突越・中生・真核（七阶）

Mutate・Mesozoa・Eukaron(R.VII)

M7M1(Eukaron) 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.

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

There are also other alternatives to capabilities, e.g. Access Control List (ACL)[6]. It was introduced by Lampson in 1974, and can also be applied to access permission management. The drawback of these mechanisms is the coarser grain and big list size. Its advantage is the easiness of permission granting and revoking (especially the latter).

In RME, by leveraging the concept of capability, it is easy to implement discretionary access control and mandatory access control, and multi-level safety mechanisms. In user-level application designs, the concept of minimal privilege should be applied.

2.2 Capability Table Operations and States of Capabilities

Capability table is a kernel object which stores capabilities. In RME it is a linear array, and the size of every slot is 8 machine words. Each slot can hold a capability. The information regarding the type, kernel virtual memory address (a pointer), parent capability, reference count and state is stored in each capability slot. There's an additional timestamp counter to ensure the quiescence of capability operations under highly parallel environments.

2.2.1 Capability Types

In RME, there are 8 capability types (without counting in the empty capability), as listed below.For detailed information about each capability type, please consult the corresponding chapter for details. This chapter will only detail capability table capabilities.

Capability type number	Type	Explanation
RME_CAP_NOP	Empty capability	This position is empty.
RME_CAP_KERN	Kernel capability	Necessary for kernel function calls.
RME_CAP_KMEM	Kernel memory capability	Necessary for kernel object creation.
RME_CAP_CAPTBL	Capability table capability	Necessary for capability management.

RME_CAP_PGTBL	Page table capability	Necessary for memory management.
RME_CAP_PROC	Process capability	Necessary for process management.
RME_CAP_THD	Thread capability	Necessary for thread management.
RME_CAP_INV	Invocation capability	Necessary for making invocations.
RME_CAP_SIG	Signal endpoint capability	Necessary for signal sending and receiving.

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

2.2.2 Reference Count of Capabilities and Parent Capabilities

The reference counter and parent capability pointer of a capability is used to track capability delegation. A capability can be passed from one capability table to another through delegation, where case we call the source capability as parent and the destination capability as child. The child's parent pointer always points to its parent, and the parent's reference count will be incremented by 1. When removing capabilities, the child capability must be removed before the parent capability gets removed. When the initial capability to a kernel object is created, its parent pointer will be NULL, and its reference count will be set to 0.[4]

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

2.2.3 States of Capabilities

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

To create a capability, the corresponding creation system call must be performed. A capability table capability is needed to designate the target capability table (because this operation adds a capability into the table, which is a modification to the table, thus requiring a capability table capability; to remove capabilities you also need that capability.), and other necessary information including the address of the kernel object must be passed in. The creation operation will be compare-and-swaped (CAS) to "creating"; when the creation is finished, the state will be marked as "valid". If an error happens during creation, the slot will be reset to "empty", and an error code is returned. When creating capabilities, if kernel memory allocations are needed, a kernel memory capability is necessary; see next chapter for details.

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

There are two design styles regarding capability revoking. One being the implementation of seL4 and Fiasco.OC, which supports system-level capability revoking. Upon kernel object deletion, the

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

2.2.4 Timestamp Counter and Multi-core Scalability

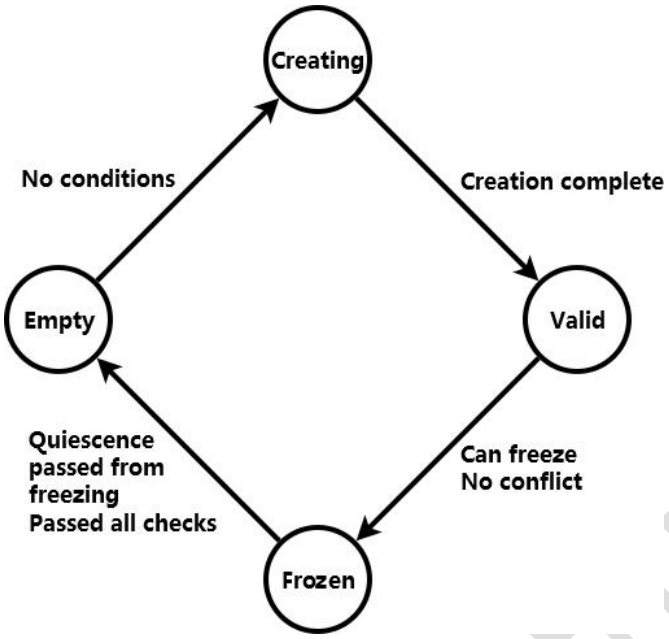
Multi-core execution pattern is highly complex. There are some kernel data structures that shouldn't be modified simultaneously in a multi-core environment, and these data structures require that all operations on them are atomic. Therefore, two solutions exist: the first being taking a lock for each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

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

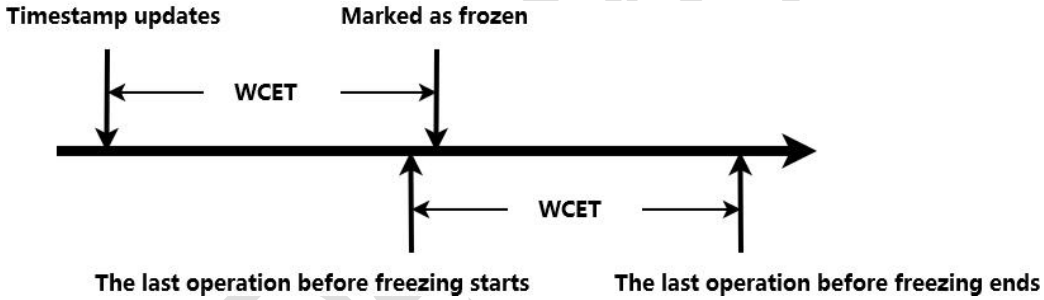
To delete or remove a capability, we must ensure that the capability have been frozen for a time period. This period is called capability quiescence period. The length of this period can be configured, and must be longer than twice the kernel WCET. Thus, we can ensure that when we delete or remove the capability, the capability have been stabilized. In other words, all kernel operations that use this capability have already finished, and there wouldn’t be a conflict between usage and deletion/removal. For example, when CPU1 is trying to delete the capability A, CPU2 tries to use it at the same time. CPU2 may have already completed the operation validation, and is progressing. If is capability gets deleted on CPU1, the operation of CPU2 may be affected. The timestamp counter of each capability is designed to log the freeze time; it will guarantee that no deletion or removal can occur before the freeze quiescence period passes.

A complete capability state transition diagram is shown below. When the creation starts, the CPU will perform a CAS operation in the empty slot, which marks the start of the creation. After the creation is finished, the CPU will mark the slot as usable, and now operations can be started on this capability. If this capability is not referenced, then it can be deleted after usage. To do this, we need to freeze it first, and wait until the quiescence period passes. After that, we can delete or remove it.

When deleting or removing capabilities, we need to check necessary conditions to confirm that the deletion or removal can be performed. After this, A CAS operation will return this capability slot to an empty state, and we guarantee that only one CPU can proceed. Then, the parent’s reference count will be decreased by one (removal) or the kernel object will be deleted (deletion). In RME, new capabilities can be created on the slot immediately after deletion or removal. This is due to the fact that we cache all the information before we change slot state to empty, and all the operations that follow will use the cached information. In Composite[4], it may still use these information in the slot, thus another quiescence period needs to be inserted here to avoid overwriting by newly created capabilities.



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



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

2.2.5 Capability Table Structure

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

32-bit	Master only	[15:8] Reserved [7] Fixed-0 [6:0] Position
	Expanded	[15] Reserved [14:8] 2nd level position [7] Fixed-1 [6:0] 1st level position
64-bit	Master only	[32:16] Reserved [15] Fixed-0 [14:0] Position
	Expanded	[32] Reserved [31:16] 2nd level position [15] Fixed-1 [14:0] 1st level position

From the table we can see that there can be at most 2^7=128 capabilities in a capability table under 32-bit, and 2^{15}=32768 capabilities under 64-bit. This value is represented by macro RME_CAPID_2L in the system. Different from systems like seL4, RME does not support look-ups that have more than two level. The capabilities with in the capability table that is pointed to by the capability table capability in the expanded capability table are not considered to be within the expanded capability table, and cannot be used directly with expanded capability number encoding ("my vassal's vassal is not my vassal"). If you need to use them, you must add them to the expanded capability table before usage.

2.3 Capability Table System Calls

The capability table-related system calls are listed as follows:

System call number	Type	Explanation
RME_SVC_CAPTBL_CRT	System call	Create a capability table
RME_SVC_CAPTBL_DEL	System call	Delete a capability table
RME_SVC_CAPTBL_ADD	System call	Add a capability into the capability table (delegation)
RME_SVC_CAPTBL_FRZ	System call	Freeze a capability
RME_SVC_CAPTBL_REM	System call	Remove a capability

The operation flags of capability table capabilities are listed as follows:

Flag	Bit	Explanation
RME_CAPTBL_FLAG_CRT	[0]	Allows creation of capabilities in this table.
RME_CAPTBL_FLAG_DEL	[1]	Allows deletion of capabilities in this table.
RME_CAPTBL_FLAG_FRZ	[2]	Allows freezing of capabilities in this table.
RME_CAPTBL_FLAG_ADD_SRC	[3]	Allows using this table as delegation source.
RME_CAPTBL_FLAG_ADD_DST	[4]	Allows using this table as delegation destination.
RME_CAPTBL_FLAG_REM	[5]	Allows removal of capabilities in this table.
RME_CAPTBL_FLAG_PROC_CRT	[6]	Allows creation of process with this table.
RME_CAPTBL_FLAG_PROC_CPT	[7]	Allows replacement of process capability table.

For detailed information about bit[6] and bit[7], please refer to process management related sections.

2.3.1 Capability Table Creation

This operation will create a new capability table and place its capability into an existing capability table. The following parameters are required for this operation:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_CAPTBL_CRT.
Cap_Captbl_Crt	cid_t	C	A capability number that corresponds to a capability table capability that possesses the RME_CAPTBL_FLAG_CRT flag. The capability table that this capability points to will accept the newly created capability table capability. This capability number can have either a master encoding or a expanded encoding.
Cap_Kmem	cid_t	P1.D1	A kernel memory capability, whose kernel memory range must be big enough to contain the whole capability table. This capability must have the RME_KMEM_FLAG_CAPTBL flag. This

			capability number have be either a master encoding or a expanded encoding.
Cap_Crt	cid_t	P1.D0	A capability number that corresponds to a capability slot that will accept the newly created capability table capability. This slot must be empty. This capability number can only have a master encoding.
Vaddr	ptr_t	P2	The kernel virtual address that this new capability table will use.
Entry_Num	ptr_t	P3	The number of capabilities contained within this capability table, must range from 1 to RME_CAPID_2L.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The number of entries exceeded what is allowed by the RME.
	The lookup of Cap_Captbl_Crt exceeded range.
	The lookup of Cap_Kmem exceeded range.
	The lookup of Cap_Crt exceeded range.
RME_ERR_CAP_FROZEN	The lookup Cap_Captbl_Crt encountered a frozen capability.
	The lookup Cap_Kmem encountered a frozen capability.
	Cap_Crt is frozen, or other CPUs are creating capabilities there.
RME_ERR_CAP_TYPE	Cap_Captbl_Crt is not a capability table capability.
	Cap_Kmem is not a kernel memory capability.
RME_ERR_CAP_FLAG	Cap_Captbl_Crt doesn't have RME_CAPTBL_FLAG_CRT flag.
	Cap_Kmem doesn't have RME_KMEM_FLAG_CAPTBL flag, or does not cover the range designated by Vaddr.
RME_ERR_CAP_EXIST	Cap_Crt is not an empty capability slot.
RME_ERR_CAP_KOTBL	Kernel memory allocation failed.

2.3.2 Capability Table Deletion

This operation will delete a capability table. The capability table being deleted must contain no capabilities; in other words, the capability table should be empty. To delete a capability table, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_CAPTBL_DEL.
Cap_Captbl_Del	cid_t	C	A capability number that corresponds to a capability table capability that possesses the RME_CAPTBL_FLAG_DEL flag. The capability table that this capability points to contains the capability to the capability table being deleted. This capability number can have either a master encoding or a expanded encoding.
Cap_Del	cid_t	P1	A capability number that corresponds to the capability to the capability table being deleted. The capability must be a capability table capability. This capability number can only have a master encoding.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Captbl_Del exceeded range.
	The lookup of Cap_Del exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Captbl_Del encountered a frozen capability.
	Cap_Del is not frozen.
RME_ERR_CAP_TYPE	Cap_Captbl_Del is not a capability table capability.
	Cap_Del is not a capability table capability.
RME_ERR_CAP_NULL	Cap_Del is an empty capability.
	Two CPUs try to delete the same capability table, and this value will be returned on the failed CPU.
RME_ERR_CAP_FLAG	Cap_Captbl_Del doesn't have RME_CAPTBL_FLAG_DEL flag.
RME_ERR_CAP_QUIE	Cap_Del is not quiescent.
RME_ERR_CAP_EXIST	There are still capabilities in the capability table that Cap_Del corresponds to.
RME_ERR_CAP_REFCNT	The reference count of Cap_Del is non-zero, or it is not a root capability.

2.3.3 Capability Delegation

This operation will delegate a capability from one capability table to an empty slot in another capbility table. The parent capability of the destination capability is the source capability, and the reference count of the source capability will be increased by 1. To delegate a capability, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_CAPTBL_ADD.
Cap_Captbl_Dst	cid_t	P1.D1	A capability number that corresponds to a capability table capability that possesses the RME_CAPTBL_FLAG_ADD_DST flag. This capability points to the destination capability table. This capability number can have either a master encoding or a expanded encoding.
Cap_Dst	cid_t	P1.D0	A capability number that corresponds to a capability slot that will accept the delegated capability. This slot must be empty. This capability number can only have a master encoding.
Cap_Captbl_Src	cid_t	P2.D1	A capability number that corresponds to a capability table capability that possesses the RME_CAPTBL_FLAG_ADD_SRC flag. This capability points to the source capability table. This capability number can have either a master encoding or a expanded encoding.
Cap_Src	cid_t	P2.D0	A capability number that corresponds to the capability to the capability table being delegated. The capability must be non-empty and not frozen. This capability number can only have a master encoding.
Flags	ptr_t	P3	The operation flags to inherit. Only the operations allowed by

			these flags can be executed on the newly delegated capability.

It is worth noting that, for kernel memory capabilities, additiona bits from system call number field N and capability number C is needed to help identifying its operation flags. Please refer to the next chapter for details.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Captbl_Dst or Cap_Captbl_Src exceeded range.
	The lookup of Cap_Dst or Cap_Src exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Captbl_Dst or Cap_Captbl_Src encountered a frozen capability.
	Cap_Src is frozen.
RME_ERR_CAP_TYPE	Cap_Captbl_Dst or Cap_Captbl_Src is not a capability table capability.
RME_ERR_CAP_NULL	Cap_Src is an empty capability.
RME_ERR_CAP_FLAG	Cap_Captbl_Src doesn't have RME_CAPTBL_FLAG_ADD_SRC flag.
	Cap_Captbl_Dst doesn't have RME_CAPTBL_FLAG_ADD_DST flag.
	Cap_Src's flags conflict with the flags passed in. In other words, the flags passed in included flags or ranges that Cap_Src does not possess.
	The flags passed in is illegal. They allow no operations, or contain range conflicts.
RME_ERR_CAP_EXIST	Cap_Dst is not an empty capability slot.
RME_ERR_CAP_REFCNT	The reference count of Cap_Src exceeded the maximum reference count allowed. In 32-bit systems this limit is 2^{23}-1, in 64-bit systems this limit is 2^{46}-1. Usually this limit is sufficient.

2.3.4 Capability Freezing

This operation will freeze a capability in a capability table. If a capability is frozen, we can guarantee that all kernel operations that initiated from this capability have finished their execution, and we can delete or remove this capability now. It is worth noting that this does not imply all operations on the kernel object have stopped, because there might be other capabilities that points to the same kernel object as well. If and only if the root capability have been frozen will all operations on the kernel object be stopped, and then the kernel object and the root capability can be deleted. To freeeze a capability, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_CAPTBL_FRZ.
Cap_Captbl_Frz	cid_t	C	A capability number that corresponds to a capability table capability that possesses the RME_CAPTBL_FLAG_FRZ flag, The capability table that this capability points to contains the capability to be frozen. This capability number can have either a master encoding or a expanded encoding.

Cap_Frz	cid_t	P1	A capability number that corresponds to the capability to the capability table being frozen. This capability number can only have a master encoding.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Captbl_Frz exceeded range.
	The lookup of Cap_Frz exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Captbl_Frz encountered a frozen capability.
	Cap_Frz is already frozen and doesn't need another freeze; or some other CPU is creating a new capability here.
RME_ERR_CAP_TYPE	Cap_Captbl_Frz is not a capability table capability.
RME_ERR_CAP_NULL	Cap_Frz is an empty capability.
RME_ERR_CAP_FLAG	Cap_Captbl_Frz doesn't have RME_CAPTBL_FLAG_FRZ flag.
RME_ERR_CAP_EXIST	Two CPUs try to freeze the same capability, and this value will be returned on the failed CPU.
RME_ERR_CAP_REFCNT	The reference count of Cap_Frz is non-zero.

2.3.5 Capability Removal

This operation will remove a capability in a capability table. The capability to be removed cannot be the root capability (in which case delete operation should be used), and cannot be referenced by other capabilities. Removing a capability will not remove the corresponding kernel object, and only the capability will be removed (the delete operation works on unreferenced root capability, and will delete the root capability together with the kernel object). To remove a capability, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_CAPTBL_REM.
Cap_Captbl_Rem	cid_t	C	A capability number that corresponds to a capability table capability that possesses the RME_CAPTBL_FLAG_REM flag, The capability table that this capability points to contains the capability to be removed. This capability number can have either a master encoding or a expanded encoding.
Cap_Rem	cid_t	P1	A capability number that corresponds to the capability to the capability table being removed. This capability number can only have a master encoding.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Captbl_Rem exceeded range.
	The lookup of Cap_Rem exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Captbl_Rem encountered a frozen capability.
	Cap_Rem is not frozen.
RME_ERR_CAP_TYPE	Cap_Captbl_Rem is not a capability table capability.

RME_ERR_CAP_NULL	Cap_Rem is an empty capability.
	Two CPUs try to remove the same capability, and this value will be returned on the failed CPU.
RME_ERR_CAP_FLAG	Cap_Captbl_Rem doesn't have RME_CAPTBL_FLAG_REM flag.
RME_ERR_CAP_QUIE	Cap_Rem is not quiescent.
RME_ERR_CAP_REFCNT	The reference count of Cap_Rem is non-zero.

Bibliography

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

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

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

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

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

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

Chapter 3 Page Table and Memory Management

3.1 Introduction

Memory management refers to the allocation and reclamation operations performed on physical memory or virtual address space. Hardware support is needed to guarantee the safety of memory management. RME supports both Memory Management Unit (MMU)-based architectures and Memory Protection Unit (MPU)-based architectures, and abstracted them into similar page table data structures. The terms and definitions used in this chapter is listed as below:

Term	Definition
Page Table (PT)	The whole page table including the top-level page directory and all intermediate levels of page directories. In other words, it refers to the whole address space trie.
Page Directory (PD)	A single level in the page table. The top most level is called the top-level page directory.
Page Entry (PE)	A single entry in the page table. It may point to a physical page, or point to the next page table level. We call the entries that point to a page as Page Terminal Entry (PTE), the entries that point to the next level of page table as Page Intermediate Entry (PTE).

3.2 Page Table Operations and Structure

Page table is a multi-level radix trie consisting of a series of page directories. Each entry of the slot is fixed to a machine word. There can be 3 kinds of data in each entry: physical address of the next-level page table, page physical address or an empty entry. If a next-level page table physical address is there, then it means that there is a next-level page table, and the virtual-to-physical mapping in this range must be decided by consulting that level; if a page physical address is there, then it means that a page is mapped, and we can get the access permissions of that page. If an empty entry is there, then it means that nothing is mapped. In RME, the user needs to explicitly construct the page table, which is similar to Composite[1].

3.2.1 Kernel Memory and User Memory

The system memory of RME is split in two: kernel memory and user memory. Different from Composite and L4, RME's kernel memory mapping is established at system boot-time (static or semi-static allocations), and is unmodifiable. This means that RME does not need memory retyping system (which is used by Composite and seL4[1][3]), and memory management is completely scalable without the possibility of leaking kernel memory. For MMU environments, when creating the top-level page directory, RME will map all the kernel pages into the top-level page directory; for MPU environments, the kernel usually always have access to all the memory in the system, thus we only need user pages in our page directories.

At system boot-time, all user physical memory is added to the Init process's page table. After such addition, (usually) no further addition of physical page frames are allowed. When creating new processes, the page table entries of its page table must be added from other processes. When adding pages to a page table, the access permissions can be restricted to a subset of that of its parent page. In RME, the standard access permission flags are as follows:

Name	Identifier	Explanation
Readable	RME_PGTBL_READ	This page is readable.
Writable	RME_PGTBL_WRITE	This page is writable.
Executable	RME_PGTBL_EXECUTE	This page is executable.
Cacheable	RME_PGTBL_CACHEABLE	This page is cacheable.
Bufferable	RME_PGTBL_BUFFERABLE	This page can use write-back policy.
Static	RME_PGTBL_STATIC	This page will always be mapped. This means that,under MPU environments or software-loaded TLB MMU environments, this page is always mapped and will not have any page fault on it.

It is worth noting that on some architectures some of the flags listed above may not make sense. On MMU architectures that have a hardware-loaded Trans Look-aside Buffer (TLB), the static flag does not take effect; for some architectures, read and write is combined into one bit and thus do not have independent read/write control.

To enable user-level kernel memory management during kernel object creation, the kernel objects are managed using a kernel object allocation table. This table is a bitmap which indicates what kernel memory is used, thus there will not be two kernel objects overlapping on the same kernel virtual address.

It is possible for some user components to create a very large number of kernel objects when they go awry or get compromised, thus making a Denial of Service (DoS) attack, kernel memory capabilities are introduced to manage kernel memory. The concept of kernel memory capability is derived from the idea of Fiasco.OC's[4] Factory. When creating any kernel objects, a kernel memory capability is required; it carries information about the kernel virtual memory range allowed for creating kernel objects, and what kernel objects are allowed in this address range. If and only if the object created falls within the address range do we allow creation operations, or an error will be returned. Thus, we can avoid kernel memory exhaustion caused by a single malfunctioning system component.

3.2.2 Page Table Properties

In RME, there are four properties of page directories. The four properties alone will decide the state of the page directory. We will introduce these properties in the sections that follow.

3.2.2.1 Mapping Start Address

Mapping start address refers to the starting virtual address of this page directory. The virtual address of the first slot is this virtual address. When we try to construct a lower-level page directory into a higher-level page directory, we may need to check whether the start mapping address of the lower-level matches that of the higher-level. This check is only necessary for MPU-based systems; for MMU-based systems, because a page directory will be mapped into different positions of different page directories, this check can be bypassed to allow memory remapping, in which case the mapping start address have no effect. If this check is disabled, no path-compressed page table feature is available (MMU-based systems does not support this in most cases, either.). For detailed description about path-compressed page tables and the difference between MMU/MPU, please refer to the sections that follow.

3.2.2.2 Top-level Page Directory Flag

This page directory is a top-level one. Only the top-level page directory can be used to create a process.

3.2.2.3 Page Directory Size Order

The size order decided the virtual address space size of each page in the page directory. For example, if the size order of a page directory is 12, then all the pages in the page table will correspond to a physical page whose size is 2^{12}=4096 bytes.

3.2.2.4 Page Directory Number Order

The number order decided the number of pages in the page directory. For example, if the number order of a page directory is 10, then the page table contains 2^{10}=1024 pages.

3.2.3 Basic Page Directory Operations

There are 6 different basic operations on a page directory, as listed below:

Operation	Explanation
Create	Create a new empty page directory.
Delete	Delete a page directory.
Add	Add a physical memory page to a virtual address of a page directory.
Remove	Delete a physical memory page from a virtual address of a page directory.
Construct	Add a child page directory to a virtual memory address of a parent page directory.
Destruct	Delete a child page directory from a virtual memory address of a parent page directory.

The detailed implementation of the six operations on MMU and MPU are very different, and different constraints apply. Refer to the two sections below to get a better understanding.

3.2.4 Implementation of Multi-Level Page Table in MMU-Based Architectures

The page directory implementation for MMU-based architectures are very straightforward: they are implemented as a simple linear table. For example, on x86-64, the first level have fixed 512 pages, and each page is 2^{39} bytes; the second level also have fixed 512 pages, and each page is 2^{30} bytes; the third level also have fixed 512 pages, and each page is 2^{21} bytes; the fourth level also have 512 pages, and each page is 2^{12} bytes. The radix trie formed by the four levels is the page table. All start addresses of these page tables should be aligned to a 4kB boundary to fall within a single page.

If it is allowed to construct a page directory into any higher-level page directory (if the size order and number order fits), we can configure the kernel to skip start address checking. By doing this we are using a normal multi-level page table, and no path compression is used.

Due to the fact that the processors can walk the page table directly, we do not need to generate page table metadata for such processors. But, the page tables created must fit the hardware walking mechanisms of the processor. For purely software-loaded TLB, such requirements are not imposed and any trie structures that satisfy the page table structure can be created.

Some popular MMU-based architectures are listed below:

Processor	Levels	Page size	Note
ARM926EJ-S	2 or 3	1MB, 64kB, 4kB, 1kB	Partial TLB lockdown
x86-64 (AMD64)	3	1GB, 2MB, 4kB	Segmentation available

Itanium (IA-64)	4	256MB, 16MB, 4MB, 1MB, 256kB, 64kB, 8kB, 4kB	Partially software-loaded TLB
e200 (PowerPC)	N/A	1kB-4GB, all powers of 2	Software-loaded TLB
ARMv7-A (32-bit)	2 or 3	4kB, 64kB, 1MB, 16MB	Partial TLB lockdown
ARMv8-A (64-bit)	3 or 4	4kB, 16kB, 64kB	Virtualization support
MIPS64	N/A	1kB-256MB, all powers of 4	Software-loaded TLB

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

On MPU-based architectures, the processor is usually unable to directly recognize a multi-level page table. Therefore we must generate MPU metadata to efficiently set the MPU when we switch processes. There are also some idiosyncrasies of MPU-based architectures:

1. The number of MPU regions is usually very limited. Cortex-M3 have 8 regions, and each of them can be divided into 8 subregions. Thus, we can allow up to 64 memory segments in one process, and these mappings are subject to many restrictions. To solve this problem, we can divide the pages into two categories: static pages and dynamic pages. The static pages are always mapped, and the applications that require high predictability can use them; the maximum number of static pages is the maximum number of MPU regions allowed by the processor. The dynamic pages, on the contrary, are only mapped when they are accessed, and there's no guarantee that they will always be mapped. There is no number limit on dynamic pages. If we access a dynamic page that is not currently mapped, the processor will enter a memory protection fault vector, and we will look up the page table manually to add the dynamic page into MPU metadata. If no empty regions are available, some other dynamic page may get replaced. The dynamic page mechanism is very similar to how Emcraft's uCLinux[2] implement paging.

2. Only a small number of pages may present in a page table. For MPU-based microcontrollers, it will be a waste of memory to keep multiple levels of page tables. We must compress the page table to save memory. Compared with normal page tables, the number of bits translated by a level of path-compressed page table at each level can be different. Let's assume that there is a very small page at a certain address that we wish to add to our page table; for normal page tables, we need multiple intermediate directories, and we will add the page in at the last level. For path-compressed page tables, we only need a single level of page table to access the page. We can store information about the start address, number order and size order of the page directory, and construct (map) it directly into the upper-level. We will certainly guarantee that the virtual memory range of this page directory falls within the virtual address slot of the upper page directory's corresponding entry.

3. MPU cannot convert physical addresses to virtual address. Our virtual address is always equal to our physical address, which means that we must check whether virtual address equals physical address for each page mapping.

4. Because we need MPU metadata to speedup MPU programming, whenever we modify a page directory, we need to guarantee the consistency between the MPU metadata and the page table. If we do not keep metadata,we can only simulate TLB by clearing MPU registers on each process switch (see section 3.2.5.2 for details).

Some popular MPU-based architectures are listed below:

Processor	Regions	Organization	Size	Alignment	Note
ARM V7-M	0 to 16	Unified	128B-4GB	Size	8 subregions

ARM V8-M	0 to 16	Unified	128B-4GB	None	None
Tensilica L106	16 to 32	Unified	4kB-1GB	None	None
MIPS M14k	1 to 16	Unified	Arbitrary	None	Can be read-only
e200z4	32	Code/Data 16 each	Arbitrary	None	None
AVR32	8	Unified	4kB-4GB	Size	16 subregions
MSP430FRXX	3	Unified	Arbitrary	None	Three segments
Coldfire-MCF	4	Code/Data 2 each	16MB-4GB	Size	None

To sum up, there are two typical page table implementations for MPU-based architectures:

3.2.5.1 MPU Metadata at Top-level Page Directory

This implementation places the MPU metadata into the top-level page directory, and page table construction should always start from the top-level (if any level is not or doesn't have a top-level page directory, we cannot construct child page directories into this page directory). Additionally, no page tables can share a common page directories or a subtree. The restrictions of such an implementation is as follows:

Operation	Restrictions and drawbacks
Process creation	No restrictions.
Swap process page table	No restrictions.
Swap processes	No restrictions, the top-level metadata can be directly loaded into MPU.
Page directory creation	No restrictions.
Page directory deletion	The page directory cannot have any child page directories and cannot be a child directory of other page directories.
Page mapping	If this page directory have a top-level, update its MPU metadata.
Page deleting	If this page directory have a top-level, update its MPU metadata.
Page directory mapping	The parent directory must be or have a top-level directory; the child directory must have no top-level directory (and it can't have any child directories for sure). Add the mapped pages of the child directory to the top-level's MPU metadata.
Page directory deleting	The child directory must have a top-level directory, and itself must not contain any child directories (its own child directories must be deleted from itself first). Delete the mapped pages of the child directory from the top-level's MPU metadata.
Memory overhead	MPU metadata maintained at the top-level only.

This is the recommended implementation. It can maximally boost the efficiency on MPU-based architecures, and we can usually use MPU metadata generated beforehand to batch load the MPU registers. The only drawback is the loss of some rarely-used functionality. The update of MPU metadata is also very quick when the page table have structural changes.

3.2.5.2 Software Page Table and TLB Simulation

This implementation takes the MPU region registers as a software-loaded TLB. Software handlers are employed to fill the MPU registers if any miss happens. It does not generate the MPU metadata beforehand, rather to generate it step by step when page miss occurs. When page table

structure or page mapping changes, it will clear all MPU registers, just like a full TLB flush under the MMU-based architectures.

Operation	Restrictions and drawbacks
Process creation	No restrictions.
Swap process page table	Clear MPU metadata in MPU registers and wait for reestablishment.
Swap processes	Clear MPU metadata in MPU registers and wait for reestablishment.
Page directory creation	No restrictions.
Page directory deletion	No restrictions.
Page mapping	No restrictions.
Page deleting	Clear MPU metadata in MPU registers and wait for reestablishment.
Page directory mapping	No restrictions.
Page directory deleting	Clear MPU metadata in MPU registers and wait for reestablishment.
Memory overhead	No extra overhead, MPU registers are used to simulate the TLB.

This implementation is maximally compatible with MMU-based architectures, and uCLinux [2] implemented its MPU support in this way. It assumes no restrictions on how the page tables are constructed, and multiple page tables can share a common portion. However, in each process switch (and especially inter-process communications) the whole MPU register set must be cleared, thus it have poor performance and its execution is unpredictable.

3.3 Page Table System Calls

The system calls that are related to the page table are listed as follows:

System call number	Type	Explanation
RME_SVC_PGTBL_CRT	System call	Create a page directory
RME_SVC_PGTBL_DEL	System call	Delete a page directory
RME_SVC_PGTBL_ADD	System call	Map a memory page
RME_SVC_PGTBL_REM	System call	Remove a memory page
RME_SVC_PGTBL_CON	System call	Construct a page directory into another
RME_SVC_PGTBL_DES	System call	Destruct a page directory from another

The operation flags of page directory capabilities are listed as follows:

Flag	Bit	Explanation
RME_PGTBL_FLAG_ADD_SRC	[0]	Allows usage as source in page mapping.
RME_PGTBL_FLAG_ADD_DST	[1]	Allows usage as destination in page mapping.
RME_PGTBL_FLAG_REM	[2]	Allows removal of pages.
RME_PGTBL_FLAG_CON_CHILD	[3]	Allows usage as child in page table construction.
RME_PGTBL_FLAG_CON_PARENT	[4]	Allows usage as parent in page table construction.
RME_PGTBL_FLAG_DES	[5]	Allows deconstructing its child page directories.
RME_PGTBL_FLAG_PROC_CRT	[6]	Allows process creation with this page directory.
RME_PGTBL_FLAG_PROC_PGT	[7]	Allows replacement with this page directory.
Other bits	Field	Operation range.

For detailed information about bit[6] and bit[7], please refer to process management related sections. When delegating page directory capabilities, the flags should be passed in as the macro "RME_PGTBL_FLAG(HIGH, LOW, FLAGS)". HIGH is the upper operation position limit allowed by the new capability, LOW is the lower operation position allowed by the new capability, and the

closed interval [HIGH, LOW] is the range of operation positions allowed. The FLAGS[7:0] is the operations that are allowed.

3.3.1 Page Directory Creation

This operation will create a page directory, and place its capability into a existing capability table. To create a page directory, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N.D0	Must be RME_PGTBL_CRT.
Cap_Captbl	cid_t	C	A capability number that corresponds to a capability table capability that possesses the RME_CAPTBL_FLAG_CRT flag. The capability table that this capability points to will accept the newly created page directory capability. This capability number can have either a master encoding or a expanded encoding.
Cap_Kmem	cid_t	P1.D1	A kernel memory capability, whose kernel memory range must be big enough to contain the whole page directory. This capability must have the RME_KMEM_FLAG_PGTBL flag. This capability number can have either a master encoding or a expanded encoding.
Cap_Pgtbl	cid_t	P1.Q1	A capability number that corresponds to a capability slot that will accept the newly created page directory capability. This slot must be empty. This capability number can only have a master encoding.
Vaddr	ptr_t	P2	The kernel virtual address that this new page directory will use.
Start_Addr	ptr_t	P3	The mapping start virtual address of the new page directory. The last bit is top-level flag as explained below.
Top_Flag	ptr_t	P3[0]	Pass in “1” for top-level, “0” for other levels.
Size_Order	ptr_t	P1.Q0	The size order of the page directory (the size of each page entry).
Num_Order	ptr_t	N.D1	The number order of the page directory.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Captbl exceeded range.
	The lookup of Cap_Kmem exceeded range.
	The lookup of Cap_Pgtbl exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Captbl encountered a frozen capability.
	The lookup of Cap_Kmem encountered a frozen capability.
	Cap_Pgtbl is frozen, or other CPUs are creating capabilities there.
RME_ERR_CAP_TYPE	Cap_Captbl is not a capability table capability.
	Cap_Kmem is not a kernel memory capability.
RME_ERR_CAP_FLAG	Cap_Captbl doesn’t have RME_CAPTBL_FLAG_CRT flag.
	Cap_Kmem doesn't have RME_KMEM_FLAG_PGTBL flag, or does not cover the range designated by Vaddr.
RME_ERR_CAP_EXIST	Cap_Pgtbl is not an empty capability slot.
RME_ERR_CAP_KOTBL	Kernel memory allocation failed.
RME_ERR_PGT_HW	Creation of this page directory are not allowed by the architecture.

3.3.2 Page Directory Deletion

This operation will delete a page directory. The page directory being deleted must not contain any child page directories. To delete a page directory, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_PGTBL_DEL.
Cap_Captbl	cid_t	C	A capability number that corresponds to a capability table capability that possesses the RME_CAPTBL_FLAG_DEL flag. The capability table that this capability points to contains the capability to the page directory being deleted. This capability number can have either a master encoding or a expanded encoding.
Cap_Pgtbl	cid_t	P1	A capability number that corresponds to the capability to the page directory being deleted. The capability must be a page directory capability. This capability number can only have a master encoding.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Captbl exceeded range.
	The lookup of Cap_Pgtbl exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Captbl encountered a frozen capability.
	Cap_Pgtbl is not frozen.
RME_ERR_CAP_TYPE	Cap_Captbl is not a capability table capability.
	Cap_Pgtbl is not a capability table capability.
RME_ERR_CAP_NULL	Cap_Pgtbl is an empty capability.
	Two CPUs try to delete the same page directory, and this value will be returned on the failed CPU.
RME_ERR_CAP_FLAG	Cap_Captbl doesn't have RME_CAPTBL_FLAG_DEL flag.
RME_ERR_CAP_QUIE	Cap_Pgtbl is not quiescent.
RME_ERR_CAP_REFCNT	The reference count of Cap_Pgtbl is non-zero, or it is not a root capability.
RME_ERR_PGT_HW	Deletion of such page directories are not allowed by the architecture. The possible reason is that the page directory still contains child page table mappings, or other restrictions imposed by the architecture is not satisfied.

3.3.3 Page Mapping

This page will delegate a page from one page directory to the empty slot of another. To pass pages between page directories, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_PGTBL_ADD.
Cap_Pgtbl_Dst	cid_t	P1.D1	A capability number that corresponds to a page directory capability that possesses the RME_PGTBL_FLAG_ADD_DST flag. This

			capability points to the destination page directory. This capability number can have either a master encoding or a expanded encoding.
Pos_Dst	ptr_t	P1.D0	A position in the destination page directory to accept the page delegation. This position must be empty.
Flags_Dst	ptr_t	P3.D1	The access permission flags of the destination page.
Cap_Pgtbl_Src	cid_t	P2.D1	A capability number that corresponds to a page table capability that possesses the RME_PGTBL_FLAG_ADD_SRC flag. This capability points to the destination source page directory. This capability number can have either a master encoding or a expanded encoding.
Pos_Src	ptr_t	P2.D0	The source page's position in the source page table.
Index	ptr_t	P3.D0	The subpage position to delegate in the source page.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Pgtbl_Dst ot Cap_Pgtbl_Src exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Pgtbl_Dst or Cap_Pgtbl_Src encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Pgtbl_Dst or Cap_Pgtbl_Src is not a page directory capability.
RME_ERR_CAP_FLAG	Cap_Pgtbl_Src doesn't have RME_PGTBL_FLAG_ADD_SRC flag.
	Cap_Pgtbl_Dst doesn't have RME_PGTBL_FLAG_ADD_DST flag.
	Pos is out of the operation range of Cap_Pgtbl_Dst or Cap_Pgtbl_Src.
RME_ERR_PGT_ADDR	The size order of destination page table is bigger than that of the source page directory, no delegation is possible.
	Pos_Dst or Pos_Src exceeded the maximum page number of the destination page directory or the source page directory.
	Index exceeded the maximum subpage index allowed.
	When virtual address is set to be always equal to physical address, the physical address is different from the virtual address in the destination.
RME_ERR_PGT_HW	Source page directory lookup failed. The specified position of the source page directory might be empty.
RME_ERR_PGT_MAP	The mapping attempt failed due to hardware reasons.
RME_ERR_PGT_PERM	The access permission flags of the destination page is not a subset of that of the source page.

3.3.4 Page Removal

This operation will remove a page from a page directory, and that slot will be empty after this operation. To remove pages in a page directory, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_PGTBL_REM.
Cap_Pgtbl	cid_t	P1	A capability number that corresponds to a page table capability that possesses the RME_PGTBL_FLAG_REM flag. This capability points to the destination target page directory. This capability

			number can have either a master encoding or a expanded encoding.
Pos	ptr_t	P2	The position of the page in the target page directory to unmap.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Pgtbl exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Pgtbl encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Pgtbl is not a page directory capability.
RME_ERR_CAP_FLAG	Cap_Pgtbl doesn't have RME_PGTBL_FLAG_REM flag.
	The specified position is out of the operation range of Cap_Pgtbl.
RME_ERR_PGT_ADDR	Pos exceeded the maximum page number of the target page directory.
RME_ERR_PGT_MAP	The unmapping attempt failed due to architecture-related reasons.

3.3.5 Page Table Construction

This operation will attempt to place the physical address pointer to the child page directory into the parent page directory's vacant slot. If path compression is used, the total size of the child directory must be smaller or equal than a single page of the parent directory; if not, the total size must be equal to a single page of the parent directory. To construct (map) a page directory into another, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_PGTBL_CON.
Cap_Pgtbl_Parent	cid_t	P1.D1	A capability number that corresponds to a page directory capability that possesses the RME_PGTBL_FLAG_CON_PARENT flag. This capability points to the parent page directory. This capability number can have either a master encoding or a expanded encoding.
Pos	ptr_t	P2	A position in the parent page directory to accept the page directory construction. This position must be empty.
Cap_Pgtbl_Child	cid_t	P1.D0	A capability number that corresponds to a page directory capability that possesses the RME_PGTBL_FLAG_CON_CHILD flag. This capability points to the child page directory. This capability number can have either a master encoding or a expanded encoding.
Flags_Child	ptr_t	P3	The flags of the child page directory when it is mapped in. These flags restrict the access permissions of all the page directories that are under this mapping. For different architectures, the meaning of this parameter is different. For some architectures that does not support page directory flags (all MPU-based systems), this value does not take effect.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Pgtbl_Parent or Cap_Pgtbl_Child exceeded range.

RME_ERR_CAP_FROZEN	The lookup of Cap_Pgtbl_Parent or Cap_Pgtbl_Child encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Pgtbl_Parent or Cap_Pgtbl_Child is not a page directory capability.
RME_ERR_CAP_FLAG	Cap_Pgtbl_Parent doesn't have RME_PGTBL_FLAG_CON_PARENT flag.
	Cap_Pgtbl_Child doesn't have RME_PGTBL_FLAG_CON_CHILD flag.
	Pos is out of the operation range of Cap_Pgtbl_Parent.
RME_ERR_PGT_ADDR	Pos exceeded the maximum page number of the parent page directory.
	The total size of the child page directory is bigger than a single page of the parent directory.
	When the virtual address is forced to be equal to the physical address, the physical address and the designated virtual address is different.
RME_ERR_PGT_MAP	The construction attempt failed due to architecture-related reasons.

3.3.6 Page Table Destruction

This operation will remove a child page directory from a parent directory, and the corresponding slot will return to vacant. To destruct (unmap) a page directory from another, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_PGTBL_DES.
Cap_Pgtbl	cid_t	P1	A capability number that corresponds to a page directory capability that possesses the RME_PGTBL_FLAG_DES flag. This capability points to the target page directory. This capability number can have either a master encoding or a expanded encoding.
Pos	ptr_t	P2	A position in the target page directory to remove from. This position must be a mapped page directory.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Pgtbl exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Pgtbl encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Pgtbl is not a page directory capability.
RME_ERR_CAP_FLAG	Cap_Pgtbl doesn't have RME_PGTBL_FLAG_DES flag.
	Pos is out of the operation range of Cap_Pgtbl.
RME_ERR_PGT_ADDR	Pos exceeded the maximum page number of the target page directory.
RME_ERR_PGT_MAP	The destruction attempt failed due to architecture-related reasons.

3.4 Kernel Memory System Calls

There is only one system call that is related to kernel memory, that is to delegate a kernel memory capability from one capability table to another. The initial kernel memory capability is created at

system boot-time and cannot be deleted; all of the capabilities that are derived from it can only be removed and cannot be deleted. The operation flags of kernel memory included not only the kernel objects allowed to create on the memory segment but also a range value that is 64-byte aligned.

The operation flags of kernel memory capabilities are listed as follows:

Flag	Bit	Explanation
RME_KMEM_FLAG_CAPTBL	[0]	Allows creation of capability tables.
RME_KMEM_FLAG_PGTBL	[1]	Allows creation of page directories.
RME_KMEM_FLAG_PROC	[2]	Allows creation of processes.
RME_KMEM_FLAG_THD	[3]	Allows creation of threads.
RME_KMEM_FLAG_SIG	[4]	Allows creation of signal endpoints.
RME_KMEM_FLAG_INV	[5]	Allows creation of invocation ports.

When adding kernel memory capabilities from one capability table to another, a range parameteris needed, thus a single parameter position P3 is not sufficient for passing all the information.Thus, we leverage the unused portions of system call number N and capability table capability number C to pass these parameters. The detailed format of these parameters are as follows:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N[5:0]	Must be RME_SVC_CAPTBL_ADD.
Cap_Captbl_Dst	cid_t	P1.D1	A capability number that corresponds to a capability table capability that possesses the RME_CAPTBL_FLAG_ADD_DST flag. This capability points to the destination capability table. This capability number can have either a master encoding or a expanded encoding.
Cap_Dst	cid_t	P1.D0	A capability number that corresponds to a capability slot that will accept the delegated capability. This slot must be empty. This capability number can only have a master encoding.
Cap_Captbl_Src	cid_t	P2.D1	A capability number that corresponds to a capability table capability that possesses the RME_CAPTBL_FLAG_ADD_SRC flag. This capability points to the source capability table. This capability number can have either a master encoding or a expanded encoding.
Cap_Src	cid_t	P2.D0	A capability number that corresponds to the capability to the capability table being delegated. The capability must be non-empty and not frozen. This capability number can only have a master encoding.
Flags	ptr_t	P3	See the paragraph below.
Ext_Flags	ptr_t	N:C	See the paragraph below.

In the delegation of kernel memory capabilities, the parameter P3 and N:C decided the (extended) operation flags of the new kernel memory capability. N:C means that the halfword N and halfword C is concatenated together to form a single word. Because RME only uses only 6 bits of N to pass in its system call number, thus the leftover portion can be used for other information. There are X-6 bits when N and C are combined together (X is the number of bits in a single machine word); P3 have X bits, we have 2X-6 bits available for the extended flags. The actual operation flags will occupy 6 bits,

thus the upper bound and lower bound of kernel memory can be X-6 bits respectively to specify an address that is aligned to 64-byte.

The detailed meaning of P3 (Flags) is as follows:

Bitfield	Explanation
High halfword (D1)	The high halfword of the kernel memory address upper limit.
Low halfword (D0)	The high halfword of the kernel memory address lower limit.

The detailed meaning of N:C (Ext_Flags) is as follows:

Bitfield	Explanation
High halfword with last 6 bits cleared ({D1[X-1:6]:0[5:0]})	The low halfword of the kernel memory address upper limit, aligned to a 64-byte boundary.
Low halfword with last 6 bits cleared ({D0[X-1:6]:0[5:0]})	The low halfword of the kernel memory address lower limit, aligned to a 64-byte boundary.
The last 6 bits of the high halfword ({D0[5:0]})	The operation flags of the kernel memory capability.

It is worth noting that when the operation address range is passed in, the upper limit itself is not included. For instance, if the address passed in is 0xC0000000-0xC1000000, then 0xC1000000 is not included in the address allowed for kernel object creation. In other words, the actual range allowed is 0xC0000000-0xC0FFFFFF. The upper limit shall be bigger than the lower limit or an error will be returned. The last 6 bits of upper limit and lower limit will be masked away when they are passed in to align them to 64-byte. If the kernel memory allocation table is configured to use a larger slot than 64-byte, then the lower/upper limit will be rounded up/down to the slot size.

Bibliography

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

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

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

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

Chapter 4 Process and Thread Management

4.1 Introduction

4.1.1 Introduction to Processes

In RME, a process refers to the minimal protection domain that have an address space (page table) and a capability table. The page table decides the address space of the process, and the capability table decides the permissions of the threads in the process. RME does not keep anything else about the information of the process; all these is left for user-level to implement.

As a matter of fact, RME processes are very different from the processes in traditional operating systems. The RME processes are ultra-light-weight virtual machines, and can also be regarded as a container. The capability tables of different processes have completely independent capability number namespaces, and there are no global capability numbers. It is also possible to take multiple processes as different software components that are isolated but still related. When RME processes are used as light-weight virtual machines, near-physical machine performance can be provided[4].

4.1.2 Introduction to Threads

In RME, a thread refers to a minimum schedulable entity that have one independent control flow and a stack. There are multiple flags in RME to identify its running status. All the threads that this chapter refers to are kernel-level threads; user-space threads are not related to the kernel and thus not discussed here.

There are four distinct scheduler design strategies techniques in different systems, as listed below:

1. The scheduler is completely implemented in kernel space. The timeslices of all threads are automatically replenished according to kernel-space scheduling policies. Typical systems that use implement this model are Linux and VxWorks. This strategy is very traditional thus not further discussed here.

2. The scheduler is completely implemented in user space, even the priority arbitration logic and control logic. When an interrupt occurs, the kernel will switch to the corresponding interrupt thread; the priority and thread scheduling logic is completely decided by the user-level scheduler. This strategy is extremely flexible, and facilitates Slack Stealing Scheduling implementation. This is a naive user-level scheduler implementation; however, the overhead of excessive context switches are too high and this model hardly found its way in real-world applications.

3. The preemptive priorities are implemented in kernel space and all the other portions of the scheduler is in user space. When an interrupt comes, it will preempt the current thread and run the interrupt servicing thread. The interrupt servicing thread simply runs a post-interrupt processing thread and hands off all the operations to it. If a low-priority interrupt comes in this interval, the low-priority interrupt will be converted to scheduler events which is sent to the corresponding scheduler. After the high priority post processing thread finished its execution, the scheduler will process all the scheduler events received in the period and decide the next thread to run. If the timeslice of the post-processing thread runs out before it finishes, we just pick a random thread that still have timeslices. The drawback of this model is that when a low priority thread gets unblocked due to an interrupt when a high-priority thread is executing, we cannot directly run the low-priority thread immediately after the high-priority thread. We must switch to the scheduler first and wait for the scheduler to read out all the events in this period, and we must do arbitration to pick the lower-priority thread to run. This is unacceptable in some scenarios; a typical implementation of this strategy is Composite[1].

4. The preemptive priorities and the ready queue are implemented in kernel space but all other portions like timeslice management is implemented at user-space. This strategy matches traditional systems more closely yet implements user-level scheduling. The benefit is lowered interrupt latency and we can guarantee that the current executing thread is always the thread with the highest priority. The drawback is that every kernel path that involves context switching must process the ready queue in the kernel. This is largely mitigated in RME by a very efficient kernel ready queue manager. The kernel still sends scheduler events to the user-level to facilitate user-level scheduling.

RME does not implement other functionality of threads such as Thread Local Storage (TLS). These functionality will be implemented at user-level.

Every thread have a Thread IDentifier (TID). The allocation of TID is globally incremental, and no number will be allocated twice. In 32-bit systems, from system power-on, the maximum cumulative number of threads created is 2^{32-2}; considering the fact that 32-bit systems are usually embedded systems (either microcontroller-based or low-end microprocessor-based), this restriction will not cause any practical difficulties. In 64-bit systems this number is 2^{64-2}, which is big enough in the foreseeable future.

In RME, every thread have a (preemption) priority. The bigger the priority number, the higher the priority (this is just the contrary to other systems!). The number of priorities is configured by defining a macro RME_MAX_PREEMPT_PRIO, and the priorities in the system ranges from 0 to RME_MAX_PREEMPT_PRIO-1. The macro can be configured to the order of 2 of the length of a processor halfword. For example, in 32-bit systems, the maximum number of priorities is 2^{16}=65536. A priority limit is also assigned to each thread when they are created. A thread cannot create a thread with a higher priority limit than that of itself. When changing a priority of a thread, or binding a thread to a processor, the thread's priority cannot be raised above its priority limit. However, a thread with a low priority limit can raise the priority of another thread to be higher than the low priority thread's own priority limit, given that the priority is not higher than the priority limit of the thread being operated on. The idea of priority limit is derived from seL4[2].

4.2 Process Operations and States

4.2.1 Process Creation and Deletion

To create a process, a capability table and a top-level page directory is needed. Processes serve merely as containers in RME; it does not have states itself. Deleting all threads in a process will not cause the process to be deleted.

To delete a process, all the threads and invocation ports (see the chapter on synchronous communication for details) in it must be deleted. Then, we just need to call the delete system call to delete the process by passing in its capability number.

4.2.2 Changing Capability Table or Page Table of Processes

The capability table and page table of a process can be dynamically during system runtime. There placement will take effect immediately.

4.3 Thread Operations and States

4.3.1 Overview

In RME, the threads must be binded to a certain CPU before they can be operated on, and only the CPU that have the thread binded can operate on it. If it is needed to change the CPU that can operate on it, a binding modification is needed.

The states of the threads are shown below:

State	Name	Explanation
Running	RME_THD_RUNNING	The thread is running.
Ready	RME_THD_READY	The thread is ready to be scheduled.
Timeout	RME_THD_TIMEOUT	The timeslice of the thread is exhausted.
Blocked	RME_THD_BLOCKED	The thread is blocked on some endpoint.
Fault	RME_THD_FAULT	The thread is killed due to an error in its execution.

States can transition between each other. A thread will be in RME_THD_TIMEOUT state, when it is created, which means that it haven't been binded to a CPU, and no timeslice is allocated to it. Then, we bind it to some core. After this operation it is still in RME_THD_TIMEOUT state. We will then set its entry and stack. At last, we will transfer some timeslices to it. If it is the highest priority thread on the CPU, then it will preempt the current thread and enter RME_THD_RUNNING state. Otherwise, it will be placed in the kernel ready queue, and enter RME_THD_READY state.

If the thread blocks on an endpoint during execution, it will enter RME_THD_BLOCKED state. When the thread unblocks, the system will see if the thread have the highest priority in the system. If yes, it will enter RME_THD_READY state; if no, it will enter RME_THD_RUNNING state.

If the thread triggers a fault during execution, then it will enter RME_THD_FAULT state, and send a scheduler event to its scheduler thread. To escape from this state, we need to reset its execution stack and entry point, then the thread will be in RME_THD_TIMEOUT state.

If the thread exhausts all its timeslices during execution, or it passed all its timeslices to other threads, or it decided to give up all its timeslices when switching to other threads, then it will enter RME_THD_TIMEOUT state, and a scheduler event will be sent to its scheduler thread.

When a thread is unbinded from a CPU, the thread must not be the scheduler thread of any other thread; if there is a scheduler event on its parent, the event will be removed if the thread is unbinded. If the thread is unbinded when it blocks, an error code RME_ERR_SIV_FREE will be returned (see chapter on asynchronous communication for details).

If the thread is not in RME_THD_FAULT state and is unbinded, then the thread will enter RME_THD_TIMEOUT state. If the thread is unbinded under RME_THD_FAULT state, it will remain in that state.

4.3.2 Thread Creation and Deletion

When creating threads, the process must be specified. Other properties such as entry and stack address is set by other system calls.

When deleting threads, the corresponding thread must be unbinded. When a thread gets deleted, we will clean up its invocation stack (see chapter on synchronous communication for details).

4.3.3 Binding and Freeing Threads from CPUs

The thread needs to be binded to a specific CPU before it can be operated on. If such binding needs to be changed, then the binding to the current CPU must be unbinded first.

Binding a thread to a CPU needs the thread priority and a parent thread as parameters. The thread will be binded to whichever CPU this system call is called. The binding operation is performed with a Read-Copy-Update (RCU) operation, thus among all the operations performed in parallel, only one will succeed.

Freeing a thread from binding just need the thread as a single parameter. When a thread is freed from a CPU, it can be binded to another CPU. This is different from systems that permanently make threads core-local on creation, e.g. Composite[1].

4.3.4 Setting the Execution Properties of Threads

After the thread is binded to a core, we need to set the entry and stack of the thread. These two values will be passed to the register set of the thread; when the thread runs for the first time, the user-level library will determine the entry and stack with these two parameters. It is worth noting that these two values are virtual addresses.

4.3.5 Setting the Hypervisor Properties of Threads

If built-in para-virtualizing hypervisor support is needed, the hypervisor parameters of the thread can be set. The hypervisor parameter of a thread is a pointer to hypervisor-dedicated memory. When the hypervisor properties of a thread is not set, its register context will be saved in the thread kernel object; if the hypervisor parameter is set, the register context will be saved at the designated location instead. This can facilitate the operation of user-level virtual machine monitors by enabling their modification of register set contents.

4.3.6 Timeslice Allocation, Priority Modification and Thread Execution

After setting the entry and stack of the thread, some timeslices can be allocated for it, so that the thread can run. The timeslice allocation of RME is organized by user-level scheduler trees, and each CPU should have such a tree to manage its timeslice allocations. The Init thread on each CPU will allocate timeslice for its user-level scheduler (the timeslice of Init is RME_THD_INIT_TIME which is infinite), then these user-level schedulers will transfer their timeslices to their child schedulers, and so on,to complete the hierarchical scheduling scheme. This infrastructure makes paravirtualizing other operating systems easy. The Init threads on each CPU holds infinite timeslices, and if there are no other threads to run, the Init thread on tha CPU will be scheduled.

After the timeslice allocation is finished, the thread will be placed in the core-local scheduling queue on each core, and its priority will be compared against the current thread. If the current thread have a lower priority, the thread will be scheduled immediately.

We can modify the priority of a thread that is binded to a CPU core. After such modification, if the thread have the highest priority and it is in RME_THD_READY state, it will be scheduled immeditely.

The idea of timeslice transfers are borrowed from Composite's TCap mechanism, and improvements and simplifications are applied[3]. There are three types of threads in the system when classified according to their amount of timeslice, as shown below:

Category	Time of creation	Explanation
Regular	After system boots	Limited timeslice, which is smaller than RME_THD_MAX_TIME.
Infinite	After system boots	Infinite timeslice, denoted as RME_THD_INF_TIME.

Init	When system boots	Infinite timeslice, denoted as RME_THD_INIT_TIME.

The explanation of these macros are explained below:

Macro	Explanation
RME_THD_INIT_TIME	The maximum positive integer value, 0x7FFFFFFF for 32-bit.
RME_THD_INF_TIME	Always RME_THD_INIT_TIME-1, 0x7FFFFFFE for 32-bit.
RME_THD_MAX_TIME	Always equal to RME_THD_INF_TIME.

Among the three categories, the regular thread and infinite thread are allowed to block, or lose all their timeslices. The Init thread does not allow either of them. There are also three ways of timeslice passing: the first being regular transfers, these transfers will always transfer a finite amount of time to other threads; the second being infinite transfers, these transfers will transfer an infinite amount of timeslice to other threads. The third variant is revoking transfer, it will transfer all the timeslice of the source thread to the target thread. It is worth noting that all the transfers are atomic; the timeslice will be transferred in a all-or-none fashion, and will never be partially transferred. The rules of the three transfers are listed below:

Regular	Source	Init thread	Infinite thread	Regular thread
Target	Init thread	--	--	T-
	Infinite thread	--	--	T-
	Regular thread	-A	TA	TA

Infinite	Source	Init thread	Infinite thread	Regular thread
Target	Init thread	--	--	S-
	Infinite thread	--	--	S-
	Regular thread	-I	-I	TA

Revoking	Source	Init thread	Infinite thread	Regular thread
Target	Init thread	--	S-	S-
	Infinite thread	--	S-	S-
	Regular thread	-I	SI	TA

In the table:

"--" means that there is no effect on both source thread and target thread;

"T" means that if all timeslices of the source thread is transferred, the source will timeout.

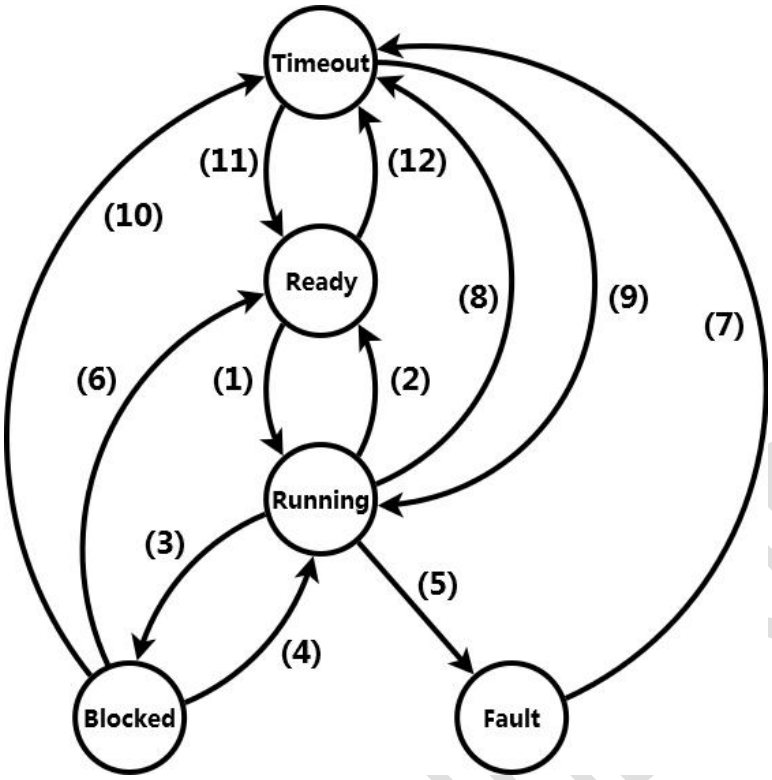
"S" means that the source will definitely timeout.

"A" means that if the target thread's timeslice does not overflow, the timeslices will be accepted.

"I" means that the target thread will definitely become an infinite thread.

4.3.7 Thread Scheduling Overview

The complete thread state transition diagram is shown below:



The meaning of the number labels are shown below:

Label	Explanation
(1)	It is the thread with the highest priority on this CPU and will transition to RUNNING from READY.
(2)	It is preempted by a thread on the same CPU with a higher priority, and will transition to READY state from RUNNING state.
(3)	The thread blocked at an endpoint and will transition from RUNNING to BLOCKED.
(4)	The thread received a signal from an endpoint, and is thus unblocked. It is the thread with the highest priority on this CPU, and its timeslice is not exhausted, thus will transition from BLOCKED to RUNNING.
(5)	An unrecoverable fault happened during the execution of the thread, and it will transition from RUNNING to FAULT.
(6)	The thread received a signal from an endpoint, and is thus unblocked. It is not the thread with the highest priority on this CPU, and its timeslice is not exhausted, thus will transition from BLOCKED to READY.
(7)	The thread's execution properties are reset after a fault, however its timeslices are deprived of during the fault handling, thus it will transition from FAULT to RUNNING.
(8)	The thread exhausted its timeslices during execution, and thus will transition from RUNNING to TIMEOUT.
(9)	The thread got some timeslice from other threads and is executable again. It is the thread with the highest priority on this CPU, and thus will transition from TIMEOUT to RUNNING.
(10)	The thread got all of its timeslice transferred to another thread when it is BLOCKED, thus its own timeslices zeroed. It will transition from BLOCKED to TIMEOUT.
(11)	The thread got some timeslices transferred to it and returned to an executable state. However

	it is not the thread with the highest priority on the CPU, and thus will transition from TIMEOUT to READY.
(12)	The thread got all of its timeslice transferred to another thread when it is READY, thus its own timeslices zeroed. It will transition from READY to TIMEOUT.

4.4 Process System Calls

The system calls that are related to the process are listed as follows:

System call number	Type	Explanation
RME_SVC_PROC_CRT	System call	Create a process
RME_SVC_PROC_DEL	System call	Delete a process
RME_SVC_PROC_CPT	System call	Replace the capability table of the process
RME_SVC_PROC_PGT	System call	Replace the page table (top-level page directory) of the process

The operation flags of process capabilities are listed as follows:

Flag	Bit	Explanation
RME_PROC_FLAG_INV	[0]	Allows creation of invocation ports in the process.
RME_PROC_FLAG_THD	[1]	Allows creation of threads in the process.
RME_PROC_FLAG_CPT	[2]	Allows replacement of the process's capability table.
RME_PROC_FLAG_PGT	[3]	Allows replacement of the process's page table.

For detailed information about bit[0], please refer to synchronous commnucation related sections.

4.4.1 Process Creation

This operation will create a process and place its capability into some capability table. The newly created process will reference its capability table capability and page directory process. If a pair of capability table/page directory capability is used to create a process, this pair of capabilities cannot be removed/deleted before the process gets deleted. To create a process, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_PROC_CRT.
Cap_Captbl_Crt	cid_t	C	A capability number that corresponds to a capability table capability that possesses the RME_CAPTBL_FLAG_CRT flag. The capability table that this capability points to will accept the newly created process capability. This capability number can have either a master encoding or a expanded encoding.
Cap_Kmem	cid_t	P1.D1	A kernel memory capability, whose kernel memory range must be big enough to contain the whole process. This capability must have the RME_KMEM_FLAG_PROC flag.This capability number can have either a master encoding or a expanded encoding.
Cap_Proc	cid_t	P1.D0	A capability number that corresponds to a capability slot that will accept the newly created process capability. This slot must be empty. This capability number can only have a master

			encoding.
Cap_Captbl	cid_t	P2.D1	A capability number that corresponds to a capability table capability that have the RME_CAPTBL_FLAG_PROC_CRT flag. This capability points to the capability table that will be used by the new process. This capability number can have either a master encoding or a expanded encoding.
Cap_Pgtbl	cid_t	P2.D0	A capability number that corresponds to a page directory capability that have the RME_PGTBL_FLAG_PROC_CRT flag. This capability points to the page table (top-level page directory) that will be used by the new process..This capability number can have either a master encoding or a expanded encoding.
Vaddr	ptr_t	P3	The kernel virtual address that this new process will use.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Captbl_Crt exceeded range.
	The lookup of Cap_Kmem exceeded range.
	The lookup of Cap_Captbl exceeded range.
	The lookup of Cap_Pgtbl exceeded range.
	The lookup of Cap_Proc exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Captbl_Crt encountered a frozen capability.
	The lookup of Cap_Kmem encountered a frozen capability.
	The lookup of Cap_Captbl encountered a frozen capability.
	The lookup of Cap_Pgtbl encountered a frozen capability.
	Cap_Proc is frozen, or other CPUs are creating capabilities there.
RME_ERR_CAP_TYPE	Cap_Captbl_Crt or Cap_Captbl is not a capability table capability.
	Cap_Kmem is not a kernel memory capability.
	Cap_Pgtbl is not a page directory capability.
RME_ERR_CAP_FLAG	Cap_Captbl_Crt doesn't have RME_CAPTBL_FLAG_CRT flag.
	Cap_Kmem doesn't have RME_KMEM_FLAG_PROC flag, or does not cover the range designated by Vaddr.
	Cap_Captbl doesn't have RME_CAPTBL_FLAG_PROC_CRT flag.
	Cap_Pgtbl doesn't have RME_PGTBL_FLAG_PROC_CRT flag.
RME_ERR_CAP_EXIST	Cap_Proc is not an empty capability slot.
RME_ERR_CAP_KOTBL	Kernel memory allocation failed.
RME_ERR_CAP_REFCNT	The reference count of Cap_Captbl or Cap_Pgtbl exceeded the maximum reference count allowed.

4.4.2 Process Deletion

This operation will delete a process. The process being deleted must not contain any threads or invocation ports (see the chapter on synchronous communication for details).To delete a process, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_PROC_DEL.
Cap_Captbl	cid_t	C	A capability number that corresponds to a capability table capability that possesses the RME_CAPTBL_FLAG_DEL flag. The capability table that this capability points to contains the capability to the process being deleted. This capability number can have either a master encoding or a expanded encoding.
Cap_Proc	cid_t	P1	A capability number that corresponds to the capability to the process being deleted. The capability must be a process capability. This capability number can only have a master encoding.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Captbl exceeded range.
	The lookup of Cap_Proc exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Captbl encountered a frozen capability.
	Cap_Proc is not frozen.
RME_ERR_CAP_TYPE	Cap_Captbl is not a capability table capability.
	Cap_Proc is not a process capability.
RME_ERR_CAP_NULL	Cap_Proc is an empty capability.
	Two CPUs try to delete the same process, and this value will be returned on the failed CPU.
RME_ERR_CAP_FLAG	Cap_Captbl does not have RME_CAPTBL_FLAG_DEL flag.
RME_ERR_CAP_QUIE	Cap_Proc is not quiescent.
RME_ERR_CAP_REFCNT	The reference count of Cap_Proc is non-zero, or it is not a root capability.
RME_ERR_PTH_REFCNT	The reference count of the process is non-zero (it still contains threads or invocation ports).

4.4.3 Changing Process Capability Table

This operation will replace the process's capability table with another one. The replacement is immediately effective. To replace the capability table of a process, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_PROC_CPT.
Cap_Proc	cid_t	P1	A capability number that corresponds to a process capability that possesses the RME_PROC_FLAG_CPT flag. The process that this capability points to will have its capability table replaced. This capability number can have either a master encoding or a expanded encoding.
Cap_Captbl	cid_t	P2	A capability number that corresponds to a capability table capability that possesses the RME_CAPTBL_FLAG_PROC_CPT flag. The capability table that this capability points to will be the

			process's new capability table. This capability number can have either a master encoding or a expanded encoding.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Proc exceeded range.
	The lookup of Cap_Captbl exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Proc encountered a frozen capability.
	The lookup of Cap_Captbl encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Proc is not a process capability.
	Cap_Captbl is not a capability table capability.
RME_ERR_CAP_NULL	Cap_Proc or Cap_Captbl is an empty capability.
RME_ERR_CAP_FLAG	Cap_Proc doesn't have RME_PROC_FLAG_CPT flag.
	Cap_Captbl doesn't have RME_CAPTBL_FLAG_PROC_CPT flag.
RME_ERR_CAP_REFCNT	The reference count of Cap_Captbl exceeded the maximum reference count allowed.
RME_ERR_PTH_CONFLICT	Two CPUs try to change the capability table of the same process, and this value will be returned on the failed CPU.

4.4.4 Changing Process Page Table

This operation will replace the process's page table with another one. The replacement is immediately effective. To replace the page table of a process, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_PROC_PGT.
Cap_Proc	cid_t	P1	A capability number that corresponds to a process capability that possesses the RME_PROC_FLAG_PGT flag. The process that this capability points to will have its page table (top-level page directory) replaced. This capability number can have either a master encoding or a expanded encoding.
Cap_Pgtbl	cid_t	P2	A capability number that corresponds to a page table capability that possesses the RME_PGTBL_FLAG_PROC_PGT flag. The page table (top-level page directory) that this capability points to will be the process's new page table. This capability number can have either a master encoding or a expanded encoding.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Proc exceeded range.
	The lookup of Cap_Pgtbl exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Proc encountered a frozen capability.
	The lookup of Cap_Pgtbl encountered a frozen capability.

RME_ERR_CAP_TYPE	Cap_Proc is not a process capability.
	Cap_Pgtbl is not a page table (top-level page directory) capability.
RME_ERR_CAP_NULL	Cap_Proc or Cap_Pgtbl is an empty capability.
RME_ERR_CAP_FLAG	Cap_Proc doesn't have RME_PROC_FLAG_PGT flag.
	Cap_Pgtbl doesn't have RME_PGTBL_FLAG_PROC_PGT flag.
RME_ERR_CAP_REFCNT	The reference count of Cap_Pgtbl exceeded the maximum reference count allowed.
RME_ERR_PTH_CONFLICT	Two CPUs try to change the page table of the same process, and this value will be returned on the failed CPU.

4.5 Thread System Calls

The system calls that are related to the thread are listed as follows:

System call number	Type	Explanation
RME_SVC_THD_CRT	System call	Create a thread
RME_SVC_THD_DEL	System call	Delete a thread
RME_SVC_THD_EXEC_SET	System call	Set the execution properties (entry & stack)
RME_SVC_THD_HYP_SET	System call	Set the hypervisor properties (register location)
RME_SVC_THD_SCHED_BIND	System call	Bind a thread to a CPU
RME_SVC_THD_SCHED_RCV	System call	Receive thread scheduler events
RME_SVC_THD_SCHED_PRIO	System call	Change a thread's priority
RME_SVC_THD_SCHED_FREE	System call	Free a thread from CPU
RME_SVC_THD_TIME_XFER	System call	Transfer timeslices between threads
RME_SVC_THD_SWT	System call	Switch to a thread with the same priority

The operation flags of thread capabilities are listed as follows:

Flag	Bit	Explanation
RME_THD_FLAG_EXEC_SET	[0]	Allows setting the execution properties.
RME_THD_FLAG_HYP_SET	[1]	Allows setting the hypervisor properties.
RME_THD_FLAG_SCHED_CHILD	[2]	Allows usage as the child thread in binding.
RME_THD_FLAG_SCHED_PARENT	[3]	Allows usage as the parent thread in binding.
RME_THD_FLAG_SCHED_PRIO	[4]	Allows changing the thread's priority.
RME_THD_FLAG_SCHED_FREE	[5]	Allows freeing the thread from a CPU.
RME_THD_FLAG_SCHED_RCV	[6]	Allows receiving the thread's scheduler notifications.
RME_THD_FLAG_XFER_SRC	[7]	Allows usage as a source in timeslice transfers.
RME_THD_FLAG_XFER_DST	[8]	Allows usage as a destination in timeslice transfers.
RME_THD_FLAG_SWT	[9]	Allows switch operations to switch to the thread.

4.5.1 Thread Creation

This operation will create a thread and place its capability into a capability table, then return its ID. The newly created thread is not binded to any CPU, and will be in RME_THD_TIMEOUT state. This thread will reference the process that it is in, and if a thread is created in a process, the process cannot be deleted before the thread is deleted. To create a thread, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_THD_CRT.

Cap_Captbl	cid_t	C	A capability number that corresponds to a capability table capability that possesses the RME_CAPTBL_FLAG_CRT flag. The capability table that this capability points to will accept the newly created thread capability. This capability number can have either a master encoding or a expanded encoding.
Cap_Kmem	cid_t	P1.D1	A kernel memory capability, whose kernel memory range must be big enough to contain the whole thread. This capability must have the RME_KMEM_FLAG_THD flag. This capability number can have either a master encoding or a expanded encoding.
Cap_Thd	cid_t	P1.D0	A capability number that corresponds to a capability slot that will accept the newly created thread capability. This slot must be empty. This capability number can only have a master encoding.
Cap_Proc	cid_t	P2.D1	A capability number that corresponds to a thread capability that possesses the RME_PROC_FLAG_THD flag. The process that this capability points to will contain the new thread. This capability number can have either a master encoding or a expanded encoding.
Max_Prio	ptr_t	P2.D0	The maximum priority limit of the thread.
Vaddr	ptr_t	P3	The kernel virtual address that this new thread will use.

The possible return values of this operation are:

Return	Explanation
Non-nagative value	Operation succesful, Thread IDentifier (TID) is returned.
RME_ERR_CAP_RANGE	The lookup of Cap_Captbl exceeded range.
	The lookup of Cap_Kmem exceeded range.
	The lookup of Cap_Proc exceeded range.
	The lookup of Cap_Thd exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Captbl encountered a frozen capability.
	The lookup of Cap_Kmem encountered a frozen capability.
	The lookup of Cap_Proc encountered a frozen capability.
	Cap_Thd is frozen, or other CPUs are creating capabilities there.
RME_ERR_CAP_TYPE	Cap_Captbl is not a capability table capability.
	Cap_Kmem is not a kernel memory capability.
	Cap_Proc is not a process capability.
RME_ERR_CAP_FLAG	Cap_Captbl doesn't have RME_CAPTBL_FLAG_CRT flag.
	Cap_Kmem doesn't have RME_KMEM_FLAG_THD flag, or does not cover the range designated by Vaddr.
	Cap_Proc doesn't have RME_PROC_FLAG_THD flag.
RME_ERR_CAP_EXIST	Cap_Thd is not an empty capability slot.
RME_ERR_CAP_KOTBL	Kernel memory allocation failed.
RME_ERR_PTH_PRIO	Attempted to create a thread with a higher priority limit, or the priority limit exceeded maximum allowed value determined at system configuration time

4.5.2 Thread Deletion

This operation will delete a thread. The thread deleted must be unbinded from CPU first. To delete a thread, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_THD_DEL.
Cap_Captbl	cid_t	C	A capability number that corresponds to a capability table capability that possesses the RME_CAPTBL_FLAG_DEL flag. The capability table that this capability points to contains the capability to the thread being deleted. This capability number can have either a master encoding or a expanded encoding.
Cap_Thd	cid_t	P1	A capability number that corresponds to the capability to the thread being deleted. The capability must be a thread capability. This capability number can only have a master encoding.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Captbl exceeded range.
	The lookup of Cap_Thd exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Captbl encountered a frozen capability.
	Cap_Thd is not frozen.
RME_ERR_CAP_TYPE	Cap_Captbl is not a capability table capability.
	Cap_Thd is not a thread capability.
RME_ERR_CAP_NULL	Cap_Thd is an empty capability.
	Two CPUs try to delete the same thread, and this value will be returned on the failed CPU.
RME_ERR_CAP_FLAG	Cap_Captbl doesn't have RME_CAPTBL_FLAG_DEL flag.
RME_ERR_CAP_QUIE	Cap_Thd is not quiescent.
RME_ERR_CAP_REFCNT	The reference count of Cap_Thd is non-zero, or it is not a root capability.
RME_ERR_PTH_INVSTATE	The thread is still binded to a CPU.

4.5.3 Setting Thread Execution Property

This operation will set the execution property of the thread, in other words, its entry and stack. The thread must be binded to a CPU, and the execution property must be set on this CPU. For a thread in RME_THD_FAULT state, setting its execution property will turn it into RME_THD_TIMEOUT state. When the Entry and Stack passed in are both 0 (NULL) which is a special value, the execution property of the thread will not be changed, and only the state will be changed. This is very useful in fault handling. To set execution property for a thread, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_THD_EXEC_SET.
Cap_Thd	cid_t	P1	A capability number that corresponds to a thread capability that possesses the RME_THD_FLAG_EXEC_SET flag. The thread that this capability points to will have its execution properties set.

			This capability number can have either a master encoding or a expanded encoding.
Entry	ptr_t	P2	The entry of the thread. This is a virtual address in the process, and the thread will start its execution from here.
Stack	ptr_t	P3	The stack of the thread. This is a virtual address in the process, and the thread's initial stack pointer will be here. Whether it is a ascending stack or descending stack is decided by the user-level library.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Thd exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Thd encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Thd is not a thread capability, or is an empty capability.
RME_ERR_CAP_FLAG	Cap_Thd doesn't have RME_THD_FLAG_EXEC_SET flag.
RME_ERR_PTH_INVSTATE	The thread is not binded to a CPU.

4.5.4 Setting Thread Hypervisor Property

This operation will set the hypervisor properties of a thread, in other words, the registers' save address of the thread. Under default state (when the thread is created for the first time), the register set of the thread is stored inside the kernel object by default; if the hypervisor property of the thread is set, the register will be saved to the designated kernel-user shared memory region. To set the hypervisor properties of a thread, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_THD_HYP_SET.
Cap_Thd	cid_t	P1	A capability number that corresponds to a thread capability that possesses the RME_THD_FLAG_HYP_SET flag. The thread that this capability points to will have its hypervisor properties set. This capability number can have either a master encoding or a expanded encoding.
Kaddr	ptr_t	P2	The virtual address to save the register set to. It must be kernel accessible and bigger than or equal to RME_HYP_VA_START, smaller than RME_HYP_VA_START+RME_HYP_SIZE, and needs to be word-aligned. If this value is set to 0, the hypervisor property of the thread will be cleared and the thread will return to initial state; its register set will be stored in the kernel object.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Thd exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Thd encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Thd is not a thread capability, or is an empty capability.
RME_ERR_CAP_FLAG	Cap_Thd doens't have RME_THD_FLAG_HYP_SET flag.

RME_ERR_PTH_INVSTATE	The thread is not binded to a CPU.
RME_ERR_PTH_PGTBL	Kaddr is unaligned or not in the designated hypervisor dedicated memory.

4.5.5 Binding Thread to CPU

This operation will bind a thread to a CPU. The thread must be not binded to any CPU to make this call. The thread will be binded to wherever this function is called. To bind a thread to a certain CPU, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_THD_SCHED_BIND.
Cap_Thd	cid_t	P1	A capability number that corresponds to a thread capability that possesses the RME_THD_FLAG_SCHED_CHILD flag. The thread that this capability points to is the child thread. This capability number can have either a master encoding or a expanded encoding.
Cap_Thd_Sched	cid_t	P2	A capability number that corresponds to a thread capability that possesses the RME_THD_FLAG_SCHED_PARENT flag. The thread that this capability points to is the parent thread. The parent thread must be already binded to the same CPU. This capability number can have either a master encoding or a expanded encoding.
Prio	ptr_t	P3	The preemption priority of the child thread to be binded. The larger the number, the higher the priority. This value cannot exceed the priority limit of the thread.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Thd or Cap_Thd_Sched exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Thd or Cap_Thd_Sched encountered a frozen capability.
RME_ERR_CAP_TYPE	The lookup of Cap_Thd or Cap_Thd_Sched is not a thread capability, or is an empty capability.
RME_ERR_CAP_FLAG	Cap_Thd doesn't have RME_THD_FLAG_SCHED_CHILD flag.
	Cap_Thd_Sched doesn't have RME_THD_FLAG_SCHED_PARENT flag.
RME_ERR_PTH_NOTIF	Attempt to register the thread itself as its own parent thread.
RME_ERR_PTH_PRIO	The priority assigned exceeded its maximum priority limit.
RME_ERR_PTH_INVSTATE	Cap_Thd is binded or Cap_Thd_Sched is not binded.
	Cap_Thd_Sched is binded to another CPU.
RME_ERR_PTH_CONFLICT	Two CPUs try to bind the same thread, and this value will be returned on the failed CPU.

4.5.6 Changing Thread Priority

This operation will change the priority of a thread that is already binded to a CPU. This operation must happen on the same CPU where the thread is binded to. To change a thread's priority, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_THD_SCHED_PRIO.
Cap_Thd	cid_t	P1	A capability number that corresponds to a thread capability that possesses the RME_THD_FLAG_SCHED_PRIO flag. The thread that this capability points to will have its preemptive priority changed. This capability number can have either a master encoding or a expanded encoding.
Prio	ptr_t	P2	The new preemptive priority of the thread. The larger the number, the higher the priority. This value cannot exceed the priority limit of the thread.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Thd exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Thd encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Thd is not a thread capability, or is an empty capability.
RME_ERR_CAP_FLAG	Cap_Thd doesn't have RME_THD_FLAG_SCHED_PRIO flag.
RME_ERR_PTH_PRIO	The priority assigned exceeded its maximum priority limit.
RME_ERR_PTH_INVSTATE	Cap_Thd is not binded.
	Cap_Thd is binded to another CPU.

4.5.7 Freeing Thread from CPU

This operation will free a thread from where it is binded. The thread cannot have any child threads. If the thread sent a scheduler notification to its parent thread, then the notification will be canceled. If the thread is blocked, the blocking will end and will return RME_ERR_SIV_FREE.To free a thread from a certain CPU, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_THD_SCHED_FREE.
Cap_Thd	cid_t	P1	A capability number that corresponds to a thread capability that possesses the RME_THD_FLAG_SCHED_FREE flag. The thread that this capability points to will be freed from its current CPU. This capability number can have either a master encoding or a expanded encoding.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Thd exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Thd encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Thd is not a thread capability, or is an empty capability.
RME_ERR_CAP_FLAG	Cap_Thd doesn't have RME_THD_FLAG_SCHED_FREE flag.

RME_ERR_PTH_REFCNT	Cap_Thd still is the parent thread of some other threads (its child threads haven't been all freed).
RME_ERR_PTH_INVSTATE	Cap_Thd is not binded.
	Cap_Thd is binded to another CPU.

4.5.8 Receiving Thread Scheduler Events

This operation will receive the scheduler notifications of a thread. This operation will not block.Ifthe thread doesn't have a scheduler notification, we will return a negative value. If the value returned is positive, then it will consist of a fault indicator (which is located at the second last bit) and a TID (all other bits). To receive scheduler notifications of a certain thread, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_THD_SCHED_RCV.
Cap_Thd	cid_t	P1	A capability number that corresponds to a thread capability that possesses the RME_THD_FLAG_SCHED_RCV flag. We are trying to receive scheduler notifications from this thread's child threads. This capability number can have either a master encoding or a expanded encoding.

The possible return values of this operation are:

Return	Explanation
Non-negative	Operation successful. If the fault indicator is zero, then the event received is a timeout event, which means that the thread this TID corresponds to have timeout by exhausting all its timeslices. If the fault indicator is 1, then it means that the thread have encountered a fault during execution.
RME_ERR_CAP_RANGE	The lookup of Cap_Thd exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Thd encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Thd is not a thread capability, or is an empty capability.
RME_ERR_CAP_FLAG	Cap_Thd doesn't have RME_THD_FLAG_SCHED_RCV flag.
RME_ERR_PTH_NOTIF	Cap_Thd doesn't have pending scheduler events.
RME_ERR_PTH_INVSTATE	Cap_Thd is not binded.
	Cap_Thd is binded to another CPU.

4.5.9 Transferring Execution Timeslices

This operation can be used to transfer timeslices from one thread to another. The delegating thread (source) and the receiving thread (destination) must be on the same CPU, and the function shall be called from the same CPU, to guarantee that we are transferring timeslices on this CPU. The timeslices transferred must be non-zero, and the target cannot be in fault (RME_THD_FAULT) state. To transfer timeslices, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_THD_TIME_XFER.
Cap_Thd_Dst	cid_t	P1	A capability number that corresponds to a thread capability that possesses the RME_THD_FLAG_XFER_DST flag. The thread

			that this capability points to is the destination thread. This capability number can have either a master encoding or a expanded encoding.
Cap_Thd_Src	cid_t	P2	A capability number that corresponds to a thread capability that possesses the RME_THD_FLAG_XFER_SRC flag. The thread that this capability points to is the source thread. This capability number can have either a master encoding or a expanded encoding.
Time	ptr_t	P3	The number of timeslices to transfer. The unit of this time value isin timeslices, whose granularity is decided at system compile time. This value cannot be 0. Pass in RME_THD_INF_TIME for infinite transfers. Pass in RME_THD_INIT_TIME for revoking transfers. Pass in other non-zero values for regular transfers.

The possible return values of this operation are:

Return	Explanation
Non-negative	Operation succeeded. The number of current timeslices will be returned. If infinite transfers and revoking transfers are performed, RME_THD_MAX_TIME will be returned.
RME_ERR_CAP_RANGE	The lookup of Cap_Thd_Dst or Cap_Thd_Src exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Thd_Dst or Cap_Thd_Src encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Thd_Dst or Cap_Thd_Src is not a thread capability, or is an empty capability.
RME_ERR_CAP_FLAG	Cap_Thd_Dst doesn't have RME_THD_FLAG_XFER_DST flag.
	Cap_Thd_Src doesn't have RME_THD_FLAG_XFER_SRC flag.
RME_ERR_PTH_FAULT	Cap_Thd_Dst is in RME_THD_FAULT state.
RME_ERR_PTH_INVSTATE	Cap_Thd_Dst or Cap_Thd_Src is not binded.
	Cap_Thd_Dst or Cap_Thd_Src is binded to another CPU.
RME_ERR_PTH_OVERFLOW	The timeslice of the target thread is full (if more is accepted, it will exceed RME_THD_MAX_TIME which is the maximum allowed value). This error is very rare, because the number of timeslices transferred hardly amounts to that.

4.5.10 Switching To Thread

This operation allows the user-level scheduler to switch to a thread that is binded on the same CPU to facilitate user-level scheduling. The thread switched to must have the same priority with the current thread, and shall be in ready (RME_THD_READY) state. To switch to a certain thread, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_THD_SWT.
Cap_Thd	cid_t	P1	A capability number that corresponds to a thread capability that possesses the RME_THD_FLAG_SWT flag. The thread that this capability points to is the target thread to switch to. This capability

			number can have either a master encoding or a expanded encoding. If you wish the kernel to decide what to run, you can alternatively pass in RME_THD_ARBITRARY.
Full_Yield	ptr_t	P2	Whether to give up all timeslices in this yield. If this is non-zero, this switch will give up all the timeslices that this thread have. If this is called in the Init thread, then it is not effective because the Init thread always have infinite timeslices.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Thd exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Thd encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Thd is not a thread capability, or is an empty capability.
RME_ERR_CAP_FLAG	Cap_Thd doesn't have RME_THD_FLAG_THD_SWT flag.
RME_ERR_PTH_FAULT	Cap_Thd is in RME_THD_FAULT state.
RME_ERR_PTH_INVSTATE	Cap_Thd is not binded.
	Cap_Thd is binded to another CPU.
	Cap_Thd is in blocked (RME_THD_BLOCKED) state.
	Cap_Thd is in timeout (RME_THD_TIMEOUT) state.
RME_ERR_PTH_PRIO	Cap_Thd have a different priority from the current thread.

Bibliography

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

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

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

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

Chapter 5 Synchronous and Asynchronous Communication

5.1 Introduction

5.1.1 Introduction to Synchronous Communication

The synchronous communication mechanisms in an operating system is used to synchronize two parallel or concurrent processes. In traditional operating systems these mechanisms are usually implemented in pipes and semaphores. The most distinct feature of synchronous communication is that it may block. If the sender sends and the receiver does not receive, then the sender will block until the receiver receives the information. This traditional synchronous communication model requires 2 distinct threads to communicate between different processes.

In RME, the synchronous communication model is further simplified to thread migration-based invocation mechanism (or local procedure calls). Its main mechanism is allowing one thread to execute a piece of code in another process, and return to its own process to continue execution. This is one of the most efficient IPC implementations. Due to the fact that the invocation allows process boundary crossing, it should only be used wen two processes highly trust each other. The apparent outcome of n invocation is one function in a process gets called by another process, which allows single-thread Inter-Process Communication (IPC). RME's synchronous invocation allows carrying one parameter. If more parameters are needed, they can be passed by shared memory. Synchronous invocation can be nested, and the nesting can be infinite.

The design of synchronous invocation is adapted from Composite[1] and Mach 3[2].

5.1.2 Introduction to Asynchronous Communication

The asynchronous communication mechanisms in the operating system are used to coordinate the producer-consumer problems. Typical such mechanisms include mailboxes and message queues. The distinct feature of asynchronous communication is that it will never block. If the sender sends and the receivers doesn't receive, the information will be cached first, and the sender will return directly. When the receiver tries to receive, if there is information, the receiver will return with the information; if there is nothing, the receiver may block indefinitely, or block for a specified period, or return immediately.

In RME, asynchronous communication is further simplified to signal endpoints. The major mechanism is that the sender can send to the signal endpoint, and the receiver can receive from the same signal endpoint. The sending is never blocking, and receiving is also non-blocking if there is any signals. If there are no signals, the receiver always blocks forever. The signal only carries a numerical information (there is only one counter), and does not carry anything else. If other information needs to be passed, user-level memory sharing is necessary.

5.2 Synchronous Communication Operations

To use synchronous communication, it is necessary to create a synchronous invocation port first. When creating the port, a process needs to be specified. After creation, we must set the entry and stack of that synchronous invocation. Now the synchronous invocation port may be activated, and the thread can migrate to another process to continue execution. If another synchronous invocation is executed in the process, then the synchronous invocations will become nested. After the synchronous invocation finishes, the synchronous invocation return system call needs to be called to return from it. Compared with asynchronous communication, synchronous communication is always immediate and guarantees

minimum response time. When a thread migrates to other processes, nothing except for the protection domain and execution properties changes. It is worth noting that the invocation will not save any general purpose registers nor change the status and register contents of coprocessors; it will only change the registers that are necessary for program execution control, such as stack pointer and program counter. The value of general-purpose registers and coprocessor status will be retained upon entering and exiting of the synchronous invocation. If some registers or coprocessor statuses need to be saved, the user-level assumes this responsibility. Additionally, the coprocessor registers can be used to pass extra parameters or receive extra return values as well.

Invocation ports are not binded to a CPU. If multiple threads from different CPUs try to activate the same port, only one thread will succeed, and all other threads will get an error code; if an synchronous invocation port gets activated again (on the same CPU or not) while it is active , an error code will be returned.

If nesting happens during an invocation, the return will be performed layer-by-layer. If the thread is unbinded while in synchronous invocation, it will stop execution, and it will resume execution from where it stopped after it is binded to another CPU. If an error happened during invocation execution, the thread's behavior is decided by the parameter Fault_Ret_Flag that is passed in when setting of the execution properties of the invocation port. If this parameter is non-zero, the thread will not stop its execution, and will simply return from the synchronous invocation with an error code. If this parameter is 0, then the thread will enter RME_THD_FAULT state and wait for error handling.

5.3 Asynchronous Communication Operations

To use asynchronous communication, it is necessary to create a signal endpoint first. Signals can be sent to the endpoint, and the receiver can receive notifications from it. If there is no signal when the receiver tries to receive, the receiver will block; if receive is successful, the return value will be number of signals left.

Signal endpoints are not binded to a specific CPU. If multiple threads on different CPUs try to block on the same endpoint, only one thread will succeed, and all other threads will get an error code; if another thread tries to block on the endpoint where another thread (on the same CPU or not) has already blocked on, an error code will be returned. In practice, it is not recommended to have multiple receivers on one endpoint because this is very difficult to manage.

It is worth noting that when a thread blocks on the endpoint, only the sender threads on the same CPU with it can unblock it. The senders on other CPUs may send to the endpoint but this will only increase the signal count and not unblock the receiver.

RME doesn't allow Init thread to receive on any endpoint. This is due to the fact that receiving from an endpoint is a potentially blocking operation; if the Init thread blocks, there's no guarantee that RME can always find a thread to run on that core. If we need a thread that have infinite timeslice yet still can block, we can create a thread and make an infinite transfer from Init to it. This will make it an infinite thread.

5.4 Synchronous Invocation System Calls

The system calls that are related to the synchronous invocation are listed as follows:

System call number	Type	Explanation
RME_SVC_INV_CRT	System call	Create a synchronous invocation

RME_SVC_INV_DEL	System call	Delete a synchronous invocation
RME_SVC_INV_SET	System call	Set the execution properties of an invocation port
RME_SVC_INV_ACT	System call	Activate (call) the invocation port
RME_SVC_INV_RET	System call	Return from an invocation

The operation flags of invocation capabilities are listed as follows:

Flag	Bit	Explanation
RME_INV_FLAG_SET	[0]	Allows setting the execution properties of the invocation port.
RME_INV_FLAG_ACT	[1]	Allows activating the invocation port.

It is worth noting that we do not need an invocation return capability to return from an invocation, thus there is no such flag. Consult the following sections for details.

5.4.1 Synchronous Invocation Creation

This operation will create a synchronous invocation port and place its capability into a capability table. This invocation will reference the process that it is in, and if a, invocation is created in a process, the process cannot be deleted before the invocation is deleted. To create an invocation capability, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_INV_CRT.
Cap_Captbl	cid_t	C	A capability number that corresponds to a capability table capability that possesses the RME_CAPTBL_FLAG_CRT flag. The capability table that this capability points to will accept the newly created invocation capability. This capability number can have either a master encoding or a expanded encoding.
Cap_Kmem	cid_t	P1.D1	A kernel memory capability, whose kernel memory range must be big enough to contain the whole invocation port object. This capability must have the RME_KMEM_FLAG_INV flag. This capability number can have either a master encoding or a expanded encoding.
Cap_Inv	cid_t	P1.D0	A capability number that corresponds to a capability slot that will accept the newly created invocation capability. This slot must be empty. This capability number can only have a master encoding.
Cap_Proc	cid_t	P2	A capability number that corresponds to a process capability that possesses the RME_PROC_FLAG_INV flag. The process that this capability points to will contain the new invocation port. This capability number can have either a master encoding or a expanded encoding.
Vaddr	ptr_t	P3	The kernel virtual address that this new invocation port will use.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Captbl exceeded range.
	The lookup of Cap_Kmem exceeded range.

	The lookup of Cap_Proc exceeded range.
	The lookup of Cap_Inv exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Captbl encountered a frozen capability.
	The lookup of Cap_Kmem encountered a frozen capability.
	The lookup of Cap_Proc encountered a frozen capability.
	Cap_Inv is frozen, or other CPUs are creating capabilities there.
RME_ERR_CAP_TYPE	Cap_Captbl is not a capability table capability.
	Cap_Kmem is not a kernel memory capability.
	Cap_Proc is not a process capability.
RME_ERR_CAP_FLAG	Cap_Captbl doesn't have RME_CAPTBL_FLAG_CRT flag.
	Cap_Kmem doesn't have RME_KMEM_FLAG_INV flag, or does not cover the range designated by Vaddr.
	Cap_Proc doesn't have RME_PROC_FLAG_INV flag.
RME_ERR_CAP_EXIST	Cap_Inv is not an empty capability slot.
RME_ERR_CAP_KOTBL	Kernel memory allocation failed.

5.4.2 Synchronous Invocation Deletion

This operation will delete an invocation port. The invocation port deleted must be idle. To delete an invocation capability, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_INV_DEL.
Cap_Captbl	cid_t	C	A capability number that corresponds to a capability table capability that possesses the RME_CAPTBL_FLAG_DEL flag. The capability table that this capability points to contains the capability to the invocation port being deleted. This capability number can have either a master encoding or a expanded encoding.
Cap_Inv	cid_t	P1	A capability number that corresponds to the capability to the invocation port being deleted. The capability must be a invocation capability. This capability number can only have a master encoding.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Captbl exceeded range.
	The lookup of Cap_Inv exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Captbl encountered a frozen capability.
	Cap_Inv is not frozen.
RME_ERR_CAP_TYPE	Cap_Captbl is not a capability table capability.
	Cap_Inv is not a synchronous invocation capability.
RME_ERR_CAP_NULL	Cap_Inv is an empty capability.
	Two CPUs try to delete the same invocation capability, and this value will be returned on the failed CPU.
RME_ERR_CAP_FLAG	Cap_Captbl doesn't have RME_CAPTBL_FLAG_DEL flag.

RME_ERR_CAP_QUIE	Cap_Inv is not quiescent.
RME_ERR_CAP_REFCNT	The reference count of Cap_Inv is non-zero, or it is not a root capability.
RME_ERR_SIV_ACT	This synchronous invocation port is still active.

5.4.3 Setting Synchronous Invocation Execution Property

This operation will set the execution properties of a synchronous invocation port, that is, its entry and stack. When we are setting the values, we do not care whether the synchronous invocation port is active or not. To set the execution properties of an invocation, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_INV_SET.
Cap_Inv	cid_t	P1.D0	A capability number that corresponds to an invocation capability that possesses the RME_INV_FLAG_SET flag. The invocation port that this capability points to will have its execution property set. This capability number can have either a master encoding or a expanded encoding.
Entry	ptr_t	P2	The entry of the invocation. This is a virtual address in the process, and the invocation will start its execution from here.
Stack	ptr_t	P3	The stack of the invocation. This is a virtual address in the process, and the invocation's initial stack pointer will be here. Whether it is a ascending stack or descending stack is decided by the user-level library.
Fault_Ret_Flag	ptr_t	P1.D1	If a non-zero value is passed in, once an error occurs in the synchronous invocation, the invocation will directly return and error handling is not allowed. If this is zero, the thread will enter RME_THD_FAULT state and wait for error handling.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Inv exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Inv encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Inv is not a synchronous invocation capability, or is an empty capability.
RME_ERR_CAP_FLAG	Cap_Inv doesn't have RME_INV_FLAG_SET flag.

5.4.4 Synchronous Invocation Activation

This operation will make an invocation call. The designated invocation port must be idle. To invoke an invocation capability, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_INV_ACT.
Cap_Inv	cid_t	P1	A capability number that corresponds to an invocation capability that possesses the RME_INV_FLAG_ACT flag. The invocation port that this capability points to will be activated. This capability number can

			have either a master encoding or a expanded encoding.
Param	ptr_t	P2	The parameter to pass to the invocation.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Inv exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Inv encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Inv is not a synchronous invocation capability, or is an empty capability.
RME_ERR_CAP_FLAG	Cap_Inv doesn't have RME_INV_FLAG_ACT flag.
RME_ERR_SIV_ACT	Cap_Inv is already activated (by some other thread).
	Two CPUs try to activate the same invocaton port, and this value will be returned on the failed CPU.

5.4.5 Returning from Synchronous Invocation

This operation will return from an invocation. This is a special operation; it does not need parameters other than a system call number and a invocation return value. If there are multiple invocations nesting each other, then the system call will return to the last invocation. If this is called when not in an invocation, an error will be returned, indicating the failure of return. To return from an invocation, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_INV_RET.

The possible return values of this operation are:

Return	Explanation
0	Operation successful. The invocation will return immediately and this value will not be received by the caller.
RME_ERR_SIV_EMPTY	A return is attempted when no synchronous invocation is executing.

5.5 Asynchronous Communication System Calls

The system calls that are related to the signal endpoint are listed as follows:

System call number	Type	Explanation
RME_SVC_SIG_CRT	System call	Create a signal endpoint
RME_SVC_SIG_DEL	System call	Delete a signal endpoint
RME_SVC_SIG_SND	System call	Send to a signal endpoint
RME_SVC_SIG_RCV	System call	Receive from a signal endpoint

The operation flags of signal endpoint capabilities are listed as follows:

Flag	Bit	Explanation
RME_SIG_FLAG_SND	[0]	Allows sending to the signal endpoint.
RME_SIG_FLAG_RCV	[1]	Allows receiving from the signal endpoint.

5.5.1 Signal Endpoint Creation

This operation will create a signal endpoint, and place its capability into a capability table. To create a signal endpoint, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_INV_CRT.
Cap_Captbl	cid_t	C	A capability number that corresponds to a capability table capability that possesses the RME_CAPTBL_FLAG_CRT flag. The capability table that this capability points to will accept the newly created signal endpoint capability. This capability number can have either a master encoding or a expanded encoding.
Cap_Kmem	cid_t	P1	A kernel memory capability, whose kernel memory range must be big enough to contain the whole signal endpoint object. This capability must have the RME_KMEM_FLAG_SIG flag.This capability number can have either a master encoding or a expanded encoding.
Cap_Sig	cid_t	P2	A capability number that corresponds to a capability slot that will accept the newly created signal endpoint capability. This slot must be empty. This capability number can only have a master encoding.
Vaddr	ptr_t	P3	The kernel virtual address that this new signal endpoint will use.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Captbl exceeded range.
	The lookup of Cap_Kmem exceeded range.
	The lookup of Cap_Sig exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Captbl encountered a frozen capability.
	The lookup of Cap_Kmem encountered a frozen capability.
	Cap_Sig is frozen, or other CPUs are creating capabilities there.
RME_ERR_CAP_TYPE	Cap_Captbl is not a capability table capability.
	Cap_Kmem is not a kernel memory capability.
RME_ERR_CAP_FLAG	Cap_Captbl doesn’t have RME_CAPTBL_FLAG_CRT flag.
	Cap_Kmem doesn't have RME_KMEM_FLAG_SIG flag, or does not cover the range designated by Vaddr.
RME_ERR_CAP_EXIST	Cap_Sig is not an empty capability slot.
RME_ERR_CAP_KOTBL	Kernel memory allocation failed.

5.5.2 Signal Endpoint Deletion

This operation will delete a signal endpoint. The signal endpoint deleted must be idle (no thread is waiting on it). To delete a signal endpoint, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_SIG_DEL.
Cap_Captbl	cid_t	C	A capability number that corresponds to a capability table capability that possesses the RME_CAPTBL_FLAG_DEL flag. The capability table that this capability points to contains the capability to the signal endpoint being deleted. This capability number can have either a master

			encoding or a expanded encoding.
Cap_Sig	cid_t	P1	A capability number that corresponds to the capability to the signal endpoint being deleted. The capability must be a signal endpoint capability. This capability number can only have a master encoding.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Captbl exceeded range.
	The lookup of Cap_Sig exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Captbl encountered a frozen capability.
	Cap_Sig is not frozen.
RME_ERR_CAP_TYPE	Cap_Captbl is not a capability table capability.
	Cap_Sig is not a signal endpoint capability.
RME_ERR_CAP_NULL	Cap_Sig is an empty capability.
	Two CPUs try to delete the same signal endpoint, and this value will be returned on the failed CPU.
RME_ERR_CAP_FLAG	Cap_Captbl doesn't have RME_CAPTBL_FLAG_DEL flag.
RME_ERR_CAP_QUIE	Cap_Sig is not quiescent.
RME_ERR_CAP_REFCNT	The reference count of Cap_Sig is non-zero, or it is not a root capability.
RME_ERR_SIV_ACT	The signal endpoint is still active.
RME_ERR_SIV_CONFLICT	The signal endpoint is a kernel endpoint and cannot be deleted. See the next chapter for details.

5.5.3 Sending to Signal Endpoint

This operation will send a signal to a signal endpoint. To send to a signal endpoint, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_SIG_SND.
Cap_Sig	cid_t	P1	A capability number that corresponds to a signal endpoint capability that possesses the RME_SIG_FLAG_SND flag. We will send a signal to the signal endpoint that this capability points to. This capability number can have either a master encoding or a expanded encoding.

The possible return values of this operation are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The lookup of Cap_Sig exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Sig encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Sig is not a signal endpoint capability, or is an empty capability.
RME_ERR_CAP_FLAG	Cap_Sig doesn't have RME_SIG_FLAG_SND flag.
RME_ERR_SIV_FULL	The signal count of the signal is already full, cannot send to it anymore. This is very rare because in 32-bit systems the maximum count allowed is 2^{32}-1; in 64-bit systems this number is 2^{64}-1, and the

	same rule goes for systems with higher bits.

5.5.4 Receiving from Signal Endpoint

This operation will attempt to receive from a signal endpoint. If there is no signal on that endpoint, the thread will block until there is one. To receive from a signal endpoint, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_SIG_RCV.
Cap_Sig	cid_t	P1	A capability number that corresponds to a signal endpoint capability that possesses the RME_SIG_FLAG_RCV flag. We will block on the signal endpoint that this capability points to.This capability number can have either a master encoding or a expanded encoding.

The possible return values of this operation are:

Return	Explanation
Non-negative	Operation successful. The number returned is the number of signals left after this receive call.
RME_ERR_CAP_RANGE	The lookup of Cap_Sig exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Sig encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Sig is not a signal endpoint capability, or is an empty capability.
RME_ERR_CAP_FLAG	Cap_Sig doesn't have RME_SIG_FLAG_RCV flag.
RME_ERR_SIV_BOOT	Attempt to make Init thread wait on some endpoint.
RME_ERR_SIV_ACT	One thread already blocked at the endpoint.
RME_ERR_SIV_CONFLICT	Two cores attempted to receive on the same endpoint and retry is needed.

Bibliography

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

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

Chapter 6 Kernel Function and Kernel Asynchronous Signal

6.1 Introduction to Kernel Function

Due to the fact that RME's generic functionality can only cover the common portions of code on multiple architectures, it lacked architecture-specific support. If there is some functionality that can only be supported at kernel-level and is architecture specific, it can be supported with RME's kernel function call mechanism.

The kernel function call mechanism introduced the kernel function capability, which allows the user to call a predefined kernel function, and the function will run at user-level. This capability will be created in the system booting sequence, and cannot be deleted. The high resolution timers, inter-processor interrupts and low-power modes can be made use of through kernel function calls. The user is not forced to implement any of these functionality if the processor does not have these features, to facilitate maximum flexibility.

Every kernel function has a distinct kernel function number. The number needs to be passed in when making kernel function calls. Please refer to the next chapter for detailed implementation of kernel function call mechanisms.

6.2 Introduction to Kernel Asynchronous Signal

In RME, the interrupt handlers are registered at user-level, thus some mechanisms are needed to route these signals to the user-level. RME employs kernel asynchronous endpoints for interrupt routing. The only difference between kernel endpoint and regular endpoint is that they are created at boot-time, and they are undeletable during system runtime. To receive signals from a kernel asynchronous endpoint, we just need to use asynchronous receive system call on that endpoint. Due to the same reason, RME's timers are implemented at user-level by passing timer interrupt signals to the user-level.

6.3 Kernel Function System Calls

The system calls that are related to the kernel function are listed as follows:

System call number	Type	Explanation
RME_SVC_KERN	System call	Call a kernel function

The operation flags of kernel function capabilities are listed as follows:

Flag	Bit	Explanation
All bits	Bitfield	The kernel function call number range. The kernel functions in this range are allowed to be called from this kernel function capability. This number is not to be confused with system call numbers. When the flag field of the kernel function capabilities are needed when making delegations, macro RME_KERN_FLAG(HIGH,LOW) is used to fill it. HIGH is the upper limit of the function number, LOW is the lower limit of the function number, and the closed interval [HIGH, LOW] contains the range allowed.

6.3.1 Initialization of Kernel Function

See the next chapter for details on initial creation of kernel function call capabilities.

6.3.2 Activating Kernel Function

This operation will execute a kernel function. A subfunction ID and two extra parameters can be passed in. To activate a kernel function, the following parameters are needed:

Parameter	Type	Pos.	Description
Svc_Num	ptr_t	N	Must be RME_SVC_KERN.
Cap_Kern	cid_t	C	A capability number that corresponds to a kernel function capability. This capability number can have either a master encoding or a expanded encoding.
Func_ID	ptr_t	P1.D0	The kernel function number.
Sub_ID	ptr_t	P1.D1	The subfunction number.
Param1	ptr_t	P2	The first parameter passed in.
Param2	ptr_t	P3	The second parameter passed in.

The possible return values of this operation are:

Return	Explanation
Non-zero	Operation successful. The meaning of the return value is decided by architecture-specific implementations.
RME_ERR_CAP_RANGE	The lookup of Cap_Kern exceeded range.
RME_ERR_CAP_FROZEN	The lookup of Cap_Kern encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Kern is not a kernel function capability, or is an empty capability.
RME_ERR_CAP_FLAG	The function number range of Cap_Kern does not include Func_ID.

6.4 Kernel Endpoint System Calls

6.4.1 Initialization of Kernel Endpoint

See the next chapter for initial creation of kernel endpoints.

6.4.2 Receiving from Kernel Endpoint

Receiving kernel signals uses the same system call that is used to receive normal signals. See the previous chapter for details.

Bibliography

None

Chapter 7 Porting RME to New Architectures

7.1 Introduction

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 Porting Checklist

7.2.1 Processor

RME 要求所选择的处理器能够容纳一个完整的全功能操作系统.处理器必须具备一定程度的处理能力，以克服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 Compiler

RME 要求编译器是 C89 标准的，并能够根据一定的函数调用约定生成代码.由于 RME 的代码非常标准，也不使用 C 运行时库中的库函数，因此只要编译器符合 ANSI C89 标准即可.通常的 gcc、clang/llvm、msvc、armcc、icc、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 Assembler

RME 要求汇编器能够引入 C 中的符号，并根据函数调用约定进行调用；此外，也要求汇编器产生的代码能够导出并根据函数调用约定被 C 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 Linker

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 Debugger

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

7.3 Introduction to Architecture Related Portion of RME

RME 的架构相关部分代码的源文件全部都放在 Platform 文件夹的对应架构名称下.如 Cortex-M 架构的文件夹名称为 Platform/CortexM.其对应的头文件在 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 Type Definitions

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 Macros

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 Architecture Related Structures

RME 的架构相关结构体一共有三个，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。
RME_Iret_Struct	与程序执行流相关的，在线程迁移调用中要保存和恢复的寄存器的结构体。

这三个结构体的实现和系统中断向量进入段汇编函数的实现有关。

7.3.4 Low-level Assembly Functions

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

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 System Interrupt Vectors

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

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

这些中断向量的具体实现方法和实现次序将在后面章节加以讲解.

7.3.6 Other Low-level Functions

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

7.3.6.1 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_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_Reg_Save	保存线程迁移调用返回用的必要寄存器。
__RME_Inv_Reg_Restore	恢复线程迁移调用返回用的必要寄存器。

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 Porting of Type Definitions, Macros and Low-level Assembly Functions

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

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

为了使得底层函数的编写更加方便，推荐使用如下的几个 typedef 来定义经常使用到的确定位数的整形.在定义这些整形时，也需要确定编译器的 char、short、int、long 等究竟是多少个机器字的长度.有些编译器不提供六十四位或者一百二十八位整数，那么这几个类型可以略去.

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 Implementation of __RME_Disable_Int

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 Implementation of __RME_Enable_Int

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 Implementation of _RME_Kmain

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 Implementation of __RME_Enter_User_Mode

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 Porting of System Interrupt Vectors

系统中断向量的移植的主要工作包括两部分：一部分是进入中断向量和退出中断向量的汇编代码，另一部分是系统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 Entry & Exit of Interrupt Vectors and Architecture Related Structures

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 Fault Handling Interrupt Vectors

在该向量中，需要调用一个系统错误处理函数.该函数的描述见上节所述.该函数的实现是与架构紧密相关的，因此需要在移植时重新设计.该函数首先需要判断发生的错误是可恢复错误还是不可恢复错误.如果发生的是不可恢复错误（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 Porting of the Kernel Debug Print Function

内核调试打印函数的底层接口只有一个函数，如下：

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 Porting of Atomic Operations and Processor Specific Operations

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 Compare-and-Swap

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 Fetch-and-Add

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 Fetch-and-And

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 Get the MSB Position of a Word

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 Porting of Initialization, Startup and CPUID Functions

初始化、启动与 CPUID 函数一共有四个，如下所示.

7.8.1 Implementation of __RME_Low_Level_Init

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 Implementation of __RME_Boot

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.This capability number can only have a master encoding.
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 exceeded range.
RME_ERR_CAP_KOTBL	Kernel memory allocation failed.

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.This capability number can have either a master encoding or a expanded encoding.
Cap_Captbl	cid_t	要接受产生的权能表权能的位置.This capability number can only have a master encoding.
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 exceeded 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 exceeded range.
RME_ERR_CAP_FROZEN（不太可能返回该值）	Cap_Captbl_Crt encountered a frozen capability.
	Cap_Crt is frozen, or other CPUs are creating capabilities there.
RME_ERR_CAP_TYPE	Cap_Captbl_Crt is not a capability table capability.
RME_ERR_CAP_FLAG	Cap_Captbl_Crt 无 RME_CAPTBL_FLAG_CRT 属性.
RME_ERR_CAP_EXIST	Cap_Crt is not an empty capability slot.
RME_ERR_CAP_KOTBL	Kernel memory allocation failed.

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.This capability number can have either a master encoding or a expanded encoding.
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.This capability number can only have a master encoding.
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 exceeded range.
	Cap_Pgtbl exceeded range.
RME_ERR_CAP_FROZEN（不太可能返回该值）	Cap_Captbl encountered a frozen capability.
	Cap_Pgtbl is frozen, or other CPUs are creating capabilities there.
RME_ERR_CAP_TYPE	Cap_Captbl is not a capability table capability.
RME_ERR_CAP_FLAG	Cap_Captbl 无 RME_CAPTBL_FLAG_CRT 属性.
RME_ERR_CAP_EXIST	Cap_Pgtbl is not an empty capability slot.
RME_ERR_CAP_KOTBL	Kernel memory allocation failed.
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.This capability number can have either a master encoding or a expanded encoding.
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.This capability number can have either a master encoding or a expanded encoding.
Flags_Child	ptr_t	子页目录被映射时的属性。这个属性限制了该映射以下的所有页目录的访问权限。对于不同的架构，这个位置的值的意义也不相同。对于有些不支持页目录属性的架构而言（比如所有的基于 MPU 的系统），这个值无效。

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 or Cap_Pgtbl_Child exceeded range.
RME_ERR_CAP_FROZEN（不太可能返回该值）	Cap_Pgtbl_Parent or Cap_Pgtbl_Child encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Pgtbl_Parent or Cap_Pgtbl_Child is not a page directory capability.
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 exceeded range.
RME_ERR_CAP_FROZEN（不太可能返回该值）	Cap_Pgtbl encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Pgtbl is not a page directory capability.
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.This capability number can have either a master encoding or a expanded encoding.
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.This capability number can only have a master encoding.
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.This capability number can have either a master encoding or a expanded encoding.
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）.This capability number can have either a master encoding or a expanded encoding.
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 exceeded range.
	Cap_Captbl exceeded range.
	Cap_Pgtbl exceeded range.
	Cap_Proc exceeded range.
RME_ERR_CAP_FROZEN（不太可能返回该值）	Cap_Captbl_Crt encountered a frozen capability.
	Cap_Captbl encountered a frozen capability.
	Cap_Pgtbl encountered a frozen capability.
	Cap_Proc is frozen, or other CPUs are creating capabilities there.
RME_ERR_CAP_TYPE	Cap_Captbl_Crt or Cap_Captbl is not a capability table capability.
	Cap_Pgtbl is not a page directory capability.
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 is not an empty capability slot.
RME_ERR_CAP_KOTBL	Kernel memory allocation failed.
RME_ERR_CAP_REFCNT	Cap_Captbl or 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.This capability number can have either a master encoding or a expanded encoding.
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.This capability number can only have a master encoding.

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 exceeded range.
	Cap_Kern exceeded range.
RME_ERR_CAP_FROZEN （不太可能返回该值）	Cap_Captbl encountered a frozen capability.
	Cap_Kern is frozen, or other CPUs are creating capabilities there.
RME_ERR_CAP_TYPE	Cap_Captbl is not a capability table capability.
RME_ERR_CAP_FLAG	Cap_Captbl 无 RME_CAPTBL_FLAG_CRT 属性.
RME_ERR_CAP_EXIST	Cap_Kern is not an empty capability slot.

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.This capability number can have either a master encoding or a expanded encoding.
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.This capability number can only have a master encoding.
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 exceeded range.
	Cap_Kmem exceeded range.
RME_ERR_CAP_FROZEN（不太可能返回该值）	Cap_Captbl encountered a frozen capability.
	Cap_Kmem is frozen, or other CPUs are creating capabilities there.
RME_ERR_CAP_TYPE	Cap_Captbl is not a capability table capability.
RME_ERR_CAP_FLAG	Cap_Captbl 无 RME_CAPTBL_FLAG_CRT 属性.
RME_ERR_CAP_EXIST	Cap_Kmem is not an empty capability slot.

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.This capability number can have either a master encoding or a expanded encoding.

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.This capability number can only have a master encoding.
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 exceeded range.
	Cap_Sig exceeded range.
RME_ERR_CAP_FROZEN（不太可能返回该值）	Cap_Captbl encountered a frozen capability.
	Cap_Sig is frozen, or other CPUs are creating capabilities there.
RME_ERR_CAP_TYPE	Cap_Captbl is not a capability table capability.
RME_ERR_CAP_FLAG	Cap_Captbl 无 RME_CAPTBL_FLAG_CRT 属性.
RME_ERR_CAP_EXIST	Cap_Sig is not an empty capability slot.
RME_ERR_CAP_KOTBL	Kernel memory allocation failed.

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.This capability number can have either a master encoding or a expanded

		encoding.
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.This capability number can only have a master encoding.
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.This capability number can have either a master encoding or a expanded encoding.
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 exceeded range.
	Cap_Proc exceeded range.
	Cap_Thd exceeded range.
RME_ERR_CAP_FROZEN（不太可能返回该值）	Cap_Captbl encountered a frozen capability.
	Cap_Proc encountered a frozen capability.
	Cap_Thd is frozen, or other CPUs are creating capabilities there.
RME_ERR_CAP_TYPE	Cap_Captbl is not a capability table capability.
	Cap_Proc is not a process capability.
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 is not an empty capability slot.
RME_ERR_CAP_KOTBL	Kernel memory allocation failed.
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 Implementation of __RME_Reboot

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 Implementation of __RME_Shutdown

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 Implementation of __RME_CPUID_Get

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 Porting of Register Set Related Functions

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 Implementation of __RME_Get_Syscall_Param

该函数用于从寄存器组中提取系统调用的参数。

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

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 Implementation of __RME_Set_Syscall_Retval

该函数用于向寄存器组中存入系统调用的返回值。

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

7.9.4 Implementation of __RME_Set_Inv_Retval

该函数用于向寄存器组中存入线程迁移调用的返回值。

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

7.9.5 Implementation of __RME_Thd_Reg_Init

该函数用于初始化线程的寄存器组。

函数原型	void __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	无。
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 Implementation of __RME_Thd_Reg_Copy

该函数用于复制线程的寄存器组。有时候需要用汇编实现这个函数以提高效率。

函数原型	void __RME_Thd_Reg_Copy(struct RME_Reg_Struct* Dst,

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

7.9.7 Implementation of __RME_Thd_Cop_Init

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	void __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	无。
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 Implementation of __RME_Thd_Cop_Save

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

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

7.9.9 Implementation of __RME_Thd_Cop_Restore

该函数用于恢复线程的协处理器寄存器组。

函数原型	void __RME_Thd_Cop_Restore(struct RME_Reg_Struct* Reg, struct RME_Cop_Struct* Cop_Reg)
意义	恢复线程的协处理器寄存器组。
返回值	无。
参数	struct RME_Reg_Struct* Reg 指向寄存器组数据结构的指针。这个参数是用来辅助判断是否需要保存协处理器寄

	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 Implementation of __RME_Inv_Reg_Init

该函数用于初始化线程迁移调用的寄存器组。

函数原型	void __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	无。
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 Implementation of __RME_Inv_Reg_Save

该函数用于保存必要的寄存器到线程迁移调用结构体中以方便返回。只需要保存那些对恢复程序执行流必要的寄存器就可以了。比如，对于 x86-64，要保存 SP 和 IP；对于 Cortex-M，要保存 LR 和 SP。

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

7.9.12 Implementation of __RME_Inv_Reg_Restore

该函数用于从线程迁移调用结构体恢复必要的寄存器以返回。只需要恢复那些对恢复程序执行流必要的寄存器就可以了。比如，对于 x86-64，要恢复 SP 和 IP；对于 Cortex-M，要恢复 LR 和 SP。

函数原型	void __RME_Inv_Reg_Restore(struct RME_Reg_Struct* Reg, struct RME_Iret_Struct* Ret)
意义	从线程迁移调用结构体中恢复必要的寄存器。
返回值	无。
参数	struct RME_Reg_Struct* Reg 该参数用于输出，是指向寄存器组的指针。 struct RME_Iret_Struct* Ret

	指向必要寄存器结构体的指针。

7.10 Porting of Kernel Functions

内核功能调用函数是一组由用户实现的、可以在操作系统内核态运行的一系列函数.这些函数由用户定义.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 Tickless Kernel Implementation

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

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

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

7.10.2 High Precision Timer System Implementation

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

7.10.3 Inter-Processor-Interrupt Implementation

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

7.10.4 Cache MaintenanceOperation Implementation

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

7.11 Porting of Page Table Related Functions

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

7.11.1 Implementation of __RME_Pgtbl_Set

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

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 Implementation of __RME_Pgtbl_Kmem_Init

该函数负责在系统启动时建立初始的内核页表.

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 Implementation of __RME_Pgtbl_Check

该函数负责检查用来创建页目录的各个参数是否能够被底层架构支持.

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 Implementation of __RME_Pgtbl_Init

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 Implementation of __RME_Pgtbl_Del_Check

该函数负责检查一个页目录能否被安全删除.

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 Implementation of __RME_Pgtbl_Page_Map

该函数负责映射一个页到一个页目录内.如果这种映射由于传入的参数不正确（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 Implementation of __RME_Pgtbl_Page_Unmap

该函数负责解除页目录内一个页的映射.如果该操作由于传入的参数不正确（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 Implementation of __RME_Pgtbl_Pgdir_Map

该函数负责映射一个子页目录到父页目录内。如果该操作由于传入的参数不正确（比如位号超标、物理地址对齐不符合要求、虚拟地址的关系不正确、该位置已经有映射、传入的标志位不正确或者在 MPU 环境下某些特殊约束不满足），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, ptr_t Flags)
意义	映射一个子页目录到父页目录内部。
返回值	ptr_t 成功返回 0，失败返回 RME_ERR_PGT_OPFAIL（-1）。
参数	struct RME_Cap_Pgtbl* Pgtbl_Parent 指向父页目录的，含有父页目录的所有信息的页目录权能。
	ptr_t Pos 需要将该子页目录映射到的父页目录表项位号。
	struct RME_Cap_Pgtbl* Pgtbl_Child 指向子页目录的，含有子页目录的所有信息的页目录权能。
	ptr_t Flags 该子页目录的 RME 标准页标志。该页标志决定了子页目录及以下各层页目录的访问权限限制。对于那些不允许设置页目录属性的架构或者基于 MPU 的架构，该值无效。

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

在 MPU 系统下，由于 MPU 的某些固有属性（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 Implementation of __RME_Pgtbl_Pgdir_Unmap

该函数负责解除父页目录内一个子页目录的映射.如果该操作由于传入的参数不正确（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 Implementation of __RME_Pgtbl_Lookup

该函数负责查找一个页目录内的某个页的信息.

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 Implementation of __RME_Pgtbl_Walk

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 Writing Interrupt Vectors

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 Entering and Exiting of Interrupt Vectors

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

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 Callable Kernel Functions in Interrupt Vectors

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

7.12.2.1 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 Explanations for Other Functions

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

7.13.1 Variable Clearing

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

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 Comparing Memory Segments

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

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 Replicating Memory Segments

该函数用来复制一段内存的内容到另一区域.该函数实质上等价于 C 语言运行时库的 memcpy.两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 Printing Signed Integers

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

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 Printing Unsigned Integers

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

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 Printing Strings

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

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.

Bibliography

无

Chapter 8 Appendix

8.1 RME Regular Coding Style

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 Organization Guideline for RME Sources and Headers

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 C Coding Style Conventions

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 Indentation Conventions

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 Supporting Special Functionality in RME

RME 可以支持诸多某些其他操作系统提供的特殊功能，诸如 CPU 热插拔、内存热插拔、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 Hotplug

CPU 热插拔分为两个功能，一个是热插，也即插入新的 CPU，增加 CPU 的数量；另外一个是热拔，也即从插槽上拔出 CPU，减少 CPU 的数量.RME 对该功能的支持依赖于底层硬件平台提供的硬件级别支持.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 Memory Hotplug

内存热插拔也分为两部分，一部分是增加内存，另一部分是减少内存.RME 对该功能的支持依赖于底层硬件平台提供的硬件级别支持.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.

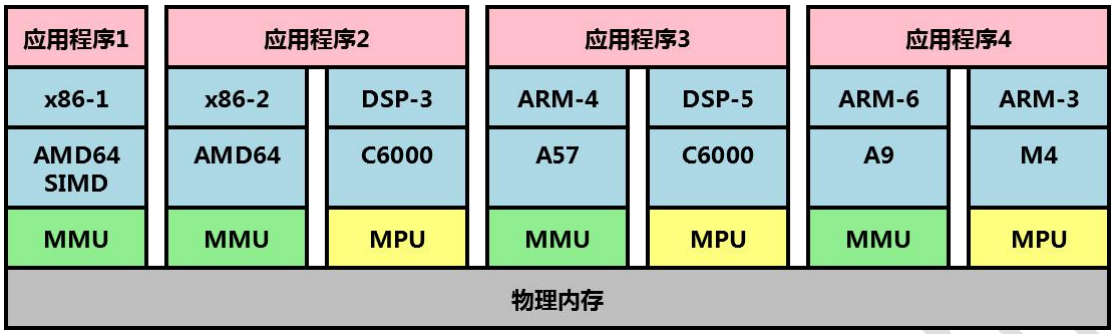
对于减少内存的情况，如果是减少用户态内存，需要用户态库确定现在这些物理页面没有被映射.此时，可以直接拔出该内存条.如果是减少内核内存，那么需要首先暂停其他处理器的运行，然后将这些将消失的内存中的页面上的数据拷贝到空白的物理页面上，并把原来在内核页表中映射的条目替换成这些空白物理页面，最后再恢复其他处理器的运行.这些空白的物理页面要由用户态库保证不存在除了根页表之外的页表中.在确定这些物理页面由内核使用后，要把这些页面从根页表中也删除，使其不会从用户态被访问.在完成所有操作后也要做一个 TLB 刷新操作.

8.2.3 Separation Kernel

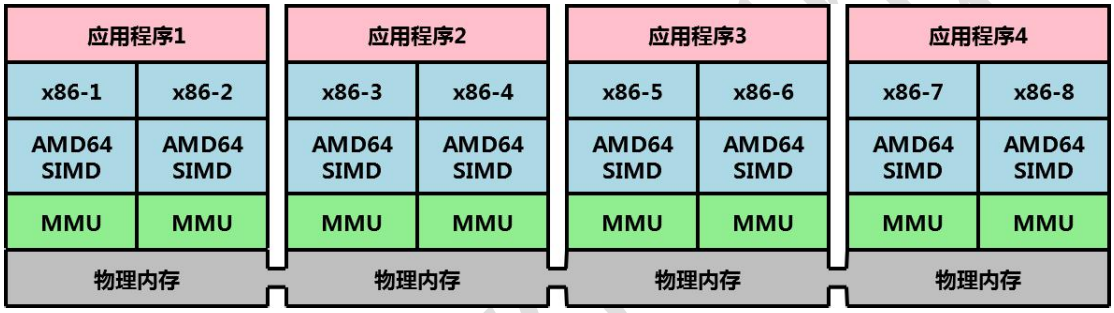
由于 RME 是一个微内核操作系统，因此，就像 Barrelfish 那样[1]，可以很方便地在一台物理机器上运行 RME 的多个实例.每个实例可以管理一个或多个 CPU 核，然后在用户态通过多个操作系统共享内存或者网络实现通信即可.这种方式不要求 CPU 间的缓存是同步的，也不要求

各个 CPU 含有的功能是一样的，甚至不要求各个 CPU 的指令集和架构是一样的.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 Afterwords

8.3.1 Non-scalable Portions of RME in Multi-core Environments

在 RME 中，并非所有的操作都可以互不影响地执行.典型的不能这样执行的操作是多个 CPU 试图同时向一块内存中创建内核对象.下面列出不能并行执行的操作，并且说明其原因.

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 Restrictions of RME on 32-bit Processors

RME 在 32 位处理器的情况下，受限于处理器字长长度，对某些功能的实现有所制约.被制约的两个功能是线程创建和安定时间的计算.

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 is not quiescent 的状态.此时该位置早已安定，这种 is not quiescent 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 is not quiescent，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 Known Potential Covert Channels in RME

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.

Bibliography

[1] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, et al., "The multikernel: a new OS architecture for scalable multicore systems," in Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009, pp. 29-44.