本的加载自增原子操作。在 x86 等架构上，这个架构有直接的指令（PREFIX LOCK XADDL）支持，此时可以考虑以汇编或内联汇编实现该指令。在 ARM 等 RISC 架构上，也可以考虑使用排他性加载（LDREX）和排他性写回（STREX）指令来支持。具体的支持方法随各个处理器而有不同。

7.7.3 逻辑与原子操作

函数原型	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 的值逻辑与上 Operand，然后写回*Ptr，并且返回与上 Operand 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。

该函数完成一个基本的加载自增原子操作。在 x86 等架构上，这个架构有直接的指令（PREFIX LOCK ANDL）支持，此时可以考虑以汇编或内联汇编实现该指令。在 ARM 等 RISC 架构上，也可以考虑使用排他性加载（LDREX）和排他性写回（STREX）指令来支持。具体的支持方法随各个处理器而有不同。

7.7.4 得到一个字的最高位位置

函数原型	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。

该函数返回该字最高位的位置。最高位的定义是第一个“1”出现的位置，位置是从 LSB 开始计算的（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。

由于该函数需要被高效实现，因此其实现方法在不同的处理器上差别很大。对于那些提供了最高位计算指令的架构，直接以汇编形式实现本函数，使用该指令即可。对于那些提供了前导零计算指令（CLZ）的架构（ARM 等），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。先判断一个字的前半部分是否为 0，如果不为 0，再在这前半段中折半查找，如果为 0，那么在后半段中折半查找，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 初始化、启动与 CPUID 函数的移植

初始化、启动与 CPUID 函数一共有四个，如下所示。

7.8.1 __RME_Low_Level_Init 的实现

函数原型	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（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）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。

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。

7.8.2 __RME_Boot 的实现

函数原型	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。
返回值	ptr_t 总是返回 0。
参数	无。

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。

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。

函数	调用次数	意义
_RME_Kotbl_Init	全系统最多调用一次	在启动时初始化内核内存登记表。
_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。需要注意的是，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 在启动时初始化内核内存登记表

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 Cortex-M 等架构，这就足够了，无需再次调用该函数。但是在某些架构（如

x86-64）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。

函数原型	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：

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 在启动时创建最初权能表

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（因为此时系统中还没有其他权能表）。该函数不需要内核内存权能。

函数原型	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	新创建的信号端点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：

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。

7.8.2.3 在启动时创建权能表

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（因为此时系统中还没有其他权能表）。该函数不需要内核内存权能。

函数原型	ret_t _RME_Captbl_Boot_Crt(struct RME_Cap_Captbl* Captbl, 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	新创建的信号端点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：

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。
	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_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_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。

7.8.2.4 在启动时创建页目录

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。

函数原型	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：

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（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_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。

7.8.2.5 在启动时构造页目录

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（也即页表）。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。

函数原型	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_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。

该函数的返回值可能如下：

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（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。

7.8.2.6 在启动时向页目录中添加页

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。这些页在未来会构成所有的用户地址可访问空间。也即，这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。

函数原型	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。
Paddr	ptr_t	物理内存地址。
Pos	ptr_t	一个该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。
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：

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。

7.8.2.7 在启动时创建第一个进程

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。

函数原型	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（顶层页目录）。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：

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	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。

7.8.2.8 在启动时创建内核功能调用权能

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。

函数原型	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：

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（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_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。

7.8.2.9 在启动时创建内核内存权能

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。

函数原型	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：

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（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_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。

7.8.2.10 在启动时创建内核信号端点

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 block 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。

函数原型	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：

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（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_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。第一种模式比较慢，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三个信号端点：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内核数组 RME_Tick_Sig[CPUID]；第二个信号端点是用来接收线程错误信号

的，指向它的指针应当被赋给内核数组 RME_Fault_Sig[CPUID]；第三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内核数组 RME_Int_Sig[CPUID]。

7.8.2.11 在启动时创建初始线程

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当前处理器，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。Init 线程的优先级上限由系统指定为 RME_MAX_PREEMPT_PRIO-1。这个函数允许通过 CPUID 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。

函数原型	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, ptr_t CPUID)	
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。
CPUID	ptr_t	要将该线程绑定到的 CPU。

该函数的返回值可能如下：

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。

7.8.2.12 设置当前页表、使能中断和进入用户态开始执行

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“页表相关函数的移植”章节；关于__RME_Enable_Int 和__RME_Enter_User_Mode 的相关信息，请参见“汇编底层函数的移植”章节。

7.8.3 __RME_Reboot 的实现

该函数用于重新启动处理器。如果不需要使用到这个功能（比如发生内核故障自动重启，或者在内核功能调用中加入重启功能），那么该函数可以不实现。该函数会将处理器进行软复位，重置 CPU 所有处理器的状态，然后进行重新启动。

函数原型	ptr_t __RME_Reboot(void)
意义	该函数重置并重新启动系统。
返回值	ptr_t 总是返回 0
参数	无

7.8.4 __RME_Shutdown 的实现

该函数用于关闭处理器系统。如果不需要使用到这个功能（系统从不关机），那么该函数可以不实现。该函数会关闭处理器系统并切断电源。

函数原型	ptr_t __RME_Shutdown(void)
意义	该函数关闭系统。
返回值	ptr_t 从不返回。
参数	无。

7.8.5 __RME_CPUID_Get 的实现

该函数在某个 CPU 上调用，用于返回该 CPU 的 CPUID，以使多个 CPU 互相区分。对于单核系统，直接返回 0 即可。这里的 CPUID 指的是一个从 0 开始的数字值，一直到 RME_CPU_NUM-1。如果该处理器返回其他形式的 CPUID 值，该函数要负责把它转换成从 0 开始的值，每一个值对应一个 CPU。

函数原型	ptr_t __RME_CPUID_Get(void)
意义	该函数得到该 CPU 的 CPUID。
返回值	ptr_t 该 CPU 的 CPUID。
参数	无。

7.9 寄存器组相关函数的移植

RME 中，和寄存器组有关的函数有以下 12 个。这 12 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 __RME_Get_Syscall_Param 的实现

该函数用于从寄存器组中提取系统调用的参数。

函数原型	ptr_t __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	ptr_t 总是返回 0。
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，输出半字长的权能表权能号（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 __RME_Get_Inv_Retval 的实现

该函数用于从寄存器组中提取线程迁移调用的返回值。

函数原型	ptr_t __RME_Get_Inv_Retval(struct RME_Reg_Struct* Reg)
意义	提取线程迁移调用的返回值并将其返回。
返回值	ptr_t 提取到的线程迁移调用的返回值。
参数	struct RME_Reg_Struct* Reg 指向寄存器组的指针。

7.9.3 __RME_Set_Syscall_Retval 的实现

该函数用于向寄存器组中存入系统调用的返回值。

函数原型	ptr_t __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	ptr_t 总是返回 0。
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。

7.9.4 __RME_Set_Inv_Retval 的实现

该函数用于向寄存器组中存入线程迁移调用的返回值。

函数原型	ptr_t __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	ptr_t 总是返回 0。
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。

7.9.5 __RME_Thd_Reg_Init 的实现

该函数用于初始化线程的寄存器组。

函数原型	ptr_t __RME_Thd_Reg_Init(ptr_t Entry, ptr_t Stack, struct RME_Reg_Struct* Reg)
意义	使用入口和栈地址初始化线程寄存器组。
返回值	ptr_t 总是返回 0。
参数	ptr_t Entry 线程的入口地址。 ptr_t Stack 线程栈的地址。 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。

7.9.6 __RME_Thd_Reg_Copy 的实现

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。

函数原型	ptr_t __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	ptr_t 总是返回 0。
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。

7.9.7 __RME_Thd_Cop_Init 的实现

该函数用于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。

函数原型	ptr_t __RME_Thd_Cop_Init(ptr_t Entry, ptr_t Stack, struct RME_Cop_Struct* Cop_Reg)
意义	使用入口和栈地址初始化线程协处理器寄存器组。
返回值	ptr_t 总是返回 0。
参数	ptr_t Entry 线程的入口地址。

	ptr_t Stack 线程栈的地址。 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。

7.9.8 __RME_Thd_Cop_Save 的实现

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。

函数原型	ptr_t __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	ptr_t 总是返回 0。
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 该参数用于输出，是指向协处理器寄存器组的指针。

7.9.9 __RME_Thd_Cop_Restore 的实现

该函数用于恢复线程的协处理器寄存器组。

函数原型	ptr_t __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	ptr_t 总是返回 0。
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（通常而言传输能力很强的）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。

7.9.10 __RME_Inv_Reg_Init 的实现

该函数用于初始化线程迁移调用的寄存器组。

函数原型	ptr_t __RME_Inv_Reg_Init(ptr_t Param, struct RME_Reg_Struct* Reg)
意义	用线程迁移初始参数初始化线程迁移调用的寄存器组。
返回值	ptr_t 总是返回 0。

参数	ptr_t Param 线程迁移调用的参数。 struct RME_Reg_Struct* Reg 该参数用于输出，是指向寄存器组的指针。

7.9.11 __RME_Inv_Cop_Init 的实现

该函数用于初始化线程迁移调用的协处理器寄存器组。只有某些系统需要这个参数。

函数原型	ptr_t __RME_Inv_Cop_Init(ptr_t Param, struct RME_Cop_Struct* Cop_Reg)
意义	用线程迁移初始参数初始化线程迁移调用的协处理器寄存器组。
返回值	ptr_t 总是返回 0。
参数	ptr_t Param 线程迁移调用的参数。 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。

7.10 内核功能调用函数的移植

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。这些函数由用户定义。这些函数的描述如下：

函数原型	ptr_t __User_Func(struct RME_Reg_Struct* Reg, 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，如果该函数返回非负值，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 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。在这些函数中有一些特定的内核函数可以调用，用来完成一些基本功能。关于这些特定函数的描述，请参见下一小节。

通常而言这些函数被用于实现一些处理器特定的功能，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 IO 操作。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。接下来介绍几个常见的功能的实现思路。

7.10.1 无节拍内核的实现

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 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 高精度定时器系统的实现

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 处理器间中断的实现

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 缓存操作的实现

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 页表相关函数的移植

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 __RME_Pgtbl_Set 的实现

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。

函数原型	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（如 x86 中的 CR3）指向顶层页目录，这一操作也会同时刷新 TLB 缓存。在这种架构下，只需要简单地将 Pgtbl 参数的值赋给该寄存器即可。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 __RME_Pgtbl_Kmem_Init 的实现

该函数负责在系统启动时建立初始的内核页表。

函数原型	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（或者这些内核页目录）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。

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，因而可以绕过所有的 MPU 保护，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 __RME_Pgtbl_Check 的实现

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。

函数原型	ptr_t __RME_Pgtbl_Check(ptr_t Start_Addr, ptr_t Top_Flag, ptr_t Size_Order, ptr_t Num_Order)
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 MMU 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。

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，对于其他的参数组合都返回 RME_ERR_PGT_OPFAIL（-1）。

7.11.4 __RME_Pgtbl_Init 的实现

该函数负责初始化一个刚刚创建的页目录。

函数原型	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。但是，在 MMU 环境下本函数仍然需要检查创建的页目录是否对齐到了硬件要求对齐的边界，比如某些处理器要求页目录对齐到 4kB（x86）的地址。

7.11.5 __RME_Pgtbl_Del_Check 的实现

该函数负责检查一个页目录能否被安全删除。

函数原型	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 __RME_Pgtbl_Page_Map 的实现

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（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）不能被完成，应当返回错误。

函数原型	ptr_t __RME_Pgtbl_Page_Map(struct RME_Cap_Pgtbl* Pgtbl_Op, 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（槽位号从 0 开始计算）。“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（比较交换，CAS）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 __RME_Pgtbl_Page_Unmap 的实现

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（比如位号超标或者位号的位置没有页存在），那么应当返回错误。

函数原型	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）。
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（如 x86 的 INVLTB 等）。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 __RME_Pgtbl_Pgdir_Map 的实现

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（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、该位置已经有映射或者在 MPU 环境下某些特殊约束不满足），那么应当返回错误。

函数原型	ptr_t __RME_Pgtbl_Pgdir_Map(struct RME_Cap_Pgtbl* Pgtbl_Parent, ptr_t Pos, struct RME_Cap_Pgtbl* Pgtbl_Child)
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。

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（见内存管理章节所述），因此要求父页目录必须具备（或者自身就是）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 __RME_Pgtbl_Pgdir_Unmap 的实现

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（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），那么应当返回错误。

函数原型	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。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 __RME_Pgtbl_Lookup 的实现

该函数负责查找一个页目录内的某个页的信息。

函数原型	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（找到该页）返回 0，失败（未找到或该位置上映射的表项为页目录）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 需要解除映射的子页目录表项位号。
	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（写入指针所指的变量内）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 即可，此时只查询其中一种信息。

7.11.11 __RME_Pgtbl_Walk 的实现

该函数负责查找整个页表（页目录树）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。

函数原型	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（页目录树）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 成功（找到该页）返回 0，失败（未找到该页）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。

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，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 中断处理向量的编写

除了系统调用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 IO，或者修改某个内存地址的变量。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。

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 中断向量的进入和退出

（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：

函数原型	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 中断向量中可以调用的特定内核函数

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 向内核端点发送信号

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。

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

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，因为信号计数的上限为 2^{32}-1。

7.12.2.2 增加 RME_Timestamp 的值

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 Timestamp 的值即可。

函数原型	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。

该函数会返回更新之前的 Timestamp 值。需要注意的是，一旦该函数失败，比如传入了小于 0 的值，那么内核会崩溃。这一点用户在调用该函数时应当加以注意。

7.13 其他函数说明

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 变量清空函数

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。

函数原型	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 比较两段内存

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。

函数原型	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 复制一段内存

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。两端内存区域不能重叠，否则该函数的行为是未定义的。

函数原型	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 打印一个有符号整数

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。

函数原型	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 打印一个无符号整数

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。

函数原型	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 打印一个字符串

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。

函数原型	cnt_t RME_Print_String(s8* String)	
参数名称	类型	描述
String	s8*	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。

本章参考文献

无

第八章 附录

8.1 RME 通常编码规范

为了维持 RME 编码和工程组织的统一性，这里列出 RME 的部分编码规范。

8.1.1 RME 头文件和源文件组织规范

参见 RME 提供的架构的模板文件，和该文件大体维持一致即可。

8.1.2 RME 的 C 语言风格约定

函数命名采用英文缩写，首字母大写并加下划线的方式。在函数的字母部分开始之前要加上“RME”前缀作为命名空间。开头一个下划线代表是用户不应直接调用到的函数，开头两个下划线表示的是属于硬件抽象层的函数，开头三个下划线的则表示是硬件相关的特有函数。

变量和结构体命名也采用英文缩写，首字母大写并加下划线的方式。在变量和结构体中，也要加上“RME”前缀作为命名空间。

RME 的宏定义全部是大写加下划线的格式，其开头也要有“RME”三个字母。类型定义则全部是小写而且不加“RME”三个字母。

8.1.3 RME 的缩进约定

RME 只使用 4 个空格进行缩进。各个语句的写法与文件中已有的写法维持尽量一致即可。下面简要列出各个语句的缩进。如果大括号中只包含一行语句，那么大括号可以省略。

8.1.3.1 if-else

```
if(Cond)
{
    Expr;
}
else
{
    Expr;
}
```

8.1.3.2 switch-case

```
switch(Cond)
{
    case Case1:
    {
        Expr;
    }
    case Case2:
    {
        Expr;
    }
```

```
        default:Expr;
    }
```

8.1.3.3 for

```
    for(Act1;Act2;Act3)
    {
        Expr;
    }
```

8.1.3.4 while

```
    do
    {
        Expr;
    }
    while(Act)
```

或者

```
    while(Act)
    {
        Expr;
    }
```

8.2 RME 对特殊功能的支持

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、多处理器上的多逻辑操作系统等等。下面简述它们的实现思路。

8.2.1 CPU 热插拔

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。以下两个操作在具体平台上的实现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。

8.2.2 内存热插拔

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。以下两个操作在具体平台上的实现都要用定制的内核功能调用完成。

对于增加内存的情况，只要将这些内存对应的物理页框加入到某个页表中即可。如果需要增加重新映射内核内存，那么需要先暂停其他处理器的运行，由一个处理器完成相应页上的内核数据和代码搬移，然后在页表中用现在的新的物理页框去替换原有物理页框即可，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。

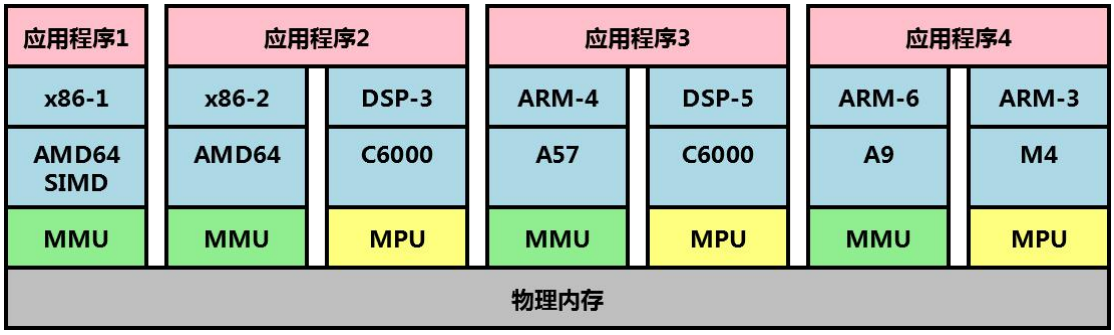
对于减少内存的情况，如果是减少用户态内存，需要用户态库确定现在这些物理页面没有被映射。此时，可以直接拔出该内存条。如果是减少内核内存，那么需要首先暂停其他处理器

的运行，然后将这些将消失的内存中的页面上的数据拷贝到空白的物理页面上，并把原来在内核页表中映射的条目替换成这些空白物理页面，最后再恢复其他处理器的运行。这些空白的物理页面要由用户态库保证不存在除了根页表之外的页表中。在确定这些物理页面由内核使用后，要把这些页面从根页表中也删除，使其不会从用户态被访问。在完成所有操作后也要做一个TLB 刷新操作。

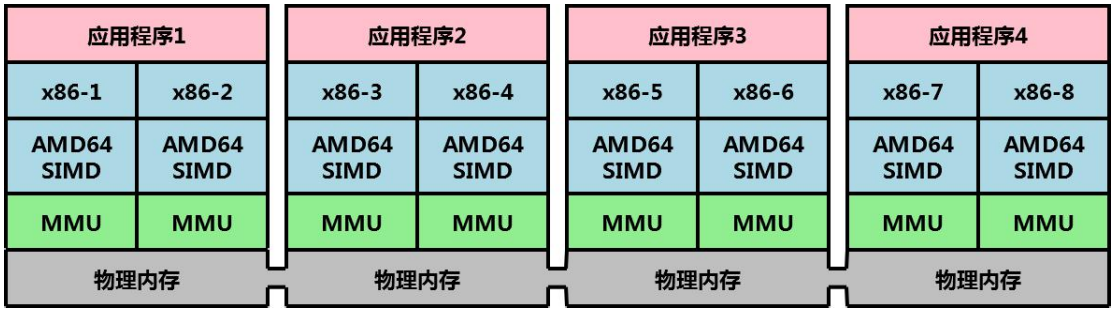
8.2.3 多核处理器多逻辑操作系统

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。因此需要针对不同架构的处理器移植 RME，并且用户态库也要针对不同的处理器进行移植。

比如，如果存在如下图所示的 SoC，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。



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：



8.3 后记

8.3.1 RME 中多核可扩展性的限制因素

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。

8.3.1.1 RME_Timestamp 的更新

RME_Timestamp 的更新是由一个处理器完成的。因此，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。

8.3.2.1 多核同时在一段内核内存区域创建内核对象

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。此外，当创建线程内核对象时，需要增加 TID 分配变量，因此这一步也是不能很好并行化的。

8.3.2.3 多核同时向某信号端点发送信号

由于这是一个原子累加操作，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。

8.3.2 RME 在 32 位系统中的限制因素

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。

8.3.2.1 线程创建

由于 RME 中每个新创建的线程都会被分配一个 TID，而且该 TID 不断自增，永不返回。该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。因此从系统上电开始累计，最多只能创建 2^{30}-1 个线程（在 64 位系统中该值为 2^{62}-1，显然不是问题）。需要注意的是，即便先创建后删除，也会使 TID 自增 1。

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，因此很少频繁创建和销毁线程。此外，还可以通过线程池的方法管理暂时不使用的线程，从而在系统启动后不进行线程的创建，因此相当于绕过了这个限制。

8.3.2.2 安定时间的计算

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（比如权能冻结），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，如果发现经过了操作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。

8.3.3 RME 中已知的潜在隐蔽通道

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。

8.3.3.1 隐蔽存储通道

RME 中存在的隐蔽存储通道主要发生在共享权能、权能表、内核内存权能的两个进程之间。对于共享权能的情况，两个进程可以通过对该权能的操作试探和设置该权能的状态，从而完成信息传递。对于共享权能表的情况，在权能表中创建权能会导致某个槽位被占用，而这种占用如果能被另一个进程通过冻结操作或其他操作试探，那么就可以传递信息。对于内核内存分配表也是一样的，如果两个进程在同一段内核内存上创建内核对象，那么就可以通过内核内存是否被占用来试探对方对内核内存的使用情况，因此也可以在两个进程间传递信息。

RME 的 TID 是全局分配的。因此，TID 也可能成为隐蔽存储通道。不过，在 RME 中，TID 仅仅被用做接收调度器事件的返回值，因此 TID 一般只被调度器等守护进程使用，不会在应用程序中造成隐蔽通道。

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，需要慎重处置。

8.3.3.2 隐蔽定时通道

由于在 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（比如 x86 的 RDTSC）。关于此部分，在此不详细说明。

8.4 术语中英翻译速查表

英文术语	中文翻译
Capability	权能
Component	组件
Coroutine	协程
Daemon	守护进程
Endpoint (Signal)	（信号）端点
Expandable/Expandability	可扩展/可扩展性
Invocation (Thread Migration)	（线程迁移）调用
Page Directory	页目录
Page Entry	页表项
Page Table	页表
Priority	优先级
Process	进程
Quiescence	安定
Scalable/Scalability	可伸缩/可伸缩性
Scheduler	调度器
Signal	信号
Thread	线程

本章参考文献

[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.