KKM Design

Overview

Kontain Monitor(KM) is a unikernel container process execution engine. KM runs each process in its own virtual jail to provide isolation. Current implementation of KM uses KVM API for virtualization services.

Currently aws does not provide nested hardware assisted virtualization. Kontain Kernel Module(KKM) is developed to provide software virtualization for KM when hardware assisted virtualization is not available. KKM is not a general purpose virtualization driver. It is targeted at exactly the functions that KM needs. KKM does this by implementing a micro kernel per KM running thread(kontext). When KKM is running in payload mode(container application) a small footprint of code required to handle forwarding requests/services to appropriate executives is mapped.

KKM shares the physical CPU's and other physical hardware resources with native linux kernel. KKM does not provide any IO device support(network, storage ...). To be precise KKM provides only services needed by KM.

KKM provides an interface similar to KVM to create/destroy VM's(kontainer), VCPU's(kontext) memory slots. The interface consists of a collection of `ioctl()` calls. Each KM+KKM container has one instance on kontainer with several kontext’s and several memory slots. Idea is similar to KVM to keep KM from having to deal with multiple
virtualization architectures. Minimal sets of KVM ioctls used by KM are implemented by KKM.

KKM provides services to the following: KM, payload and hardware. KKM forwards requests to the following executives: system call, memory management, interrupts, debugging and other faults.

KKM interfaces with the following components

- KM
- Payload
- Hardware
- Paging infrastructure(linux kernel)
- Interrupt handling
- Interrupt forwarding

Source code layout

C code

Most of the microkernel code is written in C. The following are the important files

- kkm_main.c -- All the ioctl entrypoints
- kkm_kontainer.c -- kontainer specific
- kkm_kontext.c -- kontext specific
- kkm_mmu.c -- page table management
- kkm_idt_cache.c -- idt setup

Assembly code

Some of the functionality required is implemented in assembly language to make coding easier than sticking to strict C.

- kkm_guest_entry.S -- forward path linux-kernel to payload
- kkm_guest_exit.S -- return path from payload to linux-kernel
- kkm_intr.S -- forwarding physical interrupts to linux kernel and return path
- kkm_fpu.S -- cpu extended register save and restore

Source Code Formatting

Coding is a bridge between Kontain coding guidelines and linux kernel guidelines. To make it easier to conform, ".clang-format” file is committed to the source tree.

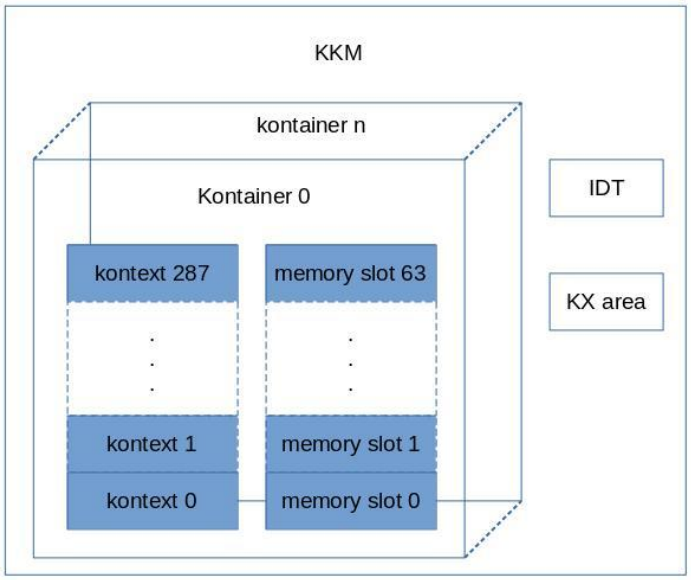
Testing

Testing is done using KM tests and running payloads such as java and python.

Interfaces

KM interface

KKM provides an interface that is compatible with KVM. KKM provides kontainer's and kontext's and memory slots(very similar to Virtual Machine, VCPU and memory slots in KVM terminology).



KKM driver creates anonymous fd for resources that need actions to be performed. (i.e) each kontainer and kontext have their own anonymous inode. KM uses this inode(fd) to communicate to KKM.

All structures are implemented to be compatible with KVM. Significant portion of the state is maintained by the KKM module to be compatible with the physical CPU.

KKM interface

Primary interface with KKM is a device node named /dev/kkm. It provides the following ioctl commands

- KKM_GET_VERSION
  - Return version of the driver

- KKM_CREATE_KONTAINER
  - Create a new kontainer, and return the fd for the kontainer. All operations on the kontainer use this fd.

- KKM_CHECK_EXTENSION
  - Currently KKM_CAP_SYNC_REGS is implemented. Current sub options are KKM_SYNC_X86_SREGS and KKM_SYNC_X86_REGS.

- KKM_GET_CONTEXT_MAP_SIZE
  - Return context map size. This is required for the shared state between KKM and monitor.

- KKM_GET_SUPPORTED_CONTEXT_INFO
  - Equivalent to CPUID.

Kontainer interface

The fd returned by KKM_CREATE_KONTAINER is the handle for all actions on Kontainer(VM). Each kontainer has multiple execution kontext's, multiple segments of memory and shared memory area for communication between monitor and KKM.

- KKM_ADD_EXECUTION_CONTEXT
  - Create a kontext and return controlling fd

- KKM_MEMORY
  - Add a memory slot to kontainer.

- KKM_SET_ID_MAP_ADDR
  - Implemented for compatibility. Not used.

Kontext interface

The fd returned by KKM_ADD_EXECUTION_CONTEXT is the handle for all actions on a kontext(VCPU). This is the controlling entity for running payload. Each payload pthread has one Kontext.

- KKM_RUN
  - Start payload execution with the cpu context set by other ioctls.
- KKM_GET_REGS
  - Get the current register state. On x86 16 GP registers, rip and rflags.
- KKM_SET_REGS
  - Set register state for the current payload. On x86 16 GP registers, rip and rflags.
- KKM_GET_SREGS
  - Get the current system register state. On x86 segment registers, tr, ldt, gdt, idt control registers and so on.
- KKM_SET_SREGS
  - Set the current system register state.
- KKM_SET_MSRS
  - Dummy call to maintain compatibility. None of the CPU MSRS are set when running this ioctl.
- KKM_GET_FPU
  - Get current FPU state.
- KKM_SET_FPU
  - Set current FPU state.
- KKM_SET_CPUID
  - Dummy call to maintain compatibility.
- KKM_SET_DEBUG
  - Set debug registers. Used by debuggers.
- KKM_KONTEXT_REUSE

        - KM reuses the kontext when pthreads are recycled. Communicate to KKM the context state needs to be invalidated.

    - KKM_KONTEXT_GET_SAVE_INFO
        - Get the current driver state for this kontext. Used to save and restore state across signals and snapshots.

    - KKM_KONTEXT_SET_SAVE_INFO
        - Set the current driver state for this kontext.

    - KKM_KONTEXT_GET_XSTATE
        - Get current cpu extended registers. FPU, MMX, XMM, YMM registers. Used to save state across the signal handler.

    - KKM_KONTEXT_SET_XSTATE
        - Set current cpu extended registers. FPU, MMX, XMM, YMM registers.

Kontext has a per-cpu shared mmap area to share data between monitor and KKM. The format of the shared memory area is dependent on the current exit reason.

Workflow of a simple thread.

- Set the system state using SET_SREGS and SET_MSRS.
- Set the current cpu state using SET_REGS, SET_FPU or SET_XSTATE.
- Run KKM_RUN.
    - KKM_RUN will execute instructions of payload with the given registers, until the payload encounters a fault or system call. At this point control is transferred to KKM. KKM microkernel forwards control to appropriate executives.

Payload interface

There are two synchronous exits from payload that are both for getting a system call processed.

syscall instruction

SYSCALL instruction is used for control transfer from user mode programs to request services from the kernel. Glibc and alpine native libraries use syscall instruction to trigger a system call.

Before starting payload code KKM saves native kernel MSR_LSTAR register value and replaces it with KKMs syscall handler address.

When payload executes SYSCALL instruction, the x86 processor transfers control to KKMs syscall handler. On x86 processor system call arguments are passed as register values. KKM copies syscall arguments(hc_args) to thread stack. Copies and the address of hc_args to per thread hc_args pointer area. And finally transfers control back to KM.

Once KM completes processing of the payloads system call request it requests KKM to start running the payload.

KM puts system call return value in hc_args.

KKM copies the system call return value to rax and undoes the modifications to thread stack adjustments.(removes hc_args from thread stack)

out instruction

Kontain's customized musl library uses OUT instruction as a way to get services from the linux kernel. Syscall requested in this way is transparent to KKM.

OUT instruction is privileged in x86. The x86 processor generates a general protection fault when OUT instruction is executed in PL 3.

When a general protection fault handler is invoked, KKM decodes the instruction at the fault address. If the decoded instruction is OUT, KKM set’s up necessary information for KM to process hypercall and transfers control to KM.

Exit reason is tracked per kontext in the shared memory area with monitor, and internal data structures of KKM. This information is used when KM returns control back to payload to undo hypercall setup..

Memory handling

KKM currently supports 4 level page tables. It allows for 512GB of virtual space per kontainer. There are several types of memory handled by KKM

- virtual memory text, data and heap
- vdso and vvar
- kx area

Virtual Memory

KM manages the payload's physical memory. Currently KM supports 512GB of memory for payload. This consists of payload code, stack and heap. KM needs to be able to access this memory for processing system calls. The physical memory provided to payload is mapped to KM virtual space when running in monitor mode starting at 16TB. Currently payload code virtual space is 0 to 512GB for text and data, managed by `brk()`, and 128TB-512GB to 128TB for mmap.

Memory aliasing

When running in payload mode KM/KKM maintain a 1-1 mapping between virtual and physical address. All memory is aliased at 2 virtual addresses.

- 0 .. 512GB physical address is mapped to 0 .. 512GB virtual address.
- 0 .. 512GB physical address is mapped to 128TB-512GB .. 128TB virtual address.

When running in monitor mode same physical memory is accessible as monitor virtual address

- 0 .. 512GB physical address is mapped to 16TB .. 16TB + 512GB virtual address.

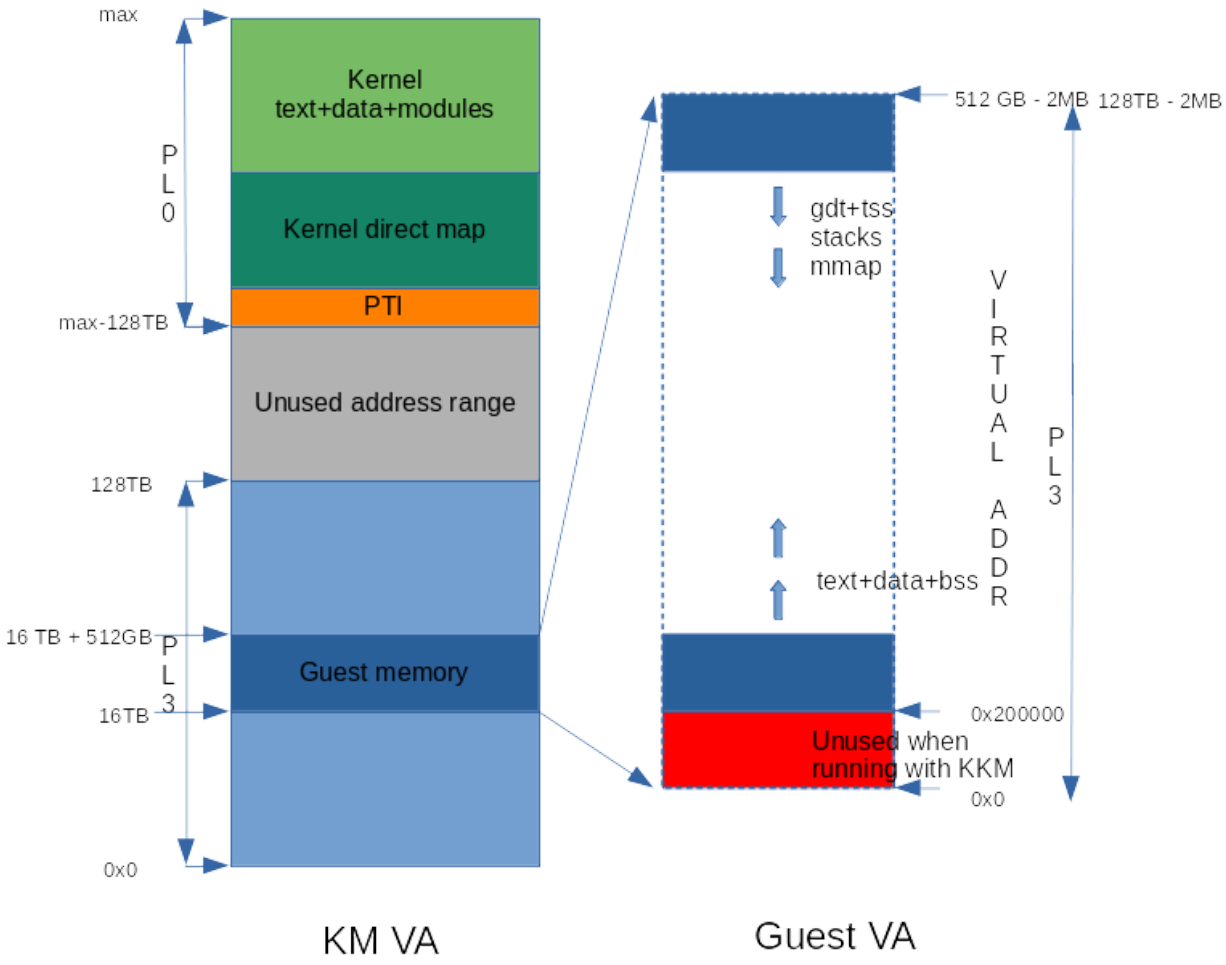
Aligned aliasing

Linux kernel manages page mapping for KM. When fault in happens a new PTE entry is added and when memory is under pressure existing PTE are removed. To manage the aliasing efficiently KM/KKM chose the start address of virtual memory to be a PML4 entry beginning. Each PML4 entry itself covers 512GB of virtual address mapping.

By using such an address range the entire KKM payload virtual memory management becomes simply copying one PML4 entry from KM(entry 32) to 2 entries in payload level 4 Page directory(entry 0 and entry 255).

Now any page faults in payload can be directly computed and faulted in KM.

Address mapping between KM and Payload



VDSO and VVAR

This is the area where some of the high frequency system call code is placed. This area keeps changing between each process. Page faults in payload VDSO and VVAR have to be correctly forwarded to the relevant VDSO and VVAR area for current monitor. This is handled by assigning a well known memory slot and is used to translate correctly.

KX area

This is where the KKM micro kernel is mapped. kx area consists of

- guest entry
- guest exit
- per cpu private area

Page tables are created to have the same address mapping when running in kernel and payload mode. This allows us to switch page tables and continue executing the same code.

Guest entry

Payload entry code is mapped to this area.

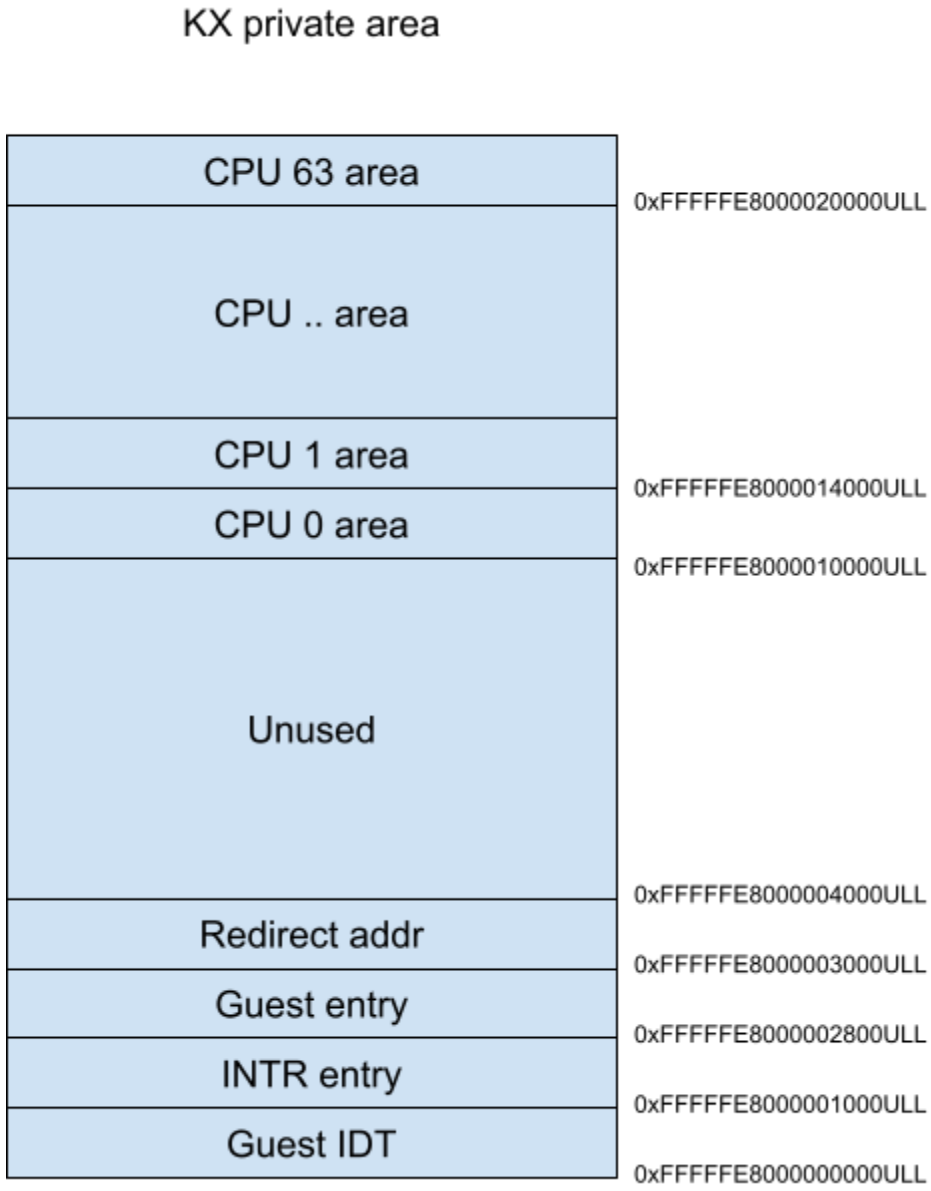
Guest exit

This is the code that executes when payload wants to request services or async service handling is needed. IDT is the entry point for all the faults when running payload. Each IDT entry points to a separate handler. These handlers forward to a common handler.

Per cpu private area

KKM assigns four pages per physical cpu. This is memory where we keep the current running kontext information. Once exited from guest payload to guest kernel mode this area allows us to identify the current thread and restore normal kernel state.

Information required during guest kernel and linux kernel is mapped to this area. When running in linux kernel mode PTE's for the current CPU are filled in.



TLB Handling

KKM shares all cpu resources with the linux kernel. This includes all the TLB caches. All the processes in the system share the same TLB cache. Kernel uses PCID's to avoid constant flushing of TLB when a context switch is required. Current algorithm uses few PCID's and rotates them when a process is switched in or out.

KKM guest address mapping is different from monitor address mappings. The TLB need to be invalidated on every payload entry and exit. This is very inefficient. KKM implements an algorithm that works with Linux kernel TLB flush/recycle algorithm.

KKM uses 2 PCID's(unused by the kernel at this point) one for the guest kernel and one for guest payload. This way we only need to flush TLB when a process context switch happens. KKM yields control to native kernels under certain conditions. We cannot predict a process context switch on the same CPU or even migrate to a different CPU.

KKM TLB flush algorithm.

Each kontext is assigned a unique id across the lifetime of the kkm module. Each kontext maintains the last cpu it was run on and each physical cpu tracks which kontext was last run on it. TLB is flushed when either of the associations have changed.

Interrupt handling

One of the design goals of KKM is zero changes to the base Linux kernel. External modules do not have visibility to kernel data structures or private function names. So the option left over is to replace IDT and find ways to process interrupts and faults.

On x86 interrupt and fault/exception handlers are set up in IDT. KKM installs its private IDT when running in the Guest kernel and Guest payload modes. This is required as some of the faults like GP, PF are handled by KKM.

When a physical interrupt of fault/exception occurs execution control is transferred to KKM private handlers. X86 hardware sets up a stack with necessary information to process appropriate faults. This information is used by KKM common interrupt handling and forwarded to appropriate executives.

Guest payload control transfer

On x86 fragments of code continue to execute when no external event happens. When running payload control transfer happens due to interrupts, faults, exceptions or system calls.

These events can be classified as following on x86 processors.

There are two models of control transfer out of running payload.

- Synchronous

        - Payload requesting services
    - Asynchronous
        - Some other system related activity needs the CPU now.

There are four different categories
- Faults -- async
- Exceptions -- async
- Hardware interrupts -- async
- System Calls -- sync

Faults/Exceptions

x86 processors support 0-255 interrupts. 0-31 are faults/exceptions and 32-255 are hardware/other interrupts. System call is another way of exiting payload. Since the workflow is similar to handling any other interrupt KKM assigns interrupt number 511 to system call.

x86 processor vectors all interrupt using IDT. IDT is an array of entry points with attributes. KKM uses a single copy of IDT for all the processes and cpus.

During initialization KKM saves the native kernel IDT and creates a private IDT. On x86 cpu hardware does not identify the interrupt. There is a separate entry point for each interrupt, which pushes the interrupt number to the stack.

There are two faults/exceptions that are important to KKM
- General Protection Fault
- Page Fault

General protection fault

GP fault can happen for a lot of reasons.Only the one caused by out instruction is of relevance to KKM. This is one of the ways payloads can request system calls. When GP happens we verify the instruction is OUT and set up the shared memory area with required data and return to the monitor to handle system calls.

Page fault

All kontext's in a kontainer have the same memory image. Multiple page faults can happen in the same kontainer with different addresses and reasons or same address and for the same reason.

Page faults can happen

- privilege violation
- read fault
- write fault
- invalid memory access

KKM handles only one page fault at a time in each kontainer(protected by a mutex). This is done to avoid read faults racing with write faults. We cannot access any of the kernel functions to handle page faults and we don't have access to any data structures required to handle page faults.

page fault handling algorithm

We handle page faults using copy from user space and copy to user space services provided by Linux kernel.

- Convert fault address to monitor address.
- Validate the fault address against the kontainer memory slots.
- one byte copy-in for read faults and copy-in followed by copy-out write faults on the faulted address to complete page fault.
- flush tlb to make sure new page translation is now in effect.

Hardware Interrupts

Only a small number of faults/interrupts are processed by KKM at this point. Rest are forwarded to the linux kernel(HW intrs need to be forwarded and processed by the correct driver). The KKM module uses a soft intr model to forward interrupts to the linux kernel.

System Calls

There are 2 types of system calls possible with payloads.

- out instruction
- syscall instruction

Since the payload is running in ring 3, OUT instruction causes a general protection fault. syscall has a different handler installed using MSR's. When syscall instruction is executed by payload

control transfers to the address pointed by MSR_LSTAR. This entry point creates a stack that is similar to a hardware created stack and transfers control to generic handlers.

Processing of certain faults can be completed in the KKM module(page fault) In this case KKM does not return to monitor, instead it continues executing payload from the same/next instruction where fault happened.

Interrupt forwarding

KKM does not have the ability to process all interrupts/faults. KKM forwards unhandled interrupts to the Linux kernel. KKM uses a soft intr model to forward interrupts to the Linux kernel.

int instruction does not have a parameter. So we have 256 intr forwarding functions one for each intr. Once the linux kernel completes processing of interrupt it will return control back to KKM. KKM then returns to running payload.

KKM initialization

At module initialization time KKM does some sanity checks and registers file operations for miscellaneous device node “/dev/kkm”.

KKM creates a page table hierarchy for the kx area. By design Linux kernel memory mapping is the same as Guest kernel mapping with additional KX area mapping. KX area VA range is unused by the linux kernel.

KKM creates an IDT that will be setup when running guest payload. KKM also copies guest entry and guest exit code to kx area.

There is one copy of the kx area page hierarchy and one copy of IDT shared by all kontainer’s and kontext’s.

KKM control flow

KM requests payload start

KM sets up registers and system registers appropriately to start payload and calls KKM_RUN. KKM will take necessary steps to handle different conditions such as

- the first time KKM_RUN is called on this kontext
- entering signal handling, exiting signal handling
- return from system call
- request from payload.

The above step adjusts the payload thread stack pointer and rip.

At this point KKM is running in native kernel mode. KKM sets current threads private guest area to current physical cpu location in KX area(When payload exits KKM will be required to identify the current thread. The only information available at that time is the current cpu area)

KKM detects if there is a payload thread switch on this cpu and does TLB flush if required.

Now KKM saves the current CPU state. This involves saving debug context, extended cpu state, system registers, MSRs. During this process some of the payload state is restored.(state that is not modified by KKM).

KKM saves the stack and general purpose registers and starts the process to switch from native kernel mode to guest kernel mode. In guest kernel mode KKM can see memory as the payload sees. In this mode KKM has KX area setup as expected.

KKM restores the cpu state required by payload. KKM sanitizes flags, restoring debug context if needed. KKM replaces stack pointer 0 in x86 cpu tss area. This will allow us to forward interrupts to native linux kernel(in case the payload exits to process a physical interrupt).

Now KKM will set up the current physical CPU register state to start the payload. This code is written in assembly. At the end of this payload is running.

On x86 process stack winding and unwinding are symmetrical. In KKM certain portions of stack unwinding make implementation hard. This is the reason for changing to a temporary stack and discarding it at the end.

Payload requests system services

Payload exits to guest kernel mode for several reasons.

- System call
- Fault
- Exception
- Physical interrupt

All of these have their separate entry points either from IDT or MSR register setup. The x86 processor saves some information regarding the payload exit. Rest of the information is derived from the entry point. All the entry points to the guest kernel from payload are written in assembly. They all do similar tasks. Save payload register context and system context that needs to be saved immediately.

KKM restores the native linux kernel context and restores native kernel page tables and switches stack to native linux kernel stack before entering the guest kernel mode.

KKM driver starts processing exit reasons based on the information saved during exit from payload. These could be system calls, page fault or hardware interrupt. General flow of these is explained in sections specific to the exit reason.

KKM address spaces and privilege levels

KKM maintains two separate address spaces per process. One for guest kernel(payload micro kernel) and second for guest payload. In effect there are 4 address spaces used by a Linux process running with KKM. Different states of process execution and privileges levels

Kontain Monitor	Ring 3
Linux Kernel	Ring 0
Guest Kernel	Ring 0
Guest Payload	Ring 3

Glossary

Kontainer Is a collection of Kontext's and Memory Slots. All the kontext's in the same Kontainer share the same memory mapping. Similar to the KVM concept of virtual machines.

Kontext Similar to the KVM concept of VCPU.

Memory Slot Provision memory for Kontext. This also creates a mapping to be accessible from KM.

Kontain Monitor KM. Kontain's proprietary unikernel engine.

KX area Virtual memory area which has KKM specific information.