突越・中生・真核（七阶）

Mutatus・Mesozoa・Eukaron(R.VII)

M7M1(Eukaron) R3T2

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

Technical 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 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 as asynchronous signals
- Synchronous invocations 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 are growing quickly, and the trend of asymmetric computation is also gaining popularity. In the meantime, as the amount of resource bloats, the need for advanced memory management is also increasing. 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 real-time operating system 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 protection 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 pose 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 newer generation of RTOSes are supposed to be stronger than current RTOSes in reliability, portability and flexibility; they should also 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. 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 makes 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 boosts the performance under multi-socket non-unified memory access (NUMA) architectures.

We will review the categories of operating systems first. In this manual, we will 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.)

None 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 port
RME_SVC_INV_ACT	1	Activate an invocation port
RME_SVC_SIG_SND	2	Send to a signal endpoint
RME_SVC_SIG_RCV	3	Receive from a signal endpoint
RME_SVC_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.
1.4.3.4 In this manual, all typedef'ed type's prefix "rme_" is omitted to save table space.

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 frequently. This is equivalent to a lower memory bandwidth, and will hamper capability operation performance. The second solution has less cache issues but are more challenging to implement. In RME, we use the latter method to achieve multi-core scalability.

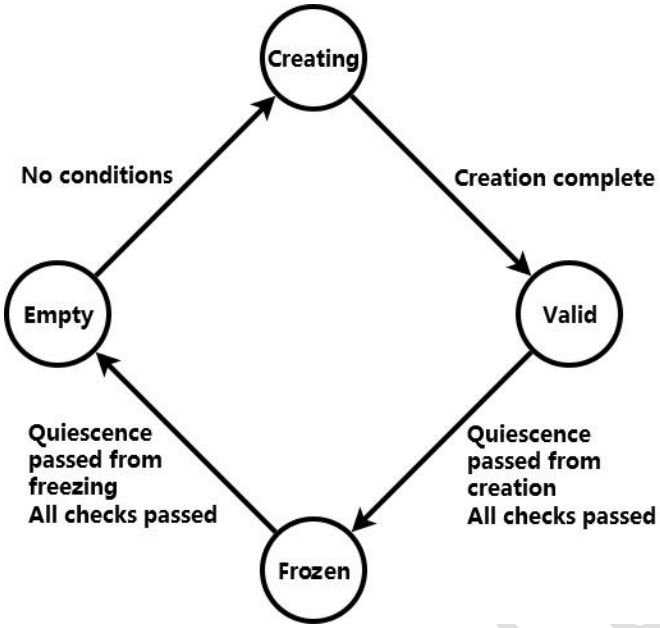
RME includes a large number of compare-and-swap (CAS) atomics and fetch-and-add (FAA) atomics. The kernel also employs a timestamp counter to ensure that no kernel operations conflict, i.e. one capability under use on one CPU gets deleted on another CPU.

To delete or remove a capability, we must ensure that the capability have been frozen for a time period. This period is called capability quiescence period. The length of this period can be configured, and must be longer than twice the kernel WCET. Thus, we can ensure that when we delete or remove the capability, the capability have been stabilized. In other words, all kernel operations 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.

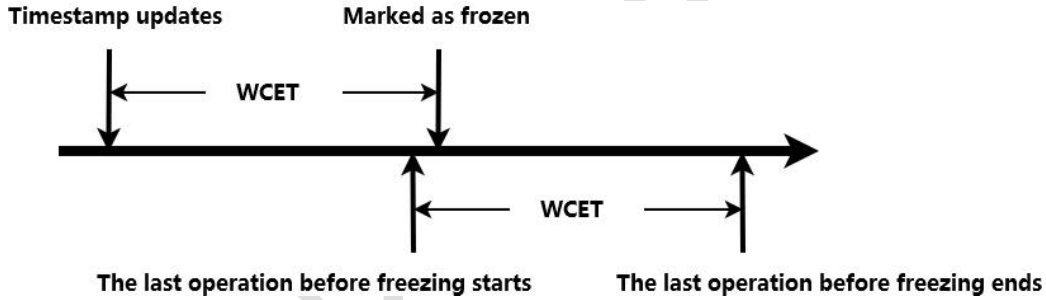
A complete capability state transition diagram is shown below. When the creation starts, the CPU will perform a CAS operation in the empty slot, which marks the start of the creation. After the creation is finished, the CPU will mark the slot as usable, and now operations can be started on this capability.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

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.

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

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 one. 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, additional 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 freeze 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_QUIE	Cap_Frz is not quiescent.
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). This TID is designated by the user-level when the thread is binded to a processor. In 32-bit systems, the range of TID is $0\text{-}2^{32-2}$; considering the fact that these systems are usually embedded systems (microcontroller-based or low-end microprocessor-based), even if the user-level requires globally unique TID (don't have to be like this), this range is sufficient. In 64-bit systems this range is $0\text{-}2^{64-2}$, 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. If a signal endpoint is specified when binding the thread, that endpoint will receive a signal as well. To escape from the fault 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. If a signal endpoint is specified when binding the thread, that endpoint will receive a signal as well.

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, a parent thread, and a TID as necessary 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. You can optionally specify an endpoint to receive its scheduler signals as well. It is worth noting that, the TID does not necessarily have to be globally universal; the user-level can decide if this is desired.

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, stack and parameter of the thread. These three 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 the first 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.

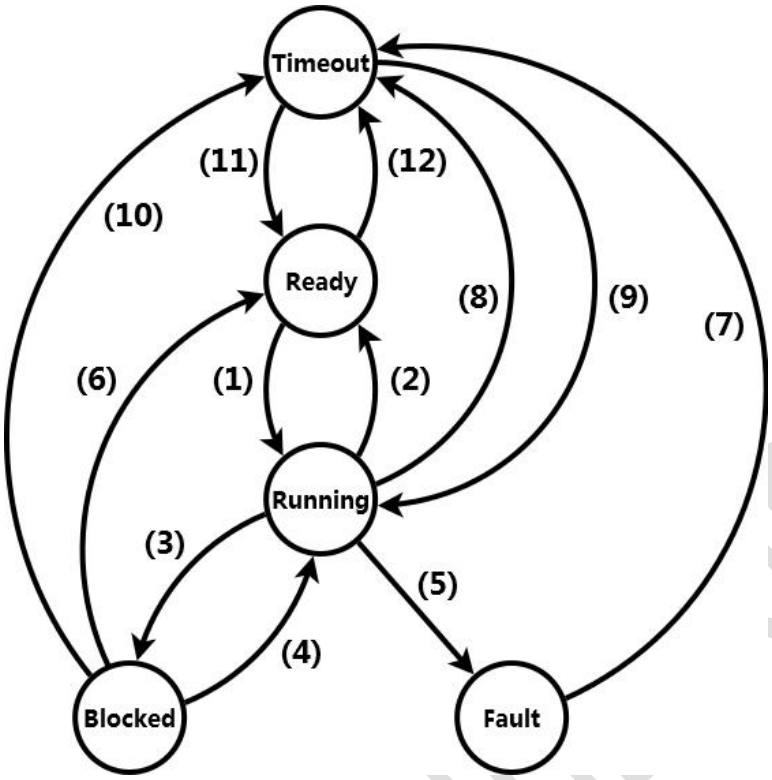
"S" means that the source will definitely timeout.

"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. The newly created thread is not binded to any CPU, does not have a TID, 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 successful, 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, stack and parameter. The thread must be binded to a CPU, and the execution property must be set on this CPU. For a thread in the RME_THD_FAULT state, setting its execution property will turn it into the 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	C	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	P1	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	P2	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.
Param	ptr_t	P3	The parameter to be passed to 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_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	C	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	P1.D1	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.
Cap_Sig	cid_t	P1.D0	A capability number that corresponds to a signal capability that possesses the RME_SIG_FLAG_SCHED flag. When there is a scheduler event (thread timeout or fault), a scheduler event will be sent to this endpoint. This capability number can have either a master encoding or a expanded encoding, and is optional. Pass in RME_CAPID_NULL (a negative value) if you don't want to receive scheduler signals of this thread.
TID	tid_t	P2	The user-supplied TID of the thread. RME will not check whether there are two threads with the same ID in the system, and this decision is up to the user. Additionally, TID shall not exceed 2^{N-2}-1 (N is the processor word length, in bits) or a negative one.
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.
	When Cap_Sig is passed in, it doesn't have RME_SIG_FLAG_SCHED flag.
RME_ERR_PTH_TID	The TID passed in is invalid.
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. If the scheduler 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_CAPID_NULL (a negative number).
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.

To make application programming more flexible, RME allows four different signal receiving options, listed hereinafter:

Receiving option	Explanation
RME_RCV_BS	Try to receive a single signal with blocking method. This may cause the thread to block if there are no signals on the endpoint.

RME_RCV_BM	Try to receive multiple signals with blocking method. This may cause the thread to block if there are no signals on the endpoint, and if there are signals on the endpoint, all of them will be received by this call.
RME_RCV_NS	Try to receive a single signal with non-blocking method. This will return immediately if there are no signals on the endpoint.
RME_RCV_NM	Try to receive multiple signals with non-blocking method. This will return immediately if there are no signals on the endpoint, and if there are signals on the endpoint, all of them will be received by this call.

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_BS	[1]	Allows blocking single-receive from the signal endpoint.
RME_SIG_FLAG_RCV_BM	[2]	Allows blocking multi-receive from the signal endpoint.
RME_SIG_FLAG_RCV_NS	[3]	Allows non-blocking single-receive from the signal endpoint.
RME_SIG_FLAG_RCV_NM	[4]	Allows non-blocking multi-receive from the signal endpoint.
RME_SIG_FLAG_SCHED	[5]	Allows usage as a scheduler signal endpoint when binding threads.

For detailed description about bit [5], please refer to thread related section.

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.
Option	ptr_t	P2	Receiving option. Can be one of the following: RME_RCV_BS: Blocking and try to receive one signal. RME_RCV_BM: Blocking and try to receive multiple signals. RME_RCV_NS: Non-blocking and try to receive one signal. RME_RCV_NM: Non- blocking and try to receive multiple signals.

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 the flag required by the specified option. RME_RCV_BS requires RME_SIG_FLAG_RCV_BS flag. RME_RCV_BM requires RME_SIG_FLAG_RCV_BM flag. RME_RCV_NS requires RME_SIG_FLAG_RCV_NS flag. RME_RCV_NM requires RME_SIG_FLAG_RCV_NM 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.
	The receiving option provided is not correct.
RME_ERR_SIV_CONFLICT	Two cores attempted to receive on the same endpoint and retry is needed.

It is worth noting that, if non-blocking option is specified, and the endpoint currently have no signals, the function will directly return 0, representing the fact that nothing is received. Also, if the option is blocking multi-receive and this receive gets blocked, then this receive will be treated as a single receive when it unblocks.

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 section 7.8.2 for initial creation of kernel endpoints.

6.4.2 Sending to Kernel Endpoint

See section 7.12 for kernel endpoint sending operations.

6.4.3 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

The porting of an operation systems refers to the process of modifying it to run on a new architecture. Sometimes, the adaptation of a system to use a new compiler is called porting as well Compared with Linux, the porting of RME is quite straightforward. All the code of RME is written in MISRA C compliant ANSI/C89 code, and contains minimal assembly. Thus, its porting only involves a few steps.

We need to make some preparations before we actually get down to the porting job. We need to make sure that the target is fit to run RME first, then we need to compose the code for each necessary module; at last, some testbenches are needed to see whether the porting is successful.

7.2 Porting Checklist

7.2.1 Processor

RME requires that the processor of your choice be equipped with the necessary hardware for a full-fledged operating system. Some processing power is also necessary for overcoming the extra overhead of system calls, which makes running a full-fledged OS useful. The processor must have some memory management mechanisms as well, and MMU & MPU are both good.

Ideally, such platforms usually runs at a few GHz, have hundreds of MBs of RAM, and consist of one or more processors that have a datapath width of at least 32-bit. However, RME can also be deployed on 32-bit microcontrollers that have more than 16kB RAM and 64kB ROM.

RME cannot be squeezed in a useful way into any platform that have less than 64kB memory; it cannot run on any processors whose datapath width is less than 32-bit. Additionally, if the processor of your choice does not feature any hardware memory protection mechanisms, running RME is nonsense; in these scenarios, RMP might be a better choice. Lastly, RME does not support any hardware stacking mechanism (which is the implemented in some architectures such as PIC microcontrollers); the stack must be implemented in software (the stack pointer is user-modifiable and the stack is implemented in RAM).

7.2.2 Compiler

RME requires that the compiler strictly adheres to the C89 standard, and generates code according to some function calling convention. Due to the fact that RME's code is very canonical and does not use any C library functions, we only require that the compiler is ANSI C89 compliant. Common compilers such as gcc, clang/llvm, msvc, armcc, icc, ewxxx, tasking satisfy this requirement very well. For maximum compiler compatibility, RME intentionally avoided extensions and implementations that vary greatly between different compilers such as bitfields, enum and struct packing; undefined behaviors of C are not relied upon as well.

When using low-quality compilers, it is recommended to turn off dead code elimination and link-time optimizations, as well as the loop invariant code motion. Some aggressive optimizations may optimize out your interrupt vectors once and for all (because nobody call them in the call graph). To be safe, do not use any aggressive optimization features (especially the experimental ones). On common compilers, the recommended optimization level is (gcc) -O2 or its equivalent optimization level.

7.2.3 Assembler

RME requires that the assembler can import symbols from C, and can call C functions according to some calling convention. It is also required that the code generated by assembler can be called by C with some calling convention. Usually these requirements are readily met. If the compiler allows inline assembly, the assembler requirements can be ignored.

7.2.4 Linker

RME requires that the linker have the ability to link many objects (.o) to a intermediate object file (.o), and that the linker can accept customized linker scripts. Common linkers such as ld, cl, armlink and ilink/xlink can meet these requirements. Different linkers have different script syntax, thus the script for different linkers are usually different.

7.2.5 Debugger

RME does not have any special requirements with regards to debuggers. It would be best if you have one available, however it is also possible to go without. If you have one, it can be used to observe kernel variables; if you go without, you should implement the low-level __RME_Putchar function first to print single characters. After that, it can be used to print log. For detailed implementation about the function, please refer to the next section.

7.3 Introduction to Architecture Related Portion of RME

The architecture-related portion of RME is located at different folders under the Platform folder. Each architecture have a different folder with their name on it. For example, the folder name for Cortex-M is Platform/CortexM, and corresponding headers are located at Include/Platform/CortexM, and the same goes for all architectures.

Each architecture includes one or more source and one or more headers. When the kernel includes the architecture-related headers, it will always include Include/rme_platform.h, and this file will include the corresponding architecture's top-level header. When switching platforms, modify this file to include the desired architecture. For example, to compile the kernel for Cortex-M, it is necessary to include all top-level headers for Cortex-M.

To begin with, you can browse the existing ports to find one that logically looks like your desired architecture most. You can copy the port as a template and start your porting work from there.

7.3.1 Type Definitions

For each architecture/compiler, the first part is to port is the type definitions for RME. There are 5 definitions that are required:

Type	Explanation
tid_t	The type of thread IDs. Should be typedef'ed to be a signed machine word integer. Example: typedef tid_t long;
ptr_t	The type of pointers. Should be typedef'ed to be an unsigned machine word integer. Example: typedef ptr_t unsigned long;
cnt_t	The type of counters. Should be typedef'ed to be a signed machine word integer. Example: typedef cnt_t long;
cid_t	The type of capability numbers. Should be typedef'ed to be a signed machine word integer. Example: typedef cid_t long;

ret_t	The type of return values. Should be typedef'ed to be a signed machine word integer. Example: typedef ret_t long;

7.3.2 Macros

The second part is to port all the macros of RME. All these macros are listed below:

Macro	Explanation
EXTERN	The extern keyword of the compiler. Some compilers does not have a standard extern keyword (or you need to link some C++ code against the kernel), and this can be dealt with with this macro. Example: #define EXTERN extern #define EXTERN extern "C"
INLINE	The inline keyword of the compiler. Some compilers does not support function inlining, in which case leaving it empty will be fine. Example: #define INLINE inline #define INLINE __inline #define INLINE __forceinline
RME_LIKELY(X)	The likely keyword of the compiler, which is a branch prediction hint that indicates this branch is likely taken. If the compiler supports such functionality, define this keyword accordingly; if it doesn't, leave it as (X). Example: #define RME_LIKELY(X) likely(X) #define RME_LIKELY(X) __builtin_expect(!!(X),1) #define RME_LIKELY(X) (X)
RME_UNLIKELY(X)	The unlikely keyword of the compiler, which is a branch prediction hint that indicates this branch is likely not taken. If the compiler supports such functionality, define this keyword accordingly; if it doesn't, leave it as (X). Example: #define RME_UNLIKELY(X) unlikely(X) #define RME_UNLIKELY(X) __builtin_expect(!!(X),0) #define RME_UNLIKELY(X) (X)
RME_CPU_LOCAL()	This macro returns a struct RME_CPU_Local* type pointer to the CPU local storage. For single-core processors, the CPU local storage can be declared directly as a variable in the hardware abstraction layer; for multi-core processors, the CPU local storage areas need to be dynamically allocated according to the number of processors when the system is booting. Example:

	#define RME_CPU_LOCAL() &RME_XXX_CPU_Local (Single-core, static allocation) #define RME_CPU_LOCAL() asm("mov special_register, r0") (Multi-core, dynamic allocation, address in special register)
RME_WORD_ORDER	The order of 2 of the processor's word length (in bits). 32-bit processors correspond to 5 and 64-bit processors correspond to 6, and the same goes for other word length. Example: #define RME_WORD_ORDER 5
RME_VA_EQU_PA	Whether the processor requires that the virtual address is always equal to the physical address. Generally speaking, for MMU-based systems, the answer is "no" (RME_FALSE), and we use regular page tables; for MPU-based systems, the answer is "yes" (RME_TRUE), and we use path-compressed page tables. Example: #define RME_VA_EQU_PA RME_TRUE
RME_QUIE_TIME	The length of the quiescence time, whose unit is timeslices. For single-core systems, due to the fact that there is no real parallelism, the capabilities are immediately quiescent, and this should be defined as zero. For multiprocessor systems, this value should be at least twice as much as kernel Worst-Case Execution Time (WCET), and in practise we recommend 10 times. Generally speaking, one timeslice (100us) is much longer than the kernel's WCET (~10us), thus this macro can be defined as 1 in most cases. Example: #define RME_QUIE_TIME 1
RME_KMEM_VA_START	The kernel virtual address of user-managed kernel memory. When creating kernel objects, the allocation will start here, and the allocations will be registered in kernel object allocation table. Example: #define RME_KMEM_VA_START 0xC0000000
RME_KMEM_SIZE	The size of user-managed kernel memory. Simply define this macro to be the size of the memory. For the cases where kernel memory size must be detected on startup (i.e. x86-64), fill in the minimal memory amount allowed here (if the kernel detected less than this amount of memory available, it will reject the booting request). Example: #define RME_KMEM_SIZE 0x30000000
RME_HYP_VA_START	The start address of the virtual machine monitor dedicated virtual address. This memory range is reserved for virtual machine monitors, and registers can be saved to this address when switching context.

	Example: #define RME_HYP_VA_START 0xF000000
RME_HYP_SIZE	The size of the virtual machine monitor dedicated memory. Just fill in the actual size of the memory. If this functionality is not used, set this macro to 0, and RME_HYP_VA_START will be ineffective at the same time. Example: #define RME_HYP_SIZE 0x10000000
RME_KMEM_SLOT_ORDER	The allocation granularity of kernel memory. The number filled in here is actually the order of 2 of this value. For example, if the minimal granularity is 16 byte, the number you should fill in is $\log_2(16) = 4$. It is worth noting that the granularity cannot be smaller than a machine word. Example: #define RME_KMEM_SLOT_ORDER 4
RME_KMEM_STACK_ADDR	The kernel stack start virtual address. If the stack grows down, this is the top of the stack; if the stack grows up, this is bottom of the stack. Example: #define RME_KMEM_STACK_ADDR 0xF0000000
RME_MAX_PREEMPT_PRIO	The number of preemption priorities that the kernel supports. This number must be a multiple of processor word length (in bits). Usually we just need to define it to be the processor word length. Example: #define RME_MAX_PREEMPT_PRIO 32
RME_PGTBL_SIZE_NOM(X)	The size of normal page directories. This macro takes a parameter whose meaning is the order of 2 of the number of entries in this table. If the page directory includes 1024 entries, the number of X will be 10, and the macro RME_PGTBL_SIZE_NOM(10) will return the size of that directory in bytes. If each entry is 4 bytes, and the appended metadata (which only exists on MPU-based architectures) is 4096 bytes, the macro should return 8192.
RME_PGTBL_SIZE_TOP(X)	The size of top-level page directories. This macro takes a parameter whose meaning is the order of 2 of the number of entries in this table. This macro in fact equals RME_PGTBL_SIZE_NOM(X) plus the size of the extra top-level metadata (which only exists on MPU-based architectures).
RME_KOTBL	The kernel virtual address of the kernel object registration table. For most architectures, it is perfectly fine to define it as the default location in the kernel, RME_Kotbl. For architectures that have a huge amount of memory (latest x86-64 can manage hundreds of TBs of memory), due to the fact that GCC will place the kernel to high 2GB of kernel VA, the default size of kernel object

	registration table will be limited to 2GB, which only supports 1TB of kernel memory. When this happens, relocation of kernel object allocation table is needed and you should define this to a unrestricted address where the table will be mapped to. Example: #define RME_KOTBL RME_Kotbl (Small kernel object table) #define RME_KOTBL ((ptr_t*)0xFFFF800001000000) (Large kernel object table)
RME_READ_ACQUIRE(X)	The read memory barrier. On strongly ordered architectures, this can be implemented as a straightforward read from address X; in other cases, it should be defined as a function that includes a read barrier instruction, which guarantees that no memory operations below this line can begin before this read operation is completed. Compiler built-in functions can be used if they serve this purpose as well. This is equivalent to sequential read-read and read-write guarantees. It is worth noting that RME does not require the write-read pair to be strongly ordered; if on some architectures only write-read pairs can be reordered (e.g. x86-64), this macro also can be implemented as a straightforward read. Example: #define RME_READ_ACQUIRE(X) (*(X)) (Cases where only a straightforward read is required) #define RME_READ_ACQUIRE(X) asm("LDA X") (Cases where load-acquire instruction is provided) #define RME_READ_ACQUIRE(X) asm("LDR X; DMB") (Cases where memory barrier instruction is provided) #define RME_READ_ACQUIRE(X) __atomic_read(X) (Cases where compiler built-in function is provided)
RME_WRITE_RELEASE(X,V)	The write memory barrier. On strongly ordered architectures, this can be implemented as a straightforward write to address X; in other cases, it should be defined as a function that includes a write barrier instruction, which guarantees that this write operation will not begin before memory operations above this line are completed. Compiler built-in functions can be used if they serve this purpose as well. This is equivalent to sequential read-write and write-write guarantees. It is worth noting that RME does not require the write-read pair to be strongly ordered; if on some architectures only write-read pairs can be reordered (e.g. x86-64), this macro can be implemented as a straightforward write. Example: #define RME_WRITE_RELEASE(X,V) ((*(X))=(V)) (Cases where only a straightforward write is required)

	#define RME_WRITE_RELEASE(X,V) asm("STL X,V") (Cases where write-release instruction is provided) #define RME_WRITE_RELEASE(X,V) asm("DMB; STR X,V") (Cases where memory barrier instruction is provided) #define RME_WRITE_RELEASE(X,V) __atomic_write(X,V) (Cases where compiler built-in function is provided)

7.3.3 Architecture Related Structures

In RME, there are only 3 structures that are related with architecture, as listed below:

Structure	Explanation
RME_Reg_Struct	The register set push structure when entering interrupts, which includes all registers of the CPU.
RME_Cop_Struct	The coprocessor (FPU, etc.) register structure when entering interrupts, which includes all registers of the coprocessor.
RME_Iret_Struct	The program control-flow related registers that should be saved and restored upon synchronous invocation.

The implementation of the three register sets are related to how the interrupt assembly stub is implemented.

7.3.4 Low-level Assembly Functions

RME only requires a minimum of 4 low-level assembly functions. The name and meaning of these functions are as follows:

Function	Explanation
__RME_Disable_Int	Disable processor interrupts.
__RME_Enable_Int	Enable processor interrupts.
_RME_Kmain	Kernel entry shell function.
__RME_Enter_User_Mode	Switch to user level to continue execution.

Refer to the sections that follow for the detailed implementation guides of these functions.

7.3.5 System Interrupt Vectors

RME requires a minimum of 3 assembly or inline assembly interrupt vectors. The name and meaning of these vectors are listed below:

Vector	Explanation
System tick timer interrupt	Process timer interrupts for timeslice management.
System call interrupt	Process system calls.
System fault interrupt	Process memory management fault and other processor faults.

Refer to the sections that follow for the detailed implementation guides of these vectors.

7.3.6 Other Low-level Functions

These low-level functions involve page table, processor specific functions, etc. They can be implemented in assembly, inline assembly or C. These functions can be grouped into these categories:

7.3.6.1 Kernel Debug Print Function

Function	Explanation
__RME_Putchar	Print a character to kernel console.

7.3.6.2 Atomic Operations and Special Operations

Function	Explanation
__RME_Comp_Swap	Compare-and-Swap atomic operation.
__RME_Fetch_Add	Fetch-and-Add atomic operation.
__RME_Fetch_And	Fetch-and-And atomic operation.
__RME_MSB_Get	Get the Most Significant Bit (MSB) position of a machine word.

7.3.6.3 Initialization, Booting and CPUID Functions

Function	Explanation
__RME_Low_Level_Init	Low-level hardware initialization.
__RME_Boot	Create initial kernel object and boot the system.
__RME_Reboot	Reboot the kernel.
__RME_Shutdown	Shutdown the processor system.
__RME_CPUID_Get	Get the CPUID of the current CPU.

7.3.6.4 Register Set Related Functions

Function	Explanation
__RME_Get_Syscall_Param	Get system call parameters from register set.
__RME_Get_Inv_Retval	Get synchronous invocation return value from register set.
__RME_Set_Syscall_Retval	Set system call return value to register set.
__RME_Set_Inv_Retval	Set synchronous invocation return value to register set.
__RME_Thd_Reg_Init	Initialize a thread's register set.
__RME_Thd_Reg_Copy	Copy one register set to another.
__RME_Thd_Cop_Init	Initialize a thread's coprocessor register set.
__RME_Thd_Cop_Save	Save a thread's coprocessor register set.
__RME_Thd_Cop_Restore	Restore a thread's coprocessor register set.
__RME_Inv_Reg_Init	Initialize the register set of synchronous invocation.
__RME_Inv_Reg_Save	Save necessary registers for synchronous invocation returning.
__RME_Inv_Reg_Restore	Restore necessary registers for synchronous invocation returning.

7.3.6.5 Kernel Function

Function	Explanation
__RME_Kern_Func_Handler	Implementation of kernel functions.

7.3.6.6 Page Table Related Functions

Function	Explanation
__RME_Pgtbl_Set	Switch current page table (top-level page directory).
__RME_Pgtbl_Kmem_Init	Initialize kernel page table.
__RME_Pgtbl_Check	Check whether the page directory parameters can be supported.

__RME_Pgtbl_Init	Initialize page directory.
__RME_Pgtbl_Del_Check	Check whether the page directory can be deleted.
__RME_Pgtbl_Page_Map	Map a page into a page directory.
__RME_Pgtbl_Page_Unmap	Remove a mapping from a page directory.
__RME_Pgtbl_Pgdir_Map	Map a child page directory into a parent page directory.
__RME_Pgtbl_Pgdir_Unmap	Remove a child page directory from a parent page directory.
__RME_Pgtbl_Lookup	Look up a physical page in a page directory according to position.
__RME_Pgtbl_Walk	Look up a physical page and its flags from top-level.

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

For type definitions, we just need to confirm the size of integers in compilers and define these types with typedefs. It is worth noting that the for some architectures and compilers, long (long integer) corresponds to two machine words instead of one; if this is the case, the int type should be used. For some other architectures and compilers, int corresponds to half a machine word and long corresponds to a machine word, in which case long should be used.

If necessary, write some applet on the compiler with sizeof() operator to see what standard the compiler adheres to.

To facilitate porting of low-level functions, it is recommended to use the typedefs listed below to define the integers that are largely used. When defining these integers, we also need to confirm the size of compiler integer types such as char, short, int, and long. Some compilers does not support 64-bit or 128-bit integers, and these types can be omitted if they do not support them.

Type	Explanation
s8_t	Signed 8-bit integer. Example: typedef char s8_t;
s16_t	Signed 16-bit integer. Example: typedef short s16_t;
s32_t	Signed 32-bit integer. Example: typedef int s32_t;
s64_t	Signed 64-bit integer. Example: typedef long s64_t;
s128_t	Signed 128-bit integer. Example: typedef long long s128_t;
u8_t	Unsigned 8-bit integer. Example: typedef unsigned char u8_t;
u16_t	Unsigned 16-bit integer. Example: typedef unsigned short u16_t;
u32_t	Unsigned 32-bit integer. Example: typedef unsigned int u32_t;
u64_t	Unsigned 64-bit integer. Example: typedef unsigned long u64_t;
u128_t	Unsigned 128-bit integer. Example: typedef unsigned long long u128_t;

For macros and structures, they should be defined according to detailed system configurations. See the former sections for their definitions.

We will then explain the porting of low-level assembly functions.

7.4.1 Implementation of __RME_Disable_Int

Prototype	void __RME_Disable_Int(void)
Function	Disable processor interrupts.
Return	None.
Parameter	None.

This function disables the interrupt of the processor, then returns. There are no special precautions, as in most cases this only involves writing to a CPU register or peripheral address to disable interrupts then return immediately.

7.4.2 Implementation of __RME_Enable_Int

Prototype	void __RME_Enable_Int(void)
Function	Enable processor interrupts.
Return	None.
Parameter	None.

This function enables the interrupt of the processor, then returns. There are no special precautions, as in most cases this only involves writing to a CPU register or peripheral address to enable interrupts then return immediately.

7.4.3 Implementation of _RME_Kmain

Prototype	void _RME_Kmain(ptr_t Stack)
Function	The low-level kernel entry.
Return	None.
Parameter	ptr_t Stack The kernel stack virtual address.

This function just needs to assign Stack to the kernel-level stack pointer, then jump to RME_Kmain. This function never returns.

Before calling this function, the following work needs to be done:

1. Load the kernel into memory correctly via the bootloader, and set the processor to privileged state.

2. Set the initial kernel boot page table up, and switch to protected mode with this page table. Only kernel virtual address to kernel physical address mapping is needed on this page table. This page table is only used once during booting and will never be used later on, and can be deleted after booting. If the creation and loading of boot-time page table is not implemented prior to this function, it needs to be implemented here.

7.4.4 Implementation of __RME_Enter_User_Mode

Prototype	void __RME_Enter_User_Mode(ptr_t Entry_Addr, ptr_t Stack_Addr, ptr_t CPUID)
Function	Enter user-level and begin to execute in the first process.
Return	None.

Parameter	ptr_t Entry_Addr The entry virtual address of the first user-level application.
	ptr_t Stack_Addr The stack virtual address of the first user-level application.
	ptr_t CPUID The CPUID of the thread.

This function switches from privileged level to user level, and is only called at the last step of system booting. After this, the system is in normal state. We just need to assign Stack_Addr to the stack pointer, CPUID to the calling convention's first parameter register, and then jump to Entry_Addr with a processor mode switch. This function never returns.

7.5 Porting of System Interrupt Vectors

The porting of system interrupt vectors includes two parts, the first being entering & exiting code, the second part being the vector itself. RME merely requires 3 vectors. It is required that the registers be pushed to stack upon entering and popped from the stack upon exiting the vector. The vectors may involve saving and restoring coprocessor registers as well.

7.5.1 Entry & Exit of Interrupt Vectors and Architecture Related Structures

Upon entering of interrupt vectors, all the registers that are saved by the vector should be pushed, and the sequence of such a push should be the same as what is defined register structure. After pushing is complete, call the corresponding processing function, and pass the pointer to the register set to it. Upon exiting of interrupt vectors, we only need to pop the registers in a reverse manner. If a context switch is involved in interrupt vectors, the system will decide whether a coprocessor register set save/restore is required. If yes, the coprocessor register set will be saved and restored. The coprocessor registers will not be pushed to stack in most cases, thus the coprocessor structure just needs to include all the registers without considering their sequence.

If the stack is a full descending stack, the stack pointer after pushing is the structure pointer;

If the stack is an empty descending stack, the stack pointer after pushing plus a processor word length (in bytes) is the structure pointer.

If the stack is a full ascending stack, the stack pointer after pushing minus structure size plus a processor word length (in bytes) is the structure pointer.

If the stack is an empty descending stack, the stack pointer after minus structure size is the structure pointer.

Timer interrupt processing function, system call processing function and system fault handling function only takes register set pointer as their only parameter. Usually all three functions are implemented in C, thus the passing of the only parameter must conform to the C calling convention.

7.5.1.1 System Timer Interrupt Vector

The following function needs to be called in timer interrupt vector:

Prototype	void _RME_Tick_Handler(struct RME_Reg_Struct* Reg)
Function	Execute timer interrupt processing.
Return	None.
Parameter	struct RME_Reg_Struct* Reg

	The register set pushed upon vector entry.

This function is implemented by the system and the user does not need to implemented it. In a multi-processor system, this function is intended for the main processor; for auxiliary processors, call _RME_Tick_SMP_Handler instead. The two functions are same except that the auxiliary processor version does not update timestamp value. This is due to the fact that only the main processor handles timer interrupts and maintains the timestamp counter, after which it will send Inter-Processor Interrupts (IPIs) to other processors to trigger a reschedule.

7.5.1.2 System Call Interrupt Vector

The following function needs to be called in system call interrupt vector:

Prototype	void _RME_Svc_Handler(struct RME_Reg_Struct* Reg)
Function	Execute system call processing.
Return	None.
Parameter	struct RME_Reg_Struct* Reg The register set pushed upon vector entry.

This function is also implemented by the system and the user does not need to implemented it.

7.5.1.3 System Fault Handling Vector

An user-supplied fault handling function needs to be called in system fault handling vector. The name of the handler can be decided by the user, but its prototype must look like what is shown below. Refer to chapter 7.5.2 for implementation details.

Prototype	void _RME_Fault_Handler(struct RME_Reg_Struct* Reg)
Function	Execute system fault handling.
Return	None.
Parameter	struct RME_Reg_Struct* Reg The register set pushed upon vector entry.

7.5.2 Fault Handling Interrupt Vectors

An error handling function whose prototype is described in the last section needs to be called in this vector. The implementation of this function is architecture-specific and must be redesigned in each port. The function needs to judge whether the fault is a recoverable one or an unrecoverable one. If it is an unrecoverable one (memory access fault or undefined instruction), the following function needs to be called:

Prototype	ret_t __RME_Thd_Fatal(struct RME_Reg_Struct* Reg)
Function	The thread encountered a fatal fault or recovery attempt for a recoverable fault failed, needs to kill the thread immediately.
Return	ret_t Always return 0.
Parameter	struct RME_Reg_Struct* Reg The register set pushed upon vector entry.

If the fault is a recoverable one (swapping, page fault or MPU dynamic page replacement), the interrupt vector can try to perform a recovery and return immediately. If the recovery fails, the function above needs to be called to kill the thread, and then return immediately. This function will send

scheduler notifications and scheduler signals (if the thread registered a signal endpoint when it is binded) to let the user-level handle this fault.

7.6 Porting of the Kernel Debug Print Function

There is only one function for kernel debug printing:

Prototype	ptr_t __RME_Putchar(char Char)
Function	Output a character to the console.
Return	ptr_t Always return 0.
Parameter	char Char The character to output to the console.

In the implementation, the function should simply output a character to the console. The most common console device is the serial port.

7.7 Porting of Atomic Operations and Processor Specific Operations

Atomic operation functions are used to implement lockless kernels in a multi-core environment. Processor-specific operations can facilitate usage of some processor-specific functionality. These functions can be implemented in either C or assembly depending on the situation. If it needs to be implemented in assembly, the calling convention needs to be adhered to because they will called by C.

7.7.1 Compare-and-Swap

Prototype	ptr_t __RME_Comp_Swap(ptr_t* Ptr, ptr_t* Old, ptr_t New)
Function	A compare-and-swap operation. This operation will compare the value of *Old and *Ptr; if they do not equal, return 0 and assign *Ptr to *Old; if they equal, return 1 and assign New to *Ptr.
Return	ptr_t Indicates whether the function is successful. 1 for successful and 0 for failure.
Parameter	ptr_t* Ptr The pointer to the target address.
	ptr_t* Old The pointer to the old value.
	ptr_t New The value to assign to the target address if the old value equals the target value.

This function accomplishes a basic atomic compare-and-swap. On x86-64, there is one dedicated instruction (PREFIX LOCK CMPXCHG) to support this, and we can consider using assembly or inline assembly to implement it. On RISC architectures such as ARMv7, load exclusive (LDREX) and store exclusive (STREX) can be considered for implementation. On newer processors (ARMv8), load acquire exclusive (LDAEX) and write release exclusive (STLEX) can be used as well. The detailed implementation is architecture-specific.

When this function is implemented, it should include a full memory barrier before and after the actual compare-and-swap operation if your architecture is not strongly ordered (PowerPC, ARMv7, etc) and the atomic instruction itself is not memory-access serializing.

7.7.2 Fetch-and-Add

Prototype	ptr_t __RME_Fetch_Add(ptr_t* Ptr, cnt_t Addend)
Function	A fetch-and-add operation. This operation will add Addend to *Ptr, then return the value of *Ptr before Addend is added to it.
Return	ptr_t The *Ptr before Addend is added to it.
Parameter	ptr_t* Ptr The pointer to the target address.
	cnt_t Addend The number to add to the target. Can be positive or negative.

This function accomplishes a basic fetch-and-add. On x86-64, there is one dedicated instruction (PREFIX LOCK XADDL) to support this, and we can consider using assembly or inline assembly to implement it. On RISC architectures such as ARM, load exclusive (LDREX) and store exclusive (STREX) can be considered for implementation. The implementation is architecture-specific, and does not have to carry a full memory barrier effect.

7.7.3 Fetch-and-And

Prototype	ptr_t __RME_Fetch_And(ptr_t* Ptr, ptr_t Operand)
Function	A fetch-and-and operation. This operation will logic AND Operand with *Ptr, then return the value of *Ptr before Operand is ANDed withit.
Return	ptr_t The *Ptr before Operand is ANDed with it.
Parameter	ptr_t* Ptr The pointer to the target address.
	ptr_t Operand The unsigned number to AND with the target.

This function accomplishes a basic fetch-and-and. On x86-64, there is one dedicated instruction (PREFIX LOCK ANDL) to support this, and we can consider using assembly or inline assembly to implement it. On RISC architectures such as ARM, load exclusive (LDREX) and store exclusive (STREX) can be considered for implementation. The implementation is architecture-specific, and does not have to carry a full memory barrier effect.

7.7.4 Get the MSB Position of a Word

Prototype	ptr_t __RME_MSB_Get(ptr_t Val)
Function	Get the Most Significant Bit (MSB) position of a processor-word-length unsigned integer. MSB position refers to the position of the first binary “1” you encounter when counting from left to right.
Return	ptr_t The position of the MSB.
Parameter	ptr_t Val The number to calculate MSB for.

This function needs to be implemented in an architecture-specific efficient manner. For those processors that provided MSB computation instructions, it is best to implement this in assembly and

apply that instruction directly. For those architectures that provided Count Leading Zero (CLZ) instruction (such as ARM), an alternative is to compute the leading zeroes, and subtract the processor word length-1 with this value. For example, 0x12345678 have 3 leading zeroes, and we subtract 31 with 3 to get 28.

For those architectures that don't have any specialized instructions, divide-and-conquer can be applied. We can test whether the higher half of a word is zero, if no, search within the higher half, if yes, search within the lower half, until we encounter the first "1". When we have narrowed it down to 16 or 8 bits, we can use a lookup table to decide the relative position of the first "1" with in these 16 or 8 bits to spare some searches. At last, we need to sum up the results of each search to get the position of the first "1" in the word.

7.8 Porting of Initialization, Startup and CPUID Functions

There are 4 initialization, booting and CPUID related functions, as listed below:

7.8.1 Implementation of __RME_Low_Level_Init

Prototype	ptr_t __RME_Low_Level_Init(void)
Function	The low-level initialization of hardware. This usually include the initialization of processor clock configuration, miscellaneous supporting hardware (Cache controllers, interrupt controllers, or other peripherals that must be initialized on power-up). After this function finishes, the internal data structure initialization will begin.
Return	ptr_t Always return 0.
Parameter	None.

This function needs to initialize processor clock configuration, cache controllers, etc. except for the memory management/protection unit. This is due to the fact that the MMU/MPU have been initialized before _RME_Kmain exits.

7.8.2 Implementation of __RME_Boot

Prototype	ptr_t __RME_Boot(void)
Function	This function initializes the first process Init, creates all kernel signal endpoints/kernel function call capabilities, and adds all user accessible pages into Init process's initial page table.
Return	ptr_t Always return 0.
Parameter	None.

This function is the most important function during the boot sequence of RME. It runs in kernel level, and creates the capability table & page table of the Init process. It then adds all user-accessible page into the Init process, and subsequently creates all kernel signal endpoints and kernel function capabilities. In a single-core system, the function creates a thread, sets its execution properties and calls _RME_Enter_User_Mode at last to switch to it. In a multi-core system however, the system needs to initialize all other processors, and they must create a thread on their core, and then switch to it to continue execution.

All the functions that this function should call are listed below. Except for the last three user-supplied function, all other functions are provided by RME.

Function	Times called	Explanation
_RME_Kotbl_Init	Once per system	Initialize the kernel memory registration table.
_RME_CPU_Local_Init	Once per processor	Initialize CPU-local storage at boot-time.
_RME_Captbl_Boot_Init	Once per system	Create the initial capability table at boot-time.
_RME_Captbl_Boot_Crt	Depends on situation	Create other capability tables at boot-time.
_RME_Pgtbl_Boot_Crt	One set per system	Create page directories at boot-time.
_RME_Pgtbl_Boot_Con	One set per system	Construct page directories at boot-time.
_RME_Pgtbl_Boot_Add	One set per system	Add pages to page directories at boot-time.
_RME_Proc_Boot_Crt	Once per system	Create the first process at boot-time.
_RME_Kern_Boot_Crt	Once per system	Create kernel function capability at boot-time.
_RME_Kmem_Boot_Crt	Depends on situation	Create kernel memory capabilities at boot-time.
_RME_Sig_Boot_Crt	Depends on situation	Create kernel signal capabilities at boot-time.
_RME_Thd_Boot_Crt	Once per processor	Create the first threads at boot-time.
__RME_Pgtbl_Set	Once per processor	Load the processor with designated page table.
__RME_Enable_Int	Once per processor	Enable interrupts.
__RME_Enter_User_Mode	Once per processor	Go into the user-level to continue execution.

These functions are listed in logical sequence in which they should be called. The detailed explanation and calling methods are listed below. Note that if any of these functions fail, the system booting process must be halted. Therefore, it is recommended to use RME_ASSERT(func(...) = = 0) macro to encapsulate these functions. If any of them fail, the kernel will go into a panic loop and print information about the failure.

7.8.2.1 Initialize the Kernel Memory Allocation Table at Boot-time

This function is used at startup to initialize the kernel memory allocation registration table. The initialize work is just to clear the memory to zero, representing that no memory is currently used. When the kernel boots, this function will get called once to initialize the portion that is known to be present at compile-time. For architectures such as Cortex-M, this is sufficient. However, on some other architectures (such as x86-64), the amount of kernel memory needs to be detected, thus the function needs to be called again here to initialize the table as needed.

Prototype	ret_t _RME_Kotbl_Init(ptr_t Words)	
Parameter	Type	Explanation
Words	ptr_t	The size of the kernel memory allocation table, the unit of which being processor words. This number should be dynamically calculated according to kernel memory allocation granularity and kernel memory size. The method is to divide the kernel memory address space size by the granularity, and then divide that further by the number of bits in a machine word.

The possible return values of this function are:

Return	Explanation
0	Operation successful.
-1	The value passed in is smaller than what is set by the minimal default size. The minimal default size is set by macro RME_KMEM_SIZE.

7.8.2.2 Initialize CPU-local Storage at Boot-time

This function is used at boot-time to initialize the CPU-local storage of each processor. The CPU-local storage includes the CPUID, timer endpoint, interrupt endpoint , and the run-queue of the processor. This function will initialize all these data structures properly. For single-core architectures, this function just needs to be called once to initialize the statically defined storage area; for multi-core architectures, each processor must call this once to initialize their respective storage areas.

Prototype	void _RME_CPU_Local_Init(struct RME_CPU_Local* CPU_Local, ptr_t CPUID)	
Parameter	Type	Explanation
CPU_Local	...	Its type is struct RME_CPU_Local*, and it is a pointer to the CPU-local memory area to initialize.
CPUID	ptr_t	The CPUID of the CPU, which we use to identify the CPU in the system.

This function does not have a return value.

7.8.2.3 Create Initial Capability Table at Boot-time

This function is used to initialize the first capability table and put the capability to itself into the designated slot. Compared with the system call that does similar stuff, this function may only be called at startup, and does not need a upper-level capability table (because there are no other capability tables in the system now). This function does not need a kernel memory capability.

Prototype	ret_t _RME_Captbl_Boot_Init(cid_t Cap_Captbl, ptr_t Vaddr, ptr_t Entry_Num)	
Parameter	Type	Explanation
Cap_Captbl	cid_t	The position to place the capability to itself in. This capability number can only have a master encoding.
Vaddr	ptr_t	The kernel virtual address that this new capability table will use.
Entry_Num	ptr_t	The number of capabilities contained within this capability table, must range from 1 to RME_CAPID_2L.

The possible return values of this function are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The number of entries exceeded what is allowed by the RME.
	Cap_Crt exceeded range.
RME_ERR_CAP_KOTBL	Kernel memory allocation failed.

This function is only called once. It will create the initial capability table, which will be used to hold other capabilities when the kernel initializes itself.

7.8.2.4 Create Other Capability Tables at Boot-time

This function is used to create other capability table at boot-time, and it will place the capability to itself to the designated capability table. Compared with the initial capability table creation function, this function requires a upper-level capability table to contain the capability to itself; it does not place the capability to itself into itself. This function does not need a kernel memory capability.

Prototype	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)	
Parameter	Type	Explanation
Captbl	...	Its type is struct RME_Cap_Captbl*, and it is a pointer to the upper-level

		capability table capability. All capability numbers are relative to this capability table.
Cap_Captbl_Crt	cid_t	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_Captbl	cid_t	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	The kernel virtual address that this new capability table will use.
Entry_Num	ptr_t	The number of capabilities contained within this capability table, must range from 1 to RME_CAPID_2L.

The possible return values of this function are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	The number of entries exceeded what is allowed by the RME.
	Cap_Captbl_Crt exceeded range.
	Cap_Crt exceeded range.
RME_ERR_CAP_FROZEN (Unlikely to be returned)	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 doesn't have RME_CAPTBL_FLAG_CRT flag.
RME_ERR_CAP_EXIST	Cap_Crt is not an empty capability slot.
RME_ERR_CAP_KOTBL	Kernel memory allocation failed.

The number of times that this function should be called depends on the situation. If we need other capability table besides what _RME_Captbl_Boot_Init has created, we need to call it.

7.8.2.5 Create Page Directories at Boot-time

This function is used to create page directories at boot-time, and place the capability to this page directory into the designated capability table. This function does not need a kernel memory capability.

Prototype	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)	
Parameter	Type	Explanation
Captbl	...	Its type is struct RME_Cap_Captbl*, and it is a pointer to the upper-level capability table capability. All capability numbers are relative to this capability table.
Cap_Captbl	cid_t	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_Pgtbl	cid_t	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	The kernel virtual address that this new page directory will use.
Start_Addr	ptr_t	The mapping start virtual address of the new page directory. The last bit is top-level flag as explained below.
Top_Flag	ptr_t	Pass in "1" for top-level, "0" for other levels.
Size_Order	ptr_t	The size order of the page directory (the size of each page entry).
Num_Order	ptr_t	The number order of the page directory.

The possible return values of this function are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	Cap_Captbl exceeded range.
	Cap_Pgtbl exceeded range.
RME_ERR_CAP_FROZEN (Unlikely to be returned)	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 doesn't have RME_CAPTBL_FLAG_CRT flag.
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.

Create as many page directories as you want at boot-time. Thus, this may get called over and over again to create a bunch of page tables. Usually, at boot-time, we only need one processor to do the job, thus this function may be called many times at one place.

7.8.2.6 Construct Page Directories at Boot-time

This function is used to construct page directories at boot-time, and form the page directories into a directory tree (that is, page table). In the steps that follow, we will create the initial process with the initial capability table and page table.

Prototype	ret_t _RME_Pgtbl_Boot_Con(struct RME_Cap_Captbl* Captbl, cid_t Cap_Pgtbl_Parent, ptr_t Pos, cid_t Cap_Pgtbl_Child)	
Parameter	Type	Explanation
Captbl	...	Its type is struct RME_Cap_Captbl*, and it is a pointer to the upper-level capability table capability. All capability numbers are relative to this capability table.
Cap_Pgtbl_Parent	cid_t	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	A position in the parent page directory to accept the page directory construction. This position must be empty.
Cap_Pgtbl_Child	cid_t	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	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 function are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	Cap_Pgtbl_Parent or Cap_Pgtbl_Child exceeded range.
RME_ERR_CAP_FROZEN (Unlikely to be returned)	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 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.

Call this function whenever you need to construct page tables at boot-time. Usually, at boot-time, we only need one processor to do the job, thus this function may be called many times at one place.

7.8.2.7 Add Pages to Page Directories at Boot-time

This function will add pages to page directories. This operation ignores the page directory's operation flags. These pages will become the user level accessible pages when the system enters user-level. This function is the only place where we allow fabricating mappings; the user-level accessible pages can only be derived from these pages in the future. When these pages are mapped in, access permission flags should be provided. In the future operations, these pages will never possess the properties that they do not possess here.

Prototype	ret_t _RME_Pgtbl_Boot_Add(struct RME_Cap_Captbl* Captbl, cid_t Cap_Pgtbl, ptr_t Paddr, ptr_t Pos, ptr_t Flags)	
Parameter	Type	Explanation
Captbl	...	Its type is struct RME_Cap_Captbl*, and it is a pointer to the upper-level capability table capability. All capability numbers are relative to this capability table.
Paddr	ptr_t	The physical address of that page.
Pos	ptr_t	A position in the page directory to place the page in. It must be empty.
Flags	ptr_t	The access permission flags of the page, which restricts all future mappings.

The possible return values of this function are:

Return	Explanation

0	Operation successful.
RME_ERR_CAP_RANGE	Cap_Pgtbl exceeded range.
RME_ERR_CAP_FROZEN (Unlikely to be returned)	Cap_Pgtbl encountered a frozen capability.
RME_ERR_CAP_TYPE	Cap_Pgtbl is not a page directory capability.
RME_ERR_PGT_ADDR	Pos exceeded the maximum page number of the page directory.
	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_MAP	The mapping attempt failed due to hardware reasons.

Call this function whenever you need to map a physical page in at boot-time. In the normal boot sequence, only one processor are required to complete the task,which means that this function will be called many times in one place. Also, we do not put restrictions on how many times a physical page can be mapped. If multiple mappings are created, then they are all legal mappings that will persist.

7.8.2.8 Create the First Process at Boot-time

This function will create the first process at boot-time. The capability to this process will be placed into the designated capability table. This function does not need kernel memory capability.

Prototype	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)	
Parameter	Type	Explanation
Captbl	...	Its type is struct RME_Cap_Captbl*, and it is a pointer to the upper-level capability table capability. All capability numbers are relative to this capability table.
Cap_Captbl_Crt	cid_t	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_Proc	cid_t	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	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	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	The kernel virtual address that this new process will use.

The possible return values of this function are:

Return	Explanation
0	Operation successful.
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 (Unlikely to be returned)	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 doesn't have RME_CAPTBL_FLAG_CRT flag.
	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 (Unlikely to be returned)	The reference count of Cap_Captbl or Cap_Pgtbl exceeded the maximum reference count allowed.

This function should be called only once by one of the CPUs.

7.8.2.9 Create Kernel Function Capability at Boot-time

This function will create the initial kernel function capability at boot-time, and place it into the designated capability table. The kernel function capability should only be created at system boot-time, and all future kernel function capabilities should be derived from it through delegation.

Prototype	ret_t _RME_Kern_Boot_Crt(struct RME_Cap_Captbl* Captbl, cid_t Cap_Captbl, cid_t Cap_Kern)	
Parameter	Type	Explanation
Captbl	...	Its type is struct RME_Cap_Captbl*, and it is a pointer to the upper-level capability table capability. All capability numbers are relative to this capability table.
Cap_Captbl	cid_t	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 kernel function capability. This capability number can have either a master encoding or a expanded encoding.
Cap_Kern	cid_t	A capability number that corresponds to a capability slot that will accept the newly created kernel function capability. This slot must be empty. This capability number can only have a master encoding.

The possible return values of this function are:

Return	Explanation
0	Operation successful.

RME_ERR_CAP_RANGE	Cap_Captbl exceeded range.
	Cap_Kern exceeded range.
RME_ERR_CAP_FROZEN (Unlikely to be returned)	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 doesn't have RME_CAPTBL_FLAG_CRT flag.
RME_ERR_CAP_EXIST	Cap_Kern is not an empty capability slot.

This function should be called only once by one of the CPUs.

7.8.2.10 Create Kernel Memory Capability at Boot-time

This function will create kernel memory capability at boot-time, and place it into the designated capability table. The kernel memory capability should only be created at system boot-time, and all future kernel memory capabilities should be derived from it through delegation.

Prototype	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)	
Parameter	Type	Explanation
Captbl	...	Its type is struct RME_Cap_Captbl*, and it is a pointer to the upper-level capability table capability. All capability numbers are relative to this capability table.
Cap_Captbl	cid_t	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 kernel memory capability. This capability number can have either a master encoding or a expanded encoding.
Cap_Kmem	cid_t	A capability number that corresponds to a capability slot that will accept the newly created kernel memory capability. This slot must be empty. This capability number can only have a master encoding.
Start	ptr_t	The start virtual address of the kernel memory. When this is passed in, it will be automatically aligned to the granularity of kernel memory allocation table.
End	ptr_t	The end virtual address of the kernel memory. When this is passed in, it will be automatically aligned to the granularity of kernel memory allocation table minus 1.
Flags	ptr_t	The operation flags of the kernel memory capability, which denotes what can be created on this memory range. This cannot be zero or the kernel will panic.

The possible return values of this function are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	Cap_Captbl exceeded range.
	Cap_Kmem exceeded range.
RME_ERR_CAP_FROZEN (Unlikely to be returned)	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 doesn't have RME_CAPTBL_FLAG_CRT flag.
RME_ERR_CAP_EXIST	Cap_Kmem is not an empty capability slot.

Usually this function is only called once on one of the CPUs. If the system memory is sliced into different partitions, wach of which hold different properties, multiple kernel memory capabilities can be created.

7.8.2.11 Create Kernel Signal Endpoints at Boot-time

This function will create kernel signal endpoints at boot-time. Kernel endpoints are used in interrupt handling, and we will send a signal to this endpoint in the interrupt to wake up the waiting thread on that endpoint. There can only be one thread blocked on one endpoint at any time, thus the number of endpoints created should be the number of "interrupt vector - handling thread" pairs.

The kernel signal endpoint capabilities should only be created at system boot-time, and all future kernel signal capabilities should be derived from them through delegation. This function does not need kernel memory capability.

Prototype	ret_t _RME_Sig_Boot_Crt(struct RME_Cap_Captbl* Captbl, cid_t Cap_Captbl, cid_t Cap_Sig, ptr_t Vaddr)	
Parameter	Type	Explanation
Captbl	...	Its type is struct RME_Cap_Captbl*, and it is a pointer to the upper-level capability table capability. All capability numbers are relative to this capability table.
Cap_Captbl	cid_t	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 kernel signal endpoint capability. This capability number can have either a master encoding or a expanded encoding.
Cap_Sig	cid_t	A capability number that corresponds to a capability slot that will accept the newly created kernel signal endpoint capability. This slot must be empty. This capability number can only have a master encoding.
Vaddr	ptr_t	The kernel virtual address that this new kernel signal endpoint will use.

The possible return values of this function are:

Return	Explanation
0	Operation successful.
RME_ERR_CAP_RANGE	Cap_Captbl exceeded range.
	Cap_Sig exceeded range.
RME_ERR_CAP_FROZEN (Unlikely to be returned)	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 doesn't have RME_CAPTBL_FLAG_CRT flag.
RME_ERR_CAP_EXIST	Cap_Sig is not an empty capability slot.
RME_ERR_CAP_KOTBL	Kernel memory allocation failed.

There are two approaches regarding the usage of this function. In the first method, one processor will create all signal endpoints. In the second method, each processor creates its own kernel signal endpoints. The second method is faster but more complex. Usually the first method is sufficient.

If the first mode is used in a multi-processor system, IPIs needs to be sent to other CPUs to let them execute their own kernel object initialization routine after this step finishes. If the second mode is used, this needs to be done before this step, and all steps after this step needs to be executed once on each processor.

At least two signal endpoints should be created for each core: the first signal endpoint is for timer signals, and the pointer should be assigned to the Tick_Sig member of the CPU's local data structure; the second is for device interrupt handling, and the pointer should be assigned to the Int_Sig member of the CPU's local data structure.

7.8.2.12 Create Initial Threads at Boot-time

This function will create a initial thread which is Init at system boot-time. The Init threads will be binded to some processor once created, and possess infinite timeslices. Each CPU will have a Init thread in the Init process. The Init threads cannot be killed, unbinded or blocked on any endpoint; however, its priority can be changed. The maximum priority limit of Init threads is assigned to RME_MAX_PREEMPT_PRIO-1 by the system. The TID of Init threads is always zero.

Different from the system call that creates threads, this function allows designating the CPU to be binded to by passing in a CPU_Local parameter. This facilitates using the boot processor to create all kernel objects, which is recommended. This function does not need kernel memory capability.

Prototype	ret_t _RME_Thd_Boot_Crt(struct RME_Cap_Captbl* Captbl, cid_t Cap_Captbl, cid_t Cap_Thd, cid_t Cap_Proc, ptr_t Vaddr, ptr_t Prio, struct RME_CPU_Local* CPU_Local)	
Parameter	Type	Explanation
Captbl	...	Its type is struct RME_Cap_Captbl*, and it is a pointer to the upper-level capability table capability. All capability numbers are relative to this capability table.
Cap_Captbl	cid_t	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_Thd	cid_t	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	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.
Vaddr	ptr_t	The kernel virtual address that this new thread will use.
Prio	ptr_t	The preemption priority of the thread. The larger the number, the higher the priority. This value cannot exceed what is allowed by the system.
CPU_Local	...	Its type is struct RME_CPU_Local*, and it is a pointer to the CPU-local data structure of the CPU to bind the thread to.

The possible return values of this function are:

Return	Explanation

Non-nagative value	Operation successful, Thread IDentifier (TID) is returned.
RME_ERR_CAP_RANGE	Cap_Captbl exceeded range.
	Cap_Proc exceeded range.
	Cap_Thd exceeded range.
RME_ERR_CAP_FROZEN (Unlikely to be returned)	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 doesn't have RME_CAPTBL_FLAG_CRT flag.
	Cap_Proc doesn't have RME_PROC_FLAG_THD property.
RME_ERR_CAP_EXIST	Cap_Thd is not an empty capability slot.
RME_ERR_CAP_KOTBL	Kernel memory allocation failed.
RME_ERR_PTH_PRIO	The priority exceeded RME_MAX_PREEMPT_PRIO-1.

This function needs to be called once on every CPU, and an Init thread that binded to itself will be created. Alternatively, one CPU can create Init threads for all other CPUs.

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

__RME_Pgtbl_Set, __RME_Enable_Int and __RME_Enter_User_Mode should be called on each processor should call, and the processor will enter user level after that. Please refer to "porting of page table related function" section for information on __RME_Pgtbl_Set, and "porting of low-level assembly functions" section for information on __RME_Enable_Int and __RME_Enter_User_Mode.

7.8.3 Implementation of __RME_Reboot

This function is for rebooting the system. If it is not needed (to reboot the kernel upon panic or user request), this can be left empty. This function will soft-reset the processor, reset all its registers, and reboot.

Prototype	ptr_t __RME_Reboot(void)
Function	Reset and reboot the system.
Return	ptr_t Never returns.
Parameter	None.

7.8.4 Implementation of __RME_Shutdown

This function is for shutting down the system. If it is not needed (the system never shuts down), this can be left empty. This function will shutdown the system and its power supply.

Prototype	ptr_t __RME_Shutdown(void)
Function	Shutdown the system.
Return	ptr_t Never returns.
Parameter	None.

7.8.5 Implementation of __RME_CPUID_Get

This function is for returning the CPUID of a particular CPU when it gets called on it, to distinguish between different CPUs between each other. For single-core systems, just return 1. For multi-core systems, CPUID is a numerical value that ranges from 0 to RME_CPU_NUM-1. If the processor hardware returns a value other than this, this function is responsible for transforming it into a value that start from 0, and each value corresponds to a processor.

Prototype	ptr_t __RME_CPUID_Get(void)
Function	Get the CPUID of the processor.
Return	ptr_t The CPUID of the processor.
Parameter	None.

7.9 Porting of Register Set Related Functions

In RME, there are 12 register-related functions. These functions are very nimble and only involves register context. The implementation of these functions are usually related to architecture-related register structures, and how user-level library use these registers.

7.9.1 Implementation of __RME_Get_Syscall_Param

This function will extract system call parameters from the register set.

Prototype	void __RME_Get_Syscall_Param(struct RME_Reg_Struct* Reg, ptr_t* Svc, ptr_t* Capid, ptr_t* Param)
Function	Extract the system call parameters and place them in return values.
Return	None。
Parameter	struct RME_Reg_Struct* Reg The pointer to the register set.
	ptr_t* Svc This parameter is for output, and outputs the half-word system call number (N).
	ptr_t* Capid This parameter is for output, and outputs the half-word capability number (C).
	ptr_t* Param This parameter is for output, and outputs three word-long parameters (P1-P3).

7.9.2 Implementation of __RME_Set_Syscall_Retval

This function will save the system call return value to the register set.

Prototype	void __RME_Set_Syscall_Retval(struct RME_Reg_Struct* Reg, ret_t Retval)
Function	Save the system call return value into the register set.
Return	None.
Parameter	struct RME_Reg_Struct* Reg This parameter is for output, and is a pointer to the register set. ret_t Retval The return value of the system call.

7.9.3 Implementation of __RME_Thd_Reg_Init

This function will initialize the register of the thread. It will be called when a thread's execution property is being set or an invocation port is being activated.

Prototype	void __RME_Thd_Reg_Init(ptr_t Entry, ptr_t Stack, ptr_t Param, struct RME_Reg_Struct* Reg)
Function	Initialize the register set of the thread with entry, stack and parameter.
Return	None.
Parameter	ptr_t Entry The entry of the thread.
	ptr_t Stack The stack address of the thread.
	ptr_t Param The parameter to assign to the thread.
	struct RME_Reg_Struct* Reg This parameter is for output, and is a pointer to the register set.

7.9.4 Implementation of __RME_Thd_Reg_Copy

This function will replicate the thread's register set. Sometimes assembly implementation is required for efficiency.

Prototype	void __RME_Thd_Reg_Copy(struct RME_Reg_Struct* Dst, struct RME_Reg_Struct* Src)
Function	Copy the register set from one to another.
Return	None.
Parameter	struct RME_Reg_Struct* Dst This parameter is for output, and is a pointer to the target register set.
	struct RME_Reg_Struct* Reg The pointer to the source register set.

7.9.5 Implementation of __RME_Thd_Cop_Init

This function will initialize the coprocessor of a thread, and is only used when setting execution properties of threads. On some systems, the coprocessor also needs to be initialized, however for most systems this does not need to be done.

Prototype	void __RME_Thd_Cop_Init(struct RME_Reg_Struct* Reg, struct RME_Cop_Struct* Cop_Reg)
Function	Initialize the coprocessor register set.
Return	None.
Parameter	struct RME_Reg_Struct* Reg A pointer to the register data structure. This parameter is here to help initialization.
	struct RME_Cop_Struct* Cop_Reg This parameter is for output, and is a pointer to the target coprocessor register set.

7.9.6 Implementation of __RME_Thd_Cop_Save

This function will save the coprocessor registers of a thread. Sometimes an assembly implementation is required for efficiency.

Prototype	void __RME_Thd_Cop_Save(struct RME_Reg_Struct* Reg, struct RME_Cop_Struct* Cop_Reg)
Function	Save the coprocessor register set of the thread.
Return	None.
Parameter	struct RME_Reg_Struct* Reg A pointer to the register data structure. This parameter is here to help determining whether the coprocessor state needs to be saved. For some processors, whether the coprocessor is active will be indicated in the program status word or some special register, and whether the coprocessor state needs to be saved can be judged from them.
	struct RME_Cop_Struct* Cop_Reg The pointer to the target coprocessor register set.

7.9.7 Implementation of __RME_Thd_Cop_Restore

This function will restore the coprocessor registers of a thread. Sometimes an assembly implementation is required for efficiency.

Prototype	void __RME_Thd_Cop_Restore(struct RME_Reg_Struct* Reg, struct RME_Cop_Struct* Cop_Reg)
Function	Restore the coprocessor register set of the thread.。
Return	None.
Parameter	struct RME_Reg_Struct* Reg The pointer to the register data structure. This parameter is here to help determining whether the coprocessor state needs to be restored. For some processors, whether the coprocessor is active will be indicated in the program status word or some special register, and whether the coprocessor state needs to be restored can be judged from them.
	struct RME_Cop_Struct* Cop_Reg This parameter is for output, and is a pointer to the target coprocessor register set.

It is worth noting that the coprocessor register set can be used as a (usually very capable) covert channel. Thus, in the implementations that emphasize security, random or zero fills should be performed if the current thread is not using the coprocessor register set; the other alternative implementation is always saving and restoring the coprocessor register sets regardless of whether they are used or not.

7.9.8 Implementation of __RME_Inv_Reg_Save

This function will save registers that are essential for synchronous invocation returning to the invocation return structure. Only the registers that are critical for execution flow control recovery need to be saved. For x86-64, SP and IP need to be saved; for Cortex-M, LR and SP need to be saved.

Prototype	void __RME_Inv_Reg_Save(struct RME_Iret_Struct* Ret, struct RME_Reg_Struct* Reg)
Function	Save the necessary registers to invocation return register structure.
Return	None.
Parameter	struct RME_Iret_Struct* Ret This parameter is for output, and is a pointer to the invocation return register set.
	struct RME_Reg_Struct* Reg

	The pointer to the register set.

7.9.9 Implementation of __RME_Inv_Reg_Restore

This function will restore registers that are essential for synchronous invocation returning from the invocation return structure. Only the registers that are critical for execution flow control recovery need to be restored. For x86-64, SP and IP need to be restored; for Cortex-M, LR and SP need to be restored.

Prototype	void __RME_Inv_Reg_Restore(struct RME_Reg_Struct* Reg, struct RME_Iret_Struct* Ret)
Function	Restore the necessary registers from invocation return register structure.
Return	None.
Parameter	struct RME_Reg_Struct* Reg This parameter is for output, and is a pointer to the register set.
	struct RME_Iret_Struct* Ret The pointer to the invocation return register set.

7.9.10 Implementation of __RME_Set_Inv_Retval

This function will save the synchronous invocation return value into the register set.

Prototype	void __RME_Set_Inv_Retval(struct RME_Reg_Struct* Reg, ret_t Retval)
Function	Save the synchronous invocation return value into the register set.
Return	None.
Parameter	struct RME_Reg_Struct* Reg This parameter is for output, and is a pointer to the register set.
	ret_t Retval The return value of the synchronous invocation.

7.10 Porting of Kernel Functions

Kernel function is a set of user-defined functions that can be called in kernel space. This functions are defined at compile-time. The prototype of these functions are as follows:

Prototype	ptr_t __User_Func(struct RME_Reg_Struct* Reg, ptr_t Sub_ID, ptr_t Param1, ptr_t Param2)
Function	Implement a user-defined kernel-level operation.
Return	ptr_t If failed, a negative value must be returned; if successful, a non-negative value must be returned. If the function is successful, it is responsible for storing its return value back to the register set.
Parameter	struct RME_Reg_Struct* Reg This parameter can be used as input or output, and is a pointer to the register set.
	ptr_t Sub_ID The subfunction ID.
	ptr_t Param1 The first parameter of the function.
	ptr_t Param2

	The second parameter of the function.

This function accepts two user-defined parameter and complete some operations at kernel level, then returns. Usually these functions are used to implement some architecture-specific functions, such as built-in peripherals, coprocessor instructions or other I/O operations that must be completed at kernel-level. These functions should be as nimble as possible and must have a bounded execution time, or the system's real-time performance will be hampered.

The most common kernel function types and their implementation intuitions are listed below for your reference.

7.10.1 Tickless Kernel Implementation

Tickless kernel usually requires a high-precision timer that can only be configured at kernel level, and that timer will be responsible for generating the system's scheduler timer interrupt. The concrete implementation varies between different processors, but the intuitions are very similar.

In a tickless kernel, the timer interrupt will not call the periodic timer processing function that RME provided. Instead, it sends to a kernel endpoint, and increase the value of RME_Timestamp accordingly. The scheduler thread that blocks on the kernel endpoint will get unblocked and make scheduling decisions at user-level.

The maximum inter-arrival time of ticks can be configured as needed. It is worth noting that this value must be as small as possible, or the RME_Timestamp will be stale for a long time, which will in turn hamper the creation, freezing, deletion and removal of capabilities. From a realistic standpoint, the recommended maximum inter-arrival time is 200ms.

7.10.2 High Precision Timer System Implementation

The implementation of high precision timer is similar to tickless kernels. We just need to design a few kernel functions, and grant them the ability to operate on timers. The interrupts from these timers will routed to the target thread through kernel endpoints; it can also be sent to a management thread that relays these timer interrupts to the target threads.

7.10.3 Inter-Processor-Interrupt Implementation

Due to the fact that asynchronous sends in RME across different CPUs cannot wakeup their threads, a kernel function is needed to handle inter-processor interrupts. This interrupt will remind the receiving CPU that one of its threads needs to be waken up.

7.10.4 Cache Maintenance Operation Implementation

The cache maintenance operations are usually privileged. Thus, these functions can be implemented as distinct kernel functions.

7.11 Porting of Page Table Related Functions

In RME, there are 11 page table related functions. The implementation of these functions are heavily architecture-specific, and will be responsible for the conflict avoidance of parallel operations under multi-core circumstances. The security and reliability of these functions will greatly impact system security and reliability, thus they are the core of system porting. See the detailed description about these functions below for details.

7.11.1 Implementation of __RME_Pgtbl_Set

This function will set the current page table that the processor is using. A virtual address is passed in, thus this function always needs to convert virtual addresses to physical addresses before actually assigning this value to the corresponding register of the processor.

Prototype	void __RME_Pgtbl_Set(ptr_t Pgtbl)
Function	Set the processor's current page table.
Return	None.
Parameter	ptr_t Pgtbl The kernel virtual address where the hardware-recognizable page table is placed.

For MMU-based architectures, this function will point the top-level page directory pointer (e.g. CR3 in x86-64) to the top-level page directory, and refresh the TLB in the meantime. For MPU-based architectures, this function will copy the top-level metadata into the corresponding MPU registers to set the regions to protect. MPU usually have a large number of registers, thus it is recommended to use assembly to implement the function for better performance.

7.11.2 Implementation of __RME_Pgtbl_Kmem_Init

This function will establish initial kernel page table at system boot-time.

Prototype	ptr_t __RME_Pgtbl_Kmem_Init(void)
Function	Establish initial kernel page table.
Return	ptr_t 0 on success, RME_ERR_PGT_OPFAIL (-1) on failure.
Parameter	None.

The kernel mappings established by this function will never be changed once created. These physical addresses will be permanently mapped into the kernel as kernel memory. These kernel page table (or kernel page directories) will be mapped into each top-level page directory of each process, and its privilege level is kernel level. Two portions of memory exist in this mapping; the first portion is the kernel virtual address space, the second portion is the hypervisor reserved virtual address space. See sections on __RME_Pgtbl_Init for how these kernel page directories are mapped.

On MPU-based architectures, directly returning success is sufficient, because the privileged mode usually have access to all memory by default, thus no need to add kernel entries into MPU registers.

7.11.3 Implementation of __RME_Pgtbl_Check

This function will check whether the parameters passed in for page directory creation is supported by the low-level architecture.

Prototype	ptr_t __RME_Pgtbl_Check(ptr_t Start_Addr, ptr_t Top_Flag, ptr_t Size_Order, ptr_t Num_Order, ptr_t Vaddr)
Function	Check whether the parameters passed in are supported by hardware.
Return	ptr_t 0 on success (supported), RME_ERR_PGT_OPFAIL (-1) on failure (unsupported).
Parameter	ptr_t Start_Addr The start mapping virtual address of the page directory. This is only effective on MPU-based architectures.
	ptr_t Top_Flag

	Whether the page directory is a top-level one. 1 for top-level, 0 for non top-level.
	ptr_t Size_Order The size order of the page directory.
	ptr_t Num_Order The number order of the page directory.
	ptr_t Vaddr The virtual address of the page directory.

This function will be called before the system call that creates page directories gets called, to check whether the parameters are valid before we allocate any kernel memory. This function needs to strictly adhere to the requirements of processor page table restrictions, and only return success for these supported combinations. For illegal parameter combinations, RME_ERR_PGT_OPFAIL (-1) must be returned.

7.11.4 Implementation of __RME_Pgtbl_Init

This function will initialize a page directory that is just created.

Prototype	ptr_t __RME_Pgtbl_Init(struct RME_Cap_Pgtbl* Pgtbl_Op)
Function	Initialize the page directory just created.
Return	ptr_t 0 on success, RME_ERR_PGT_OPFAIL (-1) on failure.
Parameter	struct RME_Cap_Pgtbl* Pgtbl_Op The pointr to the page directory capability that points to the page directory.

This function will be called when the page directory is created, and is also the critical function for page directory creation. This function will initialize the page directory to a usable format i.e. initialize all its entries to empty. Additionally, if we are creating top-level page directory, we need to add all the kernel entries that are created by __RME_Pgtbl_Kmem_Init into that page directory. Because __RME_Pgtbl_Check have already performed checks on the page directory, thus the checks can be skipped.

7.11.5 Implementation of __RME_Pgtbl_Del_Check

This function will check whether a page directory can be safely deleted.

Prototype	ptr_t __RME_Pgtbl_Del_Check(struct RME_Cap_Pgtbl* Pgtbl_Op)
Function	Check whether a page directory can be safely deleted.
Return	ptr_t 0 on success (can delete), RME_ERR_PGT_OPFAIL (-1) on failure (cannot delete).
Parameter	struct RME_Cap_Pgtbl* Pgtbl_Op The pointer to the page directory capability that points to the page directory. This capability contains all the information about the page directory to be deleted.

This function will get called before the page directory is deleted. In this function, we need to check whether this page table is referenced by upper-level page directories. If yes, this page directory cannot be directly deleted. Additionally, we need to make sure that this page directory does not contain child page directories. If there is, then it also cannot be deleted.

If this function just checks one or none of the conditions listed above, then the user-level is responsible for the correctness of page directory deletion. The user-level memory manager must

guarantee that the page directory will not be referenced or still referencing others upon deletion. If the user-level library also skips these checks, the kernel can be compromised.

7.11.6 Implementation of __RME_Pgtbl_Page_Map

This function will map a page into a page directory. If the mapping fails due to incorrect parameters (position overrun, physical page not aligned, extra flags passed in or a mapping already exists), an error should be returned.

Prototype	ptr_t __RME_Pgtbl_Page_Map(struct RME_Cap_Pgtbl* Pgtbl_Op, ptr_t Paddr, ptr_t Pos, ptr_t Flags)
Function	Map a page into a page directory.
Return	ptr_t 0 on success, RME_ERR_PGT_OPFAIL (-1) on failure.
Parameter	struct RME_Cap_Pgtbl* Pgtbl_Op The page directory capability that points to the page directory. This capability contains all the information about the page directory.
	ptr_t Paddr The physical address of the page to map.
	ptr_t Pos The position in the page table to map in.
	ptr_t Flags The RME standard access permission flags of the page.

The "Pos" listed above is the position of the page in the page directory. Let's assume that every entry of the page table represents a 4kB page. Then 12kB will be the start address of the 3rd position (the position numbers are counted from 0). "RME standard flags" refer to the abstract flags that are used by the RME. These flags are different from architecture dependent concrete flags; please refer to chapter 3 for detailed description. This function needs to convert these flags to the flags that can be directly recognized by the processor, then write them into the page table. For those processors that does not support some page flags, these flags can be directly ignored. For those hardware-loaded MMUs, the "static (RME_PGTBL_STATIC)" flag can be ignored.

In multi-core environments, this function needs to guarantee that two CPUs will not attempt to mp two pages to one position. If this happens, this can be prevented with a read-modify-write (compare-and-swap) operation. In MPU-based environments, this function is also responsible for updating the top-level MPU metadata and insert the mapping of this page.

7.11.7 Implementation of __RME_Pgtbl_Page_Unmap

This function will unmap a page in a page table. If the unmapping fails due to incorrect parameters (position overrun or physical page does not exist), an error should be returned.

Prototype	ptr_t __RME_Pgtbl_Page_Unmap(struct RME_Cap_Pgtbl* Pgtbl_Op, ptr_t Pos)
Function	Unmap a page from a page directory.
Return	ptr_t 0 on success, RME_ERR_PGT_OPFAIL (-1) on failure.
Parameter	struct RME_Cap_Pgtbl* Pgtbl_Op The page directory capability that points to the page directory. This capability contains all

	the information about the page directory.
	ptr_t Pos The position in the page table to unmap from.

This function is the reverse operation of the function above it, and just removes the mapping. In multi-core environments, we also need to make sure that two CPU will not conflict when them simultaneously remove the same mapping. In MMU-based environments, this function needs to reload and flush the TLB; when you have dedicated TLB maintenance instructions (such as INVLTB of x86-64), you can also make use of it if you can know the virtual address of this entry. Alternatively, you can skip the TLB flush for performance and predictability, and instead call special kernel functions to reload TLB periodically. When this method is applied, user-level page quiescence needs to be tracked (because the pages unmapped will disappear in the accessible memory range only after a while). In MPU-based environments, this function needs to update the top-level MPU metadata and remove the mapping of this page.

7.11.8 Implementation of __RME_Pgtbl_Pgdir_Map

This function will map a child directory into the parent directory. If the mapping fails due to incorrect parameters (position overrun, physical address alignment error, virtual address relationship error, a mapping is already present or MPU-imposed constraints are not satisfied), an error should be returned.

Prototype	ptr_t __RME_Pgtbl_Pgdir_Map(struct RME_Cap_Pgtbl* Pgtbl_Parent, ptr_t Pos, struct RME_Cap_Pgtbl* Pgtbl_Child, ptr_t Flags)
Function	Map a child page directory into a parent page directory.
Return	ptr_t 0 on success, RME_ERR_PGT_OPFAIL (-1) on failure.
Parameter	struct RME_Cap_Pgtbl* Pgtbl_Parent The page directory capability that points to the parent page directory. This capability contains all the information about the parent page directory.
	ptr_t Pos The position in the parent page table to map the child page table in.
	struct RME_Cap_Pgtbl* Pgtbl_Child The page directory capability that points to the child page directory. This capability contains all the information about the child page directory.
	ptr_t Flags The RME standard flags of the child page directory. These flags set the access permissions of the child directory and all lower-level page directories. For those architectures that does not allow this flag or MPU-based architectures, this value is invalid.

This function behaves differently between MMU-based and MPU-based environments. In MMU-based environments, this mapping does not need to check whether the starting virtual address is valid, but it requires that the address size that the child page directory includes must be a exact page size of the parent page directory. In MPU-based environments, because we can employ path-compressed page table, the address size that the child directory includes can be smaller than a

page size of the parent directory. However, we need to check whether the starting virtual address is valid.

In MPU-based environments, due to some idiosyncrasies of MPU (see chapter on memory management for details), the parent page directory itself must have (or be) the top-level page directory, and the child directory must have no (and can't be) the top-level directory. Additionally, after the mapping finishes, if there are pages in the child directory that are mapped, the top-level MPU metadata must be updated so that these pages can be mapped in.

7.11.9 Implementation of __RME_Pgtbl_Pgdir_Unmap

This function will unmap a child page directory from the parent page directory. If the mapping fails due to incorrect parameters (position overrun, physical address alignment error, virtual address relationship error, a mapping is already present or MPU-imposed constraints are not satisfied), an error should be returned.

If the mapping fails due to incorrect parameters (position overrun, no mapping presents or MPU-imposed constraints are not satisfied), an error should be returned.

Prototype	ptr_t __RME_Pgtbl_Pgdir_Unmap(struct RME_Cap_Pgtbl* Pgtbl_Op, ptr_t Pos)
Function	Unmap a child page directory's mapping from a parent page directory.
Return	ptr_t 0 on success, RME_ERR_PGT_OPFAIL (-1) on failure.
Parameter	struct RME_Cap_Pgtbl* Pgtbl_Op The page directory capability that points to the parent page directory. This capability contains all the information about the parent page directory.
	ptr_t Pos The position in the parent page table to unmap from.

This function is the reverse operation of the function above it, and just removes the mapping of the child page directory. In multi-core environments, parallel operation conflicts must also be avoided. In MMU-based environments, this function needs to reload and flush the TLB. Alternatively, you can apply the quiescence method mentioned above. In MPU-based environments, this function needs to update the top-level MPU metadata and remove the mapping of the child page directory.

7.11.10 Implementation of __RME_Pgtbl_Lookup

This function will look up a page in a page directory.

Prototype	ptr_t __RME_Pgtbl_Lookup(struct RME_Cap_Pgtbl* Pgtbl_Op, ptr_t Pos, ptr_t* Paddr, ptr_t* Flags)
Function	Look up a page in a page directory and return it.
Return	ptr_t 0 on success (page found), RME_ERR_PGT_OPFAIL (-1) on failure (not found or is a page directory).
Parameter	struct RME_Cap_Pgtbl* Pgtbl_Op The page directory capability that points to the page directory. This capability contains all the information about the page directory.
	ptr_t Pos The position in the page table to look up.

	ptr_t* Paddr This parameter is for output, and is a pointer to the physical address of the page.
	ptr_t* Flags This parameter is for output, and is a pointer to the permission flags of the page.

This function just needs to look up the corresponding information on the designated page and place that into where the output pointers points to. The hardware-specific page flags need to be converted to RME standard flags before outputting them. Also, the two output parameters are both optional. If only one is needed, just pass 0 (NULL) into another parameter to suppress its output.

7.11.11 Implementation of __RME_Pgtbl_Walk

This function will look up in a page table (page directory tree) whether a virtual address is mapped and its information. This function should only be called from top-level of the page table, and an error should be returned if it is initiated from other levels.

Prototype	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)
Function	Look up in a page table (page directory tree) whether a virtual address is mapped an return its information.
Return	ptr_t 0 on success (page found), RME_ERR_PGT_OPFAIL (-1) on failure (not found).
Parameter	struct RME_Cap_Pgtbl* Pgtbl_Op The page directory capability that points to the page directory. This capability contains all the information about the page directory. This page directory must be a top-level one.
	ptr_t Vaddr The virtual address to consult.
	ptr_t* Pgtbl This parameter is for output, and is a pointer to the page directory that contains this page.
	ptr_t* Map_Vaddr This parameter is for output, and is a pointer to the virtual address of the page.
	ptr_t* Paddr This parameter is for output, and is a pointer to the physical address of the page.
	ptr_t* Size_Order This parameter is for output, and is a pointer to the size order of the page directory.
	ptr_t* Num_Order This parameter is for output, and is a pointer to the number order of the page directory.
	ptr_t* Flags This parameter is for output, and is a pointer to the permission flags of the page.

This function needs to perform lookups according to the virtual address passed in on the (possibly path-compressed) page table, and check whether the page is in the table. If it is, the function can output the page directory address, and its position. This function should only perform lookup on user memory; if a kernel lookup is attempted, an error must be returned.

The hardware-specific page flags need to be converted to RME standard flags before outputting them. Also, all six output parameters are optional. If only a subset is needed, just pass 0 (NULL) into other parameter to suppress their output.

7.12 Writing Interrupt Vectors

There are two types of interrupts in RME besides system call interrupt, system timer interrupt and fault handling interrupts. The first type is the transparent interrupts, and you can write them as if you are writing bare-metal applications. No register pushing/popping other than necessary is needed, and they can nest in any order. Thus, the programming of this type of interrupt is usually easy, and they feature a very low interrupt latency. However, this type of interrupt cannot call any function that is built into the kernel, and can only read I/O or some variable, then write to I/O or some other variable. Thus, this type of interrupt is fit for device drivers that require a quick turnaround or have a strict timing. Typical such devices include 1-wire sensors (DS18B20, PGA300, DS2432 and SHT-XX, etc).

The second type is the sensable interrupts. The entry and exit of these interrupts will push and pop the registers according RME interrupt requirements, and they are not allowed to nest with each other. Such interrupts can call some kernel code to send signals to user applications. These interrupts are fit for those user-level drivers that hope to relay interrupts to application and hand off device data to them. Additionally, this type of interrupts may perform context switches. In the two sections that follow, we will concentrate on this type of interrupts, because the first type is no different from bare-metal interrupt vectors.

Whatever type of interrupt a vector is, it is a vital part of the system. A system's security heavily depends on the security of its interrupt vectors. The priority and nestability requirement for these interrupts includes two rules: the transparent interrupts may nest with each other and they must have a higher priority than all the sensable interrupts; the sensable interrupts cannot nest each other, and their priority must be higher than system call interrupt, system timer interrupt and fault handling interrupt.

7.12.1 Entering and Exiting of Interrupt Vectors

The entry and exit of (sensable) interrupts and system interrupts are the same. They both require pushing and popping registers according to the register set structure, and the pointer to the register structure needs to be passed into the interrupt processing function which is usually written in C. All of the prototypes of the interrupt functions in C is as follows:

Prototype	void _User_Handler(struct RME_Reg_Struct* Reg)
Function	Execute sensable interrupt processing.
Return	None.
Parameter	struct RME_Reg_Struct* Reg The registers to be pushed to stack upon interrupt entry.

7.12.2 Callable Kernel Functions in Interrupt Vectors

In (sensable) interrupt, some kernel code can be called to send signal to user processing threads to make it ready, or do some other operations. The list of these functions are as follows:

7.12.2.1 Send to Kernel Endpoint

This function can send to a kernel endpoint. This is the most important function, and can be used to pass sensable interrupts to the user-level. This function can be called multiple times in an interrupt vector, in case you need to send to multiple endpoints.

Prototype	ret_t _RME_Kern_Snd(struct RME_Reg_Struct* Reg, struct RME_Sig_Struct* Sig_Struct)	
Parameter	Type	Explanation
Reg	...	Its type is struct RME_Reg_Struct*, and it is a pointer to the register set. This is passed in from the interrupt processing function.
Sig_Struct	...	Its type is struct RME_Sig_Struct*, and it is a pointer to the signal endpoint. A signal will be sent to this endpoint.

The possible return values of this function are:

Return	Explanation
0	Operation successful.
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.

The difference between this function and a normal one is, if a high priority thread gets woken up in the send, we will just set this thread as ready and will not immediately perform a context switch. The reason for this is that we may send to many endpoints in an interrupt, and if the priority of threads escalates one by one, postponing the switch to the end of the interrupt handler might be a better idea. The description about the function to call upon interrupt exiting is in section 7.12.2.2.

7.12.2.2 Perform Context Switch upon Interrupt Exit

This function is for performing the final thread switch upon interrupt exiting. This function must be called at the end of any interrupt handler that called _RME_Kern_Snd.

Prototype	void _RME_Kern_High(struct RME_Reg_Struct* Reg, ptr_t CPUID)	
Parameter	Type	Explanation
Reg	...	Its type is struct RME_Reg_Struct*, and it is a pointer to the register set. This is passed in from the interrupt processing function.
CPUID	ptr_t	The CPUID of the current CPU.

This function does not have a return value.

7.12.2.3 Increase Value of RME_Timestamp

This function increases the value of RME_Timestamp for a few timeslices, and is mainly used in tickless kernels to update the system timestamp counter. It is worth noting that we just need one core to update this value.

Prototype	ptr_t _RME_Timestamp_Inc(cnt_t Value)	
Parameter	Type	Explanation
Value	cnt_t	The value to increase. Must be non-negative, or the kernel will panic.

This function returns the value of timestamp before it is increased.

7.13 Explanations for Other Functions

When writing drivers and debugging kernel code, there are some helper functions out there that you can make use of. The kernel provides these functions, so that they are compiler and C runtime independent. All prototypes of these functions are located in kernel.h; include kernel.h if you want to use them. A list of them is as follows:

7.13.1 Variable Clearing

This function can clear a memory region in the kernel. This function is a equivalent of C library memset when the filled data is 0.

Prototype	void _RME_Clear(void* Addr, ptr_t Size)	
Parameter	Type	Explanation
Addr	void*	The start address of the area to clear.
Size	ptr_t	The number of bytes to clear.

7.13.2 Comparing Memory Segments

This function can compare whether the two memory segments are equal. This function is a equivalent of C library memcmp.

Prototype	ret_t _RME_Memcmp(const void* Ptr1, const void* Ptr2, ptr_t Num)	
Parameter	Type	Explanation
Ptr1	const void*	The first segment to compare.
Ptr2	const void*	The second segment to compare.
Num	ptr_t	The number of bytes to compare.

If the two memory segments are equal within given number of bytes, 0 will be returned; if not, a non-zero value will be returned.

7.13.3 Replicating Memory Segments

This function can copy a memory segment to another. This function is a equivalent of C library memcpy. The two memory segments shall not overlap, or the behavior of the function is undefined.

Prototype	void _RME_Memcpy(void* Dst, void* Src, ptr_t Num)	
Parameter	Type	Explanation
Dst	void*	The destination address to copy to.
Src	void*	The destination address to copy from.
Num	ptr_t	The number of bytes to copy.

It is worth noting that the three function listed in 7.13.1-7.13.3 are naive implementations, and no optimization of any kind is considered, so it is best to avoid large operations. This is for maximum compatibility and efficiency (some architectures have alignment requirements about word-level operations; or they need special instructions to reach a higher speed; or the compiler built-in functions will use FPU registers. All three need to be avoided in the kernel). The architecture agnostic portions of RME don't use these functions; avoid large memory operations in hardware abstraction layer if possible as well. If large memory operations are required, then you can either write your own version or use the compiler supplied one. Anyway, the implementation must avoid using FPU registers, or causing memory access alignment errors. Special care needs to be taken if you use compiler library functions.

7.13.4 Printing Signed Integers

This function can print a signed integer in its decimal (10) format. The print is blocking, and the function will return after the print is complete. The string printed includes the sign.

Prototype	cnt_t RME_Print_Int(cnt_t Int)	
Parameter	Type	Explanation
Int	cnt_t	The signed integer to print.

The return value of the function is the number of characters printed.

7.13.5 Printing Unsigned Integers

This function can print an unsigned integer in its decimal (16) format. The print is blocking, and the function will return after the print is complete. The string printed does not include the "0x" suffix, and A-F are all upper case.

Prototype	cnt_t RME_Print_Uint(ptr_t Uint)	
Parameter	Type	Explanation
Uint	ptr_t	The unsigned integer to print.

The return value of the function is the number of characters printed.

7.13.6 Printing Strings

This function can print a string. The print is blocking, and the function will return after the print is complete.

Prototype	cnt_t RME_Print_String(s8_t* String)	
Parameter	Type	Explanation
String	s8_t*	The string to print.

The return value of the function is the number of characters printed, "\0" not included.

Bibliography

None

Chapter 8 Appendix

8.1 Supporting Special Functionality in RME

RME can support many special functionality that are provided by some other operating systems, such as CPU hotplug, memory hotplug and separation kernel. The implementation intuitions are described below.

8.1.1 CPU Hotplug

CPU hotplug can be devided into two categories: CPU hot installation to increase the number of CPUs and CPU hot removal to reduce the number of CPUs. RME's support of these functions relies on the low-level hardware platform support, and must be implemented with kernel function calls.

For hot installation, we can initialize the processor after plug-in, and create the Init thread on these processors. After this, these processor cores can be used.

For hot removal, it is just the opposite. We need to stop the processors, and make sure no IPIs are sent to these processors anymore before they get removed.

8.1.2 Memory Hotplug

Memory hotplug can also be divided into two categories: memory hot installation and memory hot removal. RME's support of these functions also relies on the low-level hardware platform support, and must be implemented with kernel function calls.

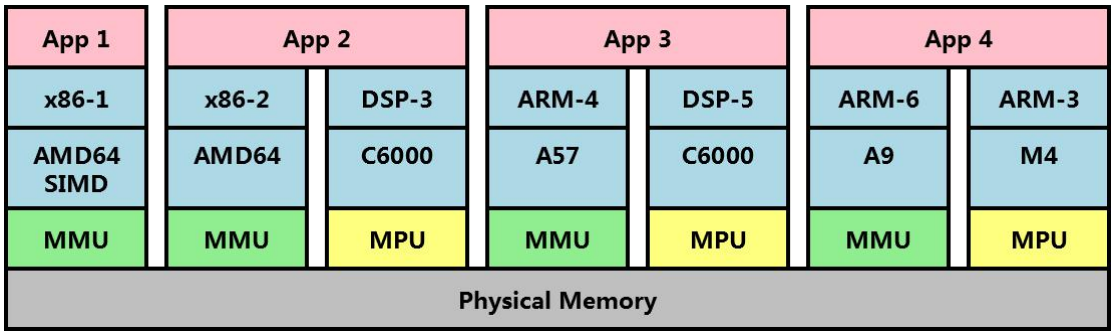
For hot installation, if we are adding user memory, we just need to add the physical frames of these memory pages into some user page directory. If more kernel memory is desired, we need to pause the execution of all other processors, and modify the kernel page table to reflect this change. After all these operations, the execution can be resumed, and a TLB flush needs to be performed.

For hot removal, if we are reducing the available user memory, we must make sure that these memory pages are not mapped. If this is true, we can delete these mappings directly and unplug the memory. If we are reducing available kernel memory, we need to pause the execution on all other processors and copy the data on these physical pages onto empty pages. We also need to modify the kernel page table to reflect these changes, then resume execution after that. A TLB flush is also required.

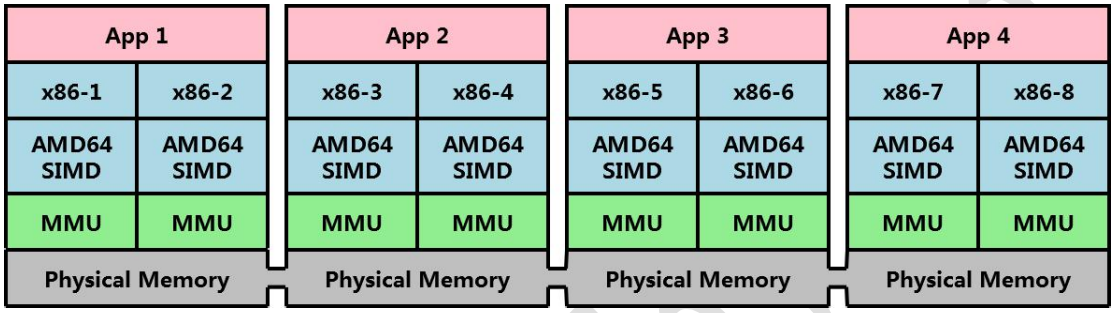
8.1.3 Separation Kernel

Due to the fact that RME is a microkernel, we can run many instances of RME on the same physical machine, just like Barrelfish[1]. Each instance can manage one or more CPU, and the memory sharing and network communication is implemented at user-level. This configuration does not require cache coherency, and does not even require the same instruction set and architecture. If processors of different architectures are used in such a system, porting of RME and user-level library must be performed for each architecture.

As an example, if we have a System-on-Chip (SoC), we can run RME on all cores, and share memory between different subsystems to pass information between different instances of RME.



The picture above shows a instance where different architectures run together. It is also possible to run a instance of RME on every NUMA node of a machine, as shown below:



8.2 Afterwords

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

In RME, not all operations are completely scalable. A typical inherently unscalable operation is that multiple CPUs try to create kernel objects in the same kernel memory location. The unscalable operations are listed below, accompanied by their explanation.

8.2.1.1 Update of RME_Timestamp

The update of RME_Timestamp is performed by a single processor. In multi-core environments, all other processors needs to synchronize this variable to their own cache. Thus, this operation is not completely scalable. However, this is not a problem, because timer interrupt frequency can be configured to be very low on large systems.

8.2.1.2 Multiple CPUs Try to Create Kernel Object at the Same Location

Due to the fact that RME uses kernel object registration table to track kernel object creations, when multiple CPUs try to write to the same location, a huge amount of cache line bouncing can occur, thus making the operation unscalable.

8.2.1.3 Multiple CPUs Try to Send to the Same Signal Endpoint Simultaneously

Due to the fact that this is a compare-and-swap operation and the bus might be locked, the fetch-and-add is in fact performed in a serial manner, and thus is unscalable. However, if your system features mechanisms to execute additions in a parallel manner (which is the case for most modern architectures), this is not a problem.

8.2.2 Restrictions of RME on 32-bit Processors

Due to processor word length limitations, 32-bit RME implementations have a few functionality restrictions. The two restricted functionality is thread creation and quiescence calculation.

8.2.2.1 Thread Creation

Each thread in RME is associated with a TID. The size of TID is always a machine word, and two bits in it are used for other purposes. Thus, in 32-bit systems, we can only have 2^{30}-1 distinct TIDs (in 64-bit systems this value is 2^{62}-1 and is certainly fine). This means that, if we have more than this number of threads, there will be two threads that have the same TID when binded to a certain processor. If the user-level is not designed to cope with this, then there could be a problem.

However, in common 32-bit systems this is not a problem. The majority of 32-bit systems are embedded systems, and seldom have this number of threads. When this is the case, the user-level can always find an unused TID for the new thread.

8.2.2.2 Calculation of Quiescence Period

All quiescence time in the system are calculated with reference to RME_Timestamp. This value will increase at each timer tick. The length of RME_Timestamp is always a machine word, thus there are some wraparound problems.

For example, when the value is 0x00000000, an operation (e.g. freeze a capability) is performed, and it needs 10 timeslices to reach quiescence. When the value of RME_Timestamp exceeds 0x0000000B, further operation can be performed. However, if we did not do any further operation for long enough, and waited until the counter goes back to 0, we will find that the capability is no longer quiescent again. In fact, this position is already quiescent, and such illusion is caused by counter overflow.

Thus, in 32-bit multi-core systems, if the object is not quiescent after its quiescence period, we can try again after another quiescence period, until we are successful. In 64-bit systems, the variable RME_Timestamp never overflows, and this problem will not happen.

8.2.3 Known Potential Covert Channels in RME

There are some known covert channels in RME. They can be divided into two categories, as listed below.

8.2.3.1 Covert Storage Channels

The covert storage channels in RME mainly exist between two processes that share capabilities, kernel objects and capability table. If capabilities or kernel objects are shared, two processes can probe the state of the capabilities or kernel objects and change their state to pass information between each other.

Additionally, in systems that feature a coprocessor, if the coprocessor context save and restore routing is designed incorrectly, the coprocessor register set can be used as a covert storage channel with a massive bandwidth. A wrong kernel function implementation can also cause this problem, and its design also needs to be handled with care.

8.2.3.2 Covert Timing Channels

In RME system, the timeslice and scheduling is managed by the user-level. Thus, if the user scheduling algorithm or timeslice allocation algorithm is not appropriate, many covert timing channels

can appear. To reduce the bandwidth of such channels, banning user-level high-precision counter instructions (such as RDTSC on x86-64) is a feasible solution. We will not dive into the details of this issue here.

8.2.4 Memory Consistency Model

RME largely employs lock-free algorithm, and is thus susceptible to memory consistency model. Generally speaking, there are two categories of memory models: strongly ordered and weakly ordered. In a strongly ordered model, all memory accesses from multiple cores have a global sequence; i.e., from all processor's view, the other processor's behavior is the same across all memory operations. In a weakly ordered model, however, not only one processor's view of other processors' behavior is out-of-order, but also two processors' view of the third processor's memory access order different.

In a weakly ordered processor, four types of reordering can happen:

Read-read reordering, i.e., adjacent read operations can be reordered.

Read-write reordering, i.e., adjacent read-write operations can be reordered.

Write-read reordering, i.e., adjacent write-read operations can be reordered.

Write-write reordering, i.e., adjacent write operations can be reordered.

In fact, there's a continuum of spectrum between the two extreme models. Some processors maybe unordered with respect to write-read pairs, but all other pairs may be ordered (x86-64).

Memory reordering can cause some synchronization flags to go awry. Generally speaking, there's a flag guarding each shared memory region. The state of the flag represents the current state of the shared memory region. We usually check and try to take the flag first before we operate on the shared memory region; after the operation finishes, we will clean up the job first before we release the flag. When there's memory reordering, shared memory operations may be ongoing before the flag is even taken; the flag may also be released before the job clean-up is complete. These situations break the consistency of the shared memory region. Thus, a memory barrier should be added to guarantee that operations on shared memory will not start until the flag is taken; a memory barrier should also be added before the job clean-up to guarantee that the flag will be released after. RME does not make use of locks in the kernel, however, memory ordering still impacts its kernel synchronization.

To eliminate the effect of memory reordering on kernel operation, RME provided two macros, namely RME_READ_ACQUIRE() and RME_WRITE_RELEASE() to synchronize memory accesses. Also, atomic Compare-And-Swap(CAS) implementation is required to have a full memory barrier effect. Please refer to section 7.3.2 and 7.7 for detailed implementation guide.

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.