Splitting the Linux Kernel for Fun & Profit

Chris I Dalton cid@hp.com*

HP Labs, Bristol UK

* Work in collaboration with Nigel Edwards and Theo Koulouris @ Hewlett-Packard Enterprise

Splitting the Linux Kernel for Fun & Profit

‘ How to hack a Micro-Kernel interface into Linux’

Splitting the Linux Kernel for Fun & Profit

‘ How to hack a Micro-Kernel interface into Linux’

OR

‘Adding Intra-Kernel protection to Linux using silicon-based Virtualization Extensions’

(Without needing a Hypervisor)

Outline

- Part 1: Motivation for the work
  - If I wanted a secure OS I wouldn't start from Linux…but what if that was all you had?
- Part 2: Background
  - Ways to structure Operating Systems
  - Linux Kernel Weaknesses & Split-Kernel Demo Video
- Part 3: Splitting the Kernel
  - Restructuring Linux using HW Virtualization Support
  - High-level & Some code details
  - Performance & Invasiveness
- Part 4: Current Status, Opportunities & Futures
  - Open Source Links

Outline

- Part 1: Motivation for the work
  - If I wanted a secure OS I wouldn't start from Linux…but what if that was all you had?
- Part 2: Background
  - Ways to structure Operating Systems
  - Linux Kernel Weaknesses & Split-Kernel Demo Video
- Part 3: Splitting the Kernel
  - Restructuring Linux using HW Virtualization Support
  - High-level & Some code details
  - Performance & Invasiveness
- Part 4: Current Status, Opportunities & Futures
  - Open Source Links

Outline

- Part 1: Motivation for the work
  - If I wanted a secure OS I wouldn't start from Linux…but what if that was all you had?
- Part 2: Background
  - Ways to structure Operating Systems
  - Linux Kernel Weaknesses & Split-Kernel Demo Video
- Part 3: Splitting the Kernel
  - Restructuring Linux using HW Virtualization Support
  - High-level & Some code details
  - Performance & Invasiveness
- Part 4: Current Status, Opportunities & Futures
  - Open Source Links

Outline

- Part 1: Motivation for the work
  - If I wanted a secure OS I wouldn't start from Linux…but what if that was all you had?
- Part 2: Background
  - Ways to structure Operating Systems
  - Linux Kernel Weaknesses & Split-Kernel Demo Video
- Part 3: Splitting the Kernel
  - Restructuring Linux using HW Virtualization Support
  - High-level & Some code details
  - Performance & Invasiveness
- Part 4: Current Status, Opportunities & Futures
  - Open Source Links

Part 1: Original motivation for the work

- Containers offer alternative to using Hypervisors for SW deployments
  - Docker, CoreOS / Rkt, etc

Original Motivation for the Work

- Containers offer alternative to using Hypervisors for SW deployments
  - Docker, CoreOS / Rkt, etc
- Upside: Lighter-weight than using VMs
  - Only one underlying OS to manage
  - Plus better integration opportunities

Original Motivation for the Work

- Containers offer alternative to using Hypervisors for SW deployments
  - Docker, CoreOS / Rkt, etc
- Upside: Lighter-weight than using VMs
  - Only one underlying OS to manage
  - Plus better integration opportunities
- Downside: Shared ‘host’ kernel a significant vulnerability
  - Currently Linux has no ‘**intra-kernel**’ protection
  - All bets are off if you manage to get into the kernel

Getting into the kernel is not that hard!

E.g. Malware triggering buffer overflows / stack / heap attacks, un-authorized module loading, User-space kernel hijack via code & data redirection (ROP), DMA attacks, etc. even with SMAP / SMEP / PXN support

And of course more recent Spectre / Meltdown / L1TF attacks

What did we want to do?

What did we want to do?

Reduce the consequences of a Kernel compromise by introducing a degree of **Intra-kernel** Protection into the Linux Kernel

How?

How?

Restructure kernel into Outer and Inner region based on MMU access

Inner Region can access the MMU directly

Outer Region needs to go through a virtual MMU interface to modify page mappings, etc.

Inner/Outer region separation enforced through CPU HW support for virtualization

Related Research

- ‘Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation’ (Dautenhahn, et al.)
  - Looks at implementing a vMMU for FreeBSD
  - Relies on privileged instruction removing from kernel binary and code scanning for enforcement not VT-x extensions
- ‘Dune: Safe User-level Access to Privileged CPU Features’ (Belay et al.)
  - Uses Intel Vt-x extensions to safely expose HW to user-space processes (e.g. each process has access to cpu rings 0-3)
- ‘Address space isolation Inside the Linux Kernel’ (Rapoport, et al 2019)
  - Tries to achieve similar properties and capabilities to our work
  - Uses restricted page table mappings and kernel direct map modifications not VT-x extensions

What does this buy us?

- Strong control over the integrity of kernel code and core data
  - No ‘un-authorized’ code gets to run in kernel mode
  - Can protect the integrity of data even against malicious kernel mode code
- Can offer confidentiality guarantees
  - Can protect application secrets even against malicious kernel mode code
- Guard against cross-process code + data compromise
  - Enhanced risk when running multiple ‘isolated’ containers

What does this buy us?

- Strong control over the integrity of kernel code and core data
  - No ‘un-authorized’ code gets to run in kernel mode
  - Can protect the integrity of data even against malicious kernel mode code
- Can offer confidentiality guarantees
  - Can protect application secrets even against malicious kernel mode code
- Guard against cross-process code + data compromise
  - Enhanced risk when running multiple ‘isolated’ containers

What does this buy us?

- Strong control over the integrity of kernel code and core data
  - No ‘un-authorized’ code gets to run in kernel mode
  - Can protect the integrity of data even against malicious kernel mode code
- Can offer confidentiality guarantees
  - Can protect application secrets even against malicious kernel mode code
- Guard against cross-process code + data compromise
  - Enhanced risk when running multiple ‘isolated’ containers

Constraints

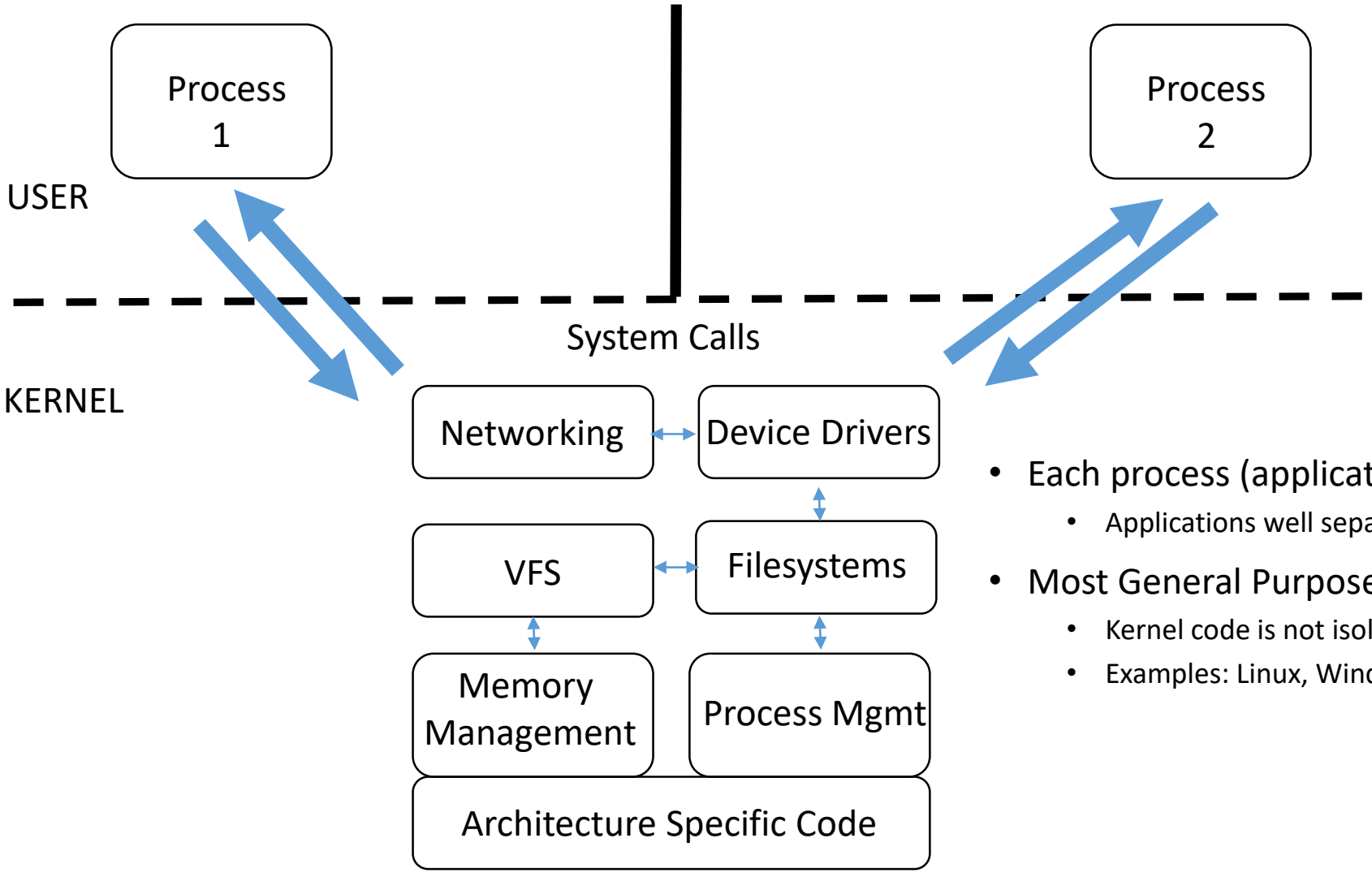
- What no Hypervisor??
  - Strategic control point for others in the Industry (Dell, VMWare, etc)
- Is it still Linux?
  - Can’t afford long-term engineering support
  - Needs to be upstream-able
  - Minimize performance overhead c.f. Hypervisors
  - Minimize intrusiveness c.f. L4Linux
  - But still has to offer significant security improvements over ‘normal’ Linux

Constraints

- What no Hypervisor??
  - Strategic control point for others in the Industry (Dell, VMWare, etc)
- Is it still Linux?
  - Can’t afford long-term engineering support
  - Needs to be upstream-able
  - Minimize performance overhead c.f. Hypervisors
  - Minimize intrusiveness c.f. L4Linux
  - But still has to offer significant security improvements over ‘normal’ Linux

Part 2: Operating Systems Background

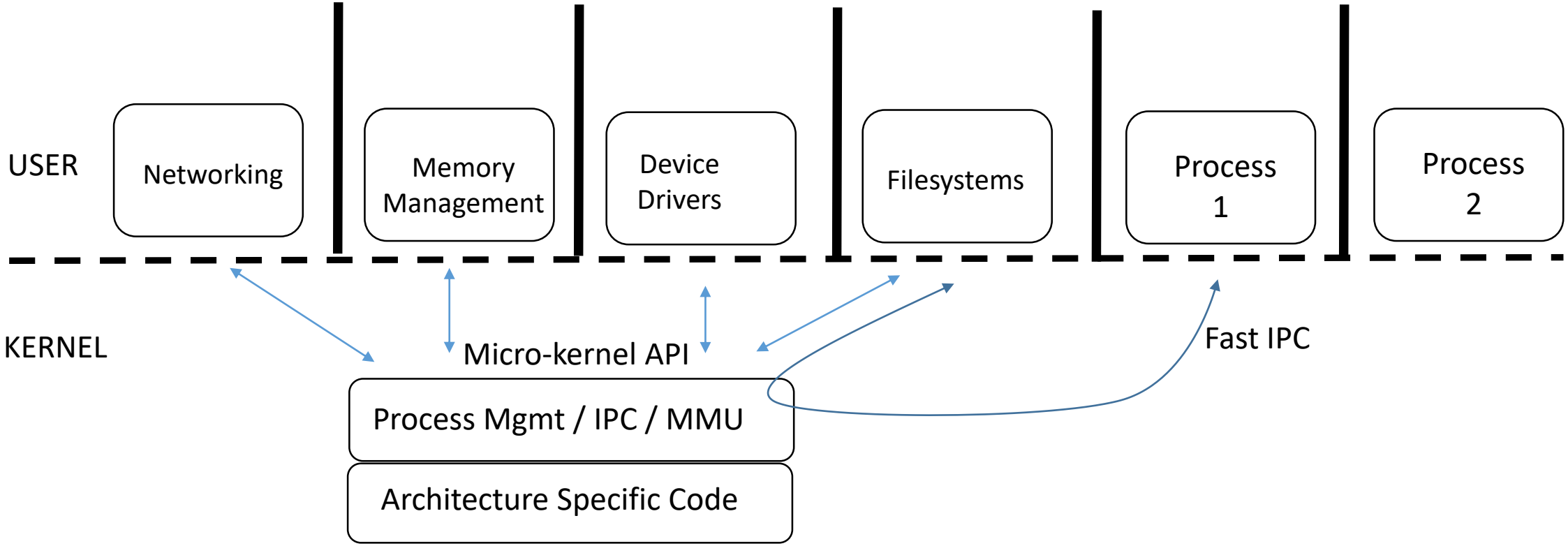
(Monolithic) Operating Systems



- Each process (application) has its own isolated space
  - Applications well separated
- Most General Purpose Operating Systems are ‘Monolithic’
  - Kernel code is not isolated from itself
  - Examples: Linux, Windows

Diagram adapted from Mauerer.

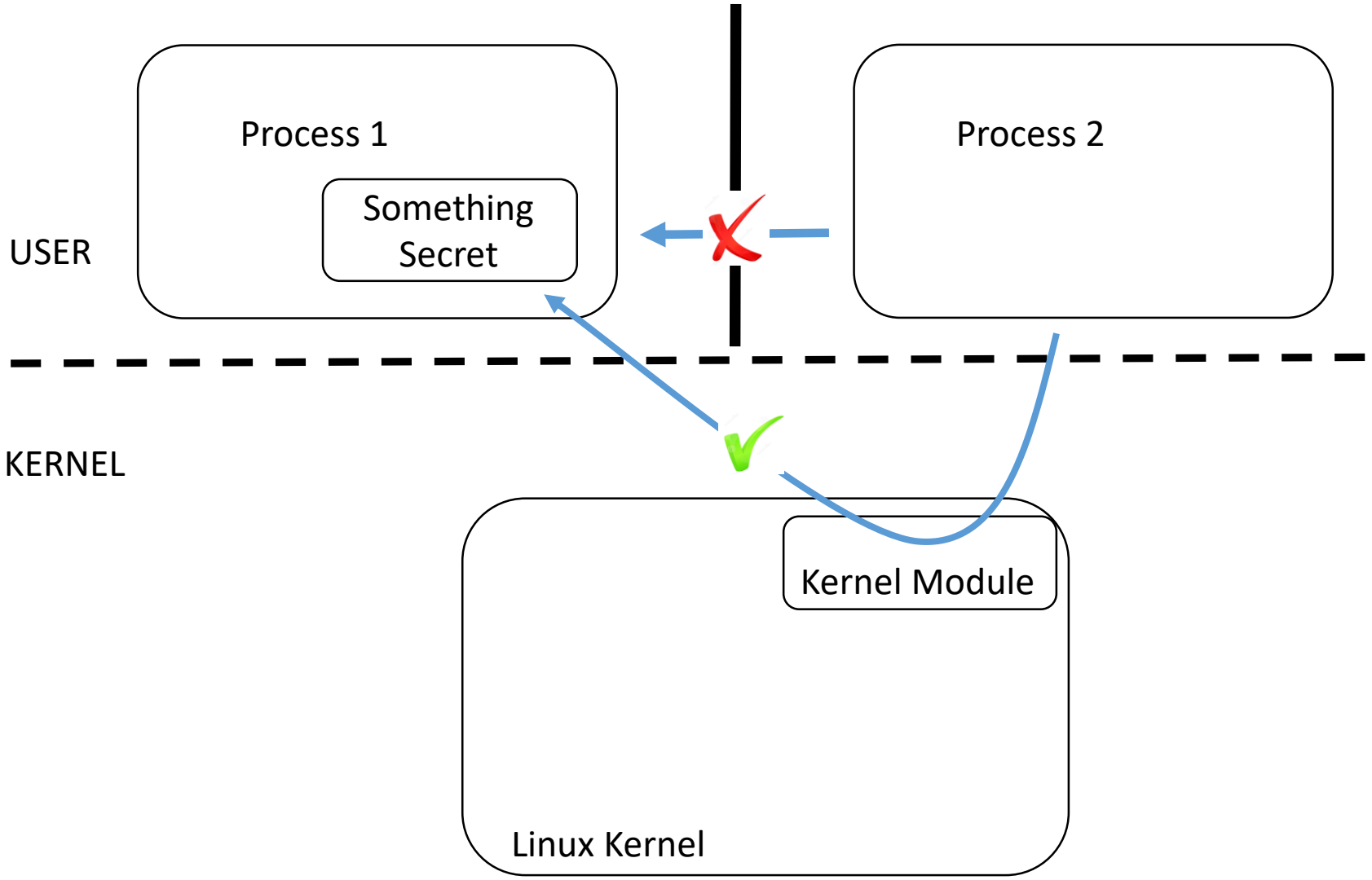
(Micro-Kernel) Operating Systems



- Examples: Fiasco.OC (L4 Micro-kernel), Composite
- Google Fuscia/Zircon

Part 3: Monolithic Linux Kernel (In-)Security Demo

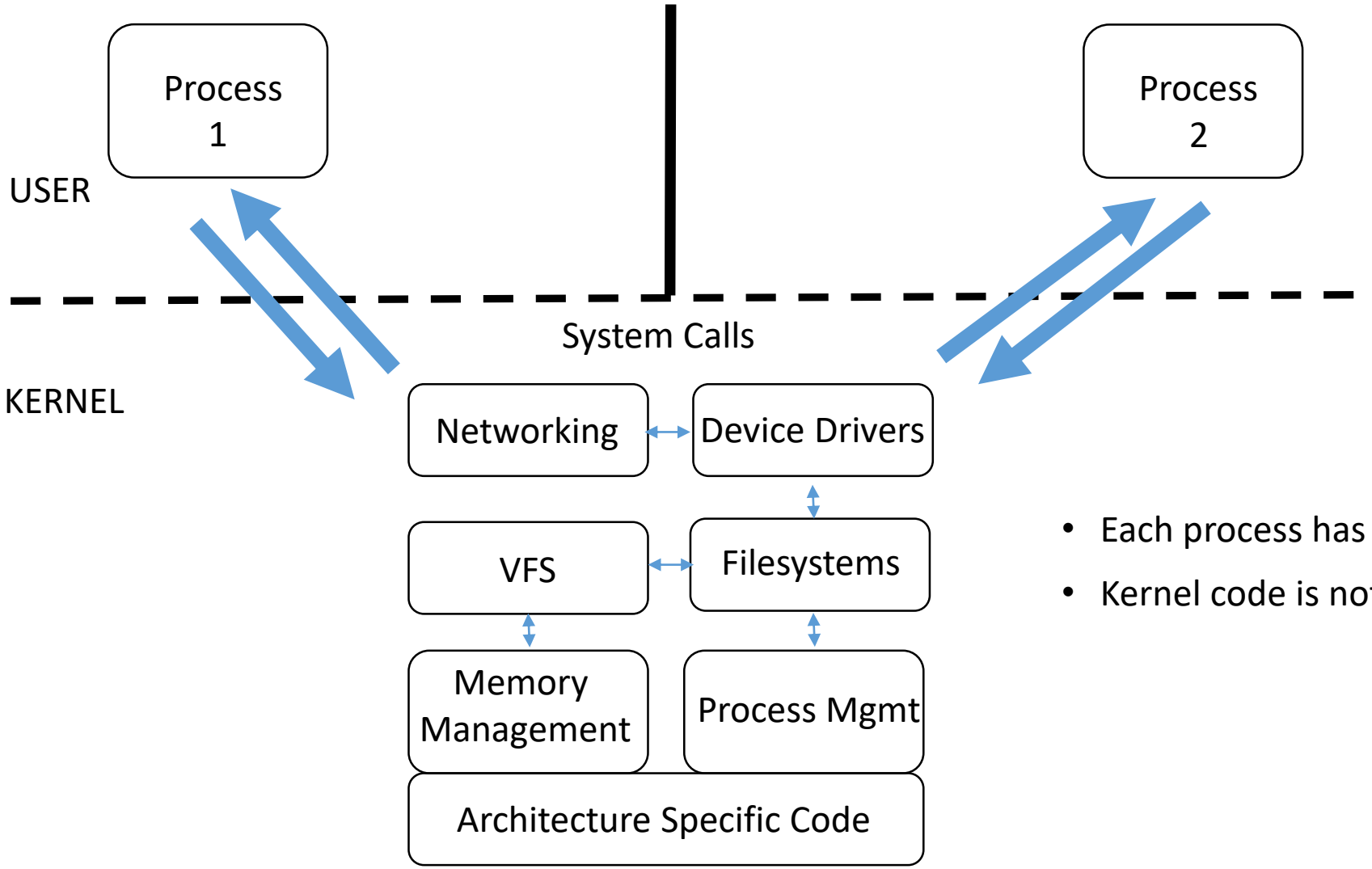
Demo: Attack Scenario



Part 4: Splitting the Kernel

(Hacking a Micro-Kernel interface into Linux)

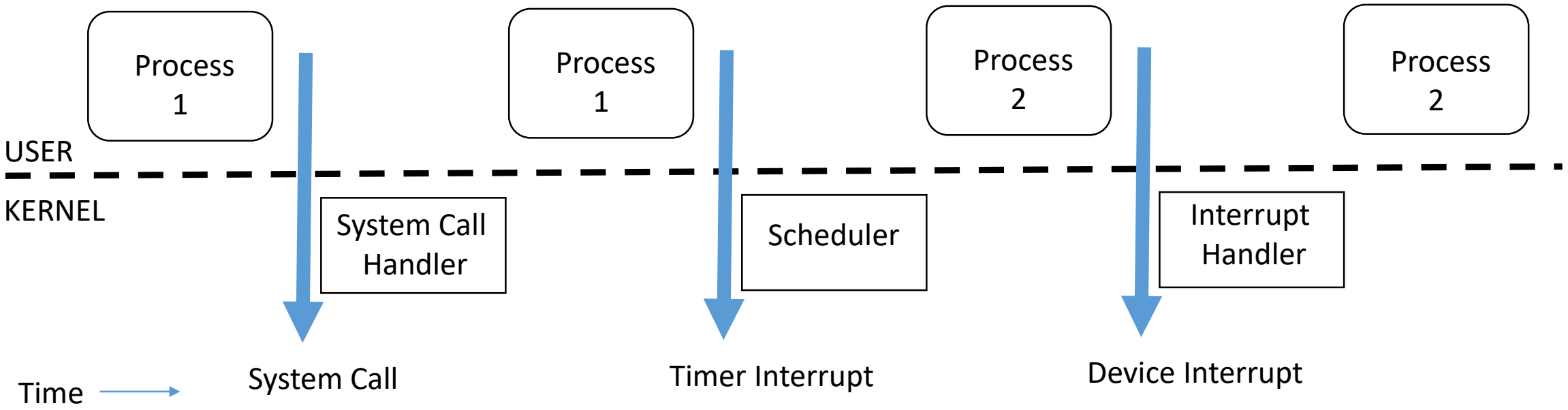
Linux Kernel



- Each process has its own isolated user space
- Kernel code is not isolated from itself

Diagram adapted from Mauerer.

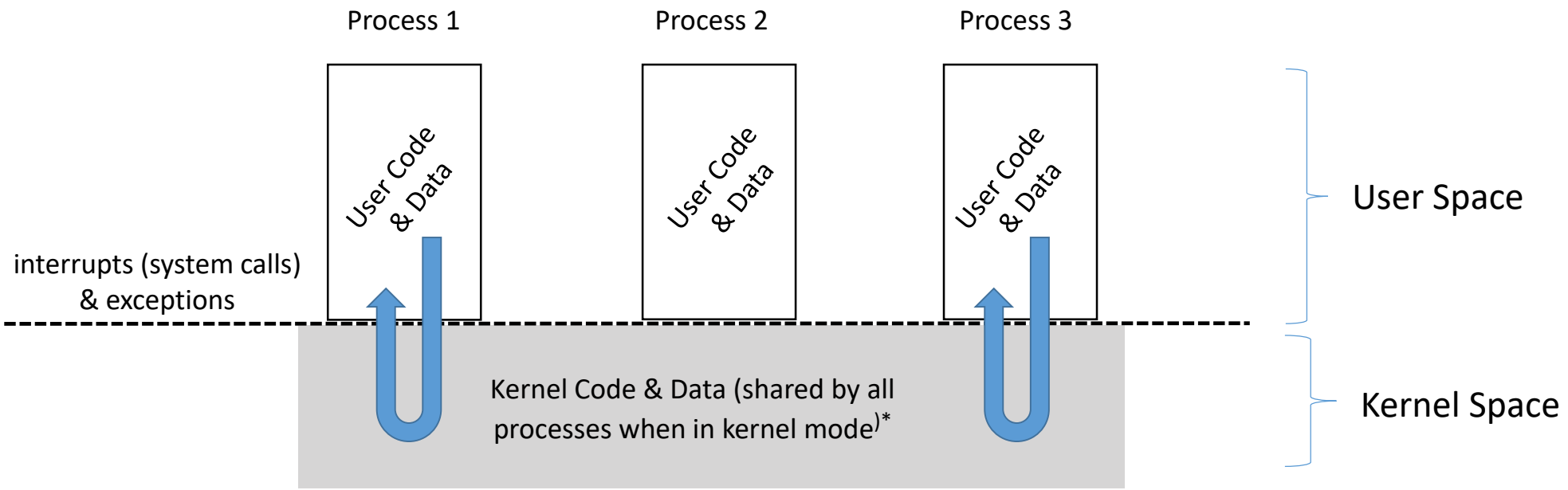
Some More Linux Kernel Background



- Each process has its own user space Page table mappings
  - Ignoring process cloning/threads
- Shares kernel mapping (synced from init_mm.pgd)
- Separate kernel stack per process
- Kernel entered through process context
  - It is not a separate 'thing' that runs concurrently
  - Does support the notion of kernel services via kthreads though

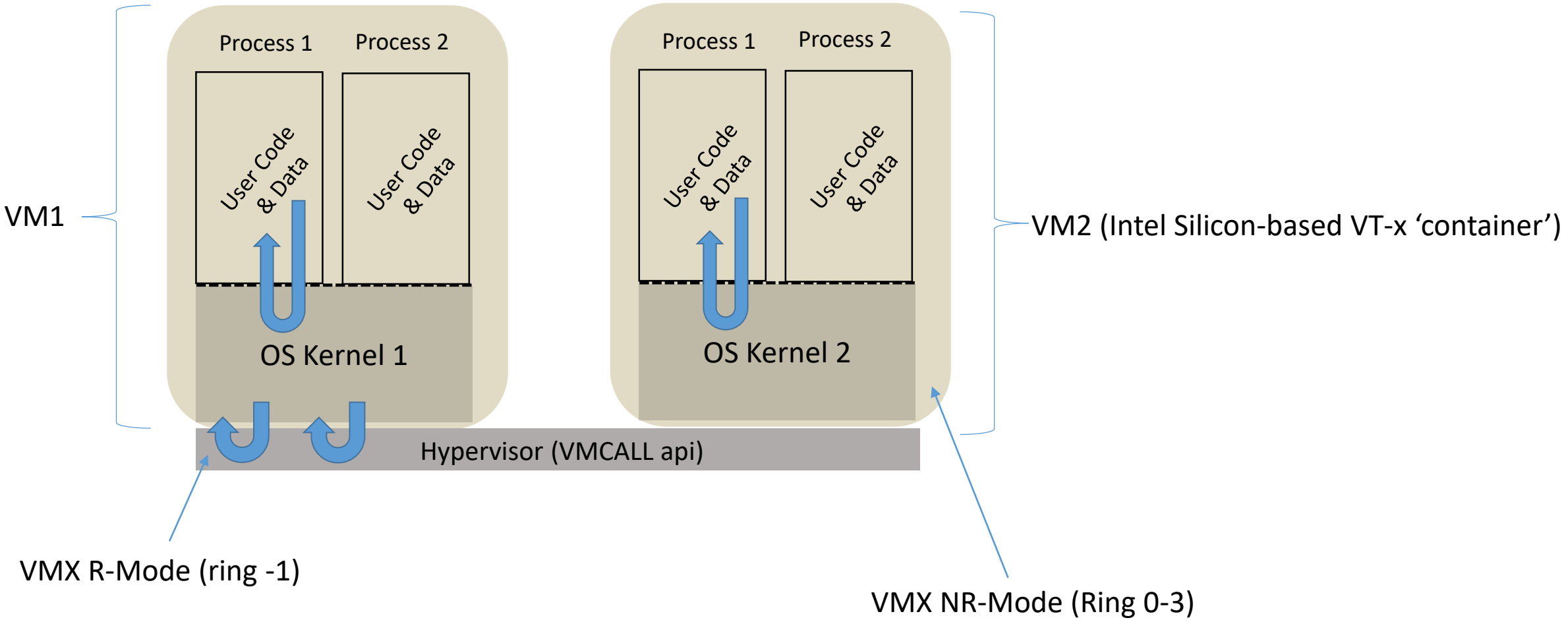
Diagram from Bovet,et al.

Linux Kernel Design



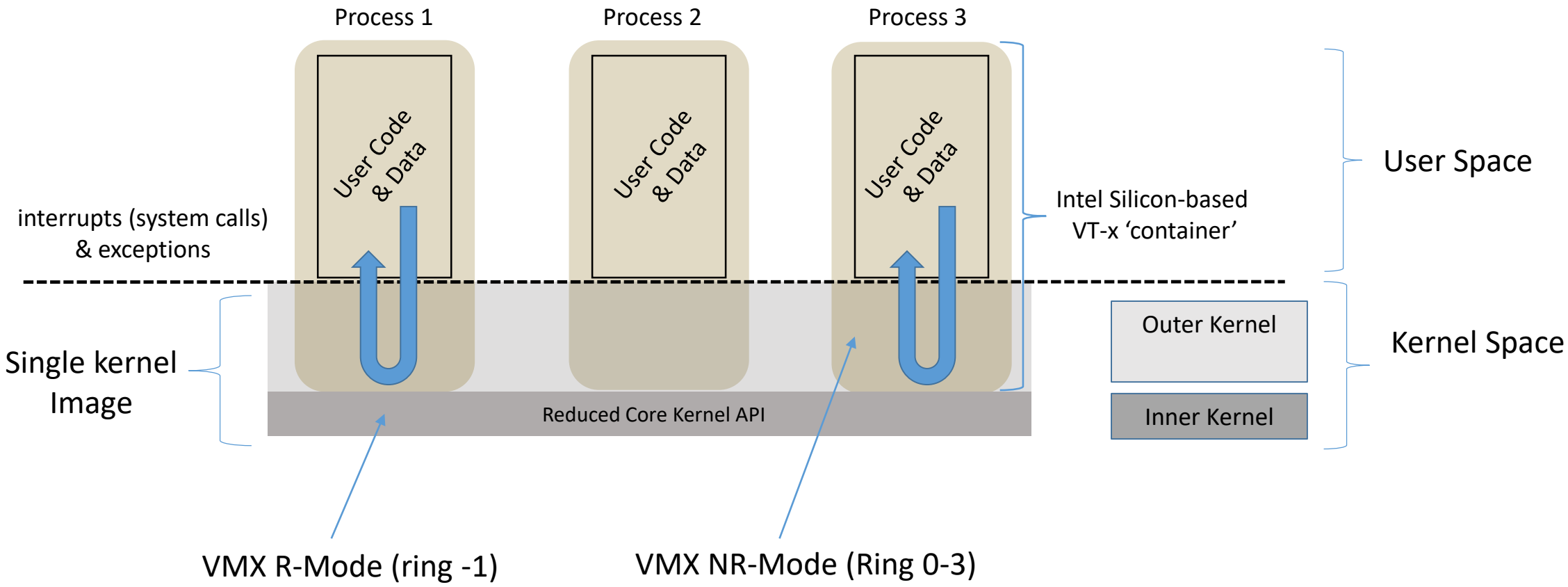
*separate kernel stack per process

Intel Virtualization HW support for VMMs

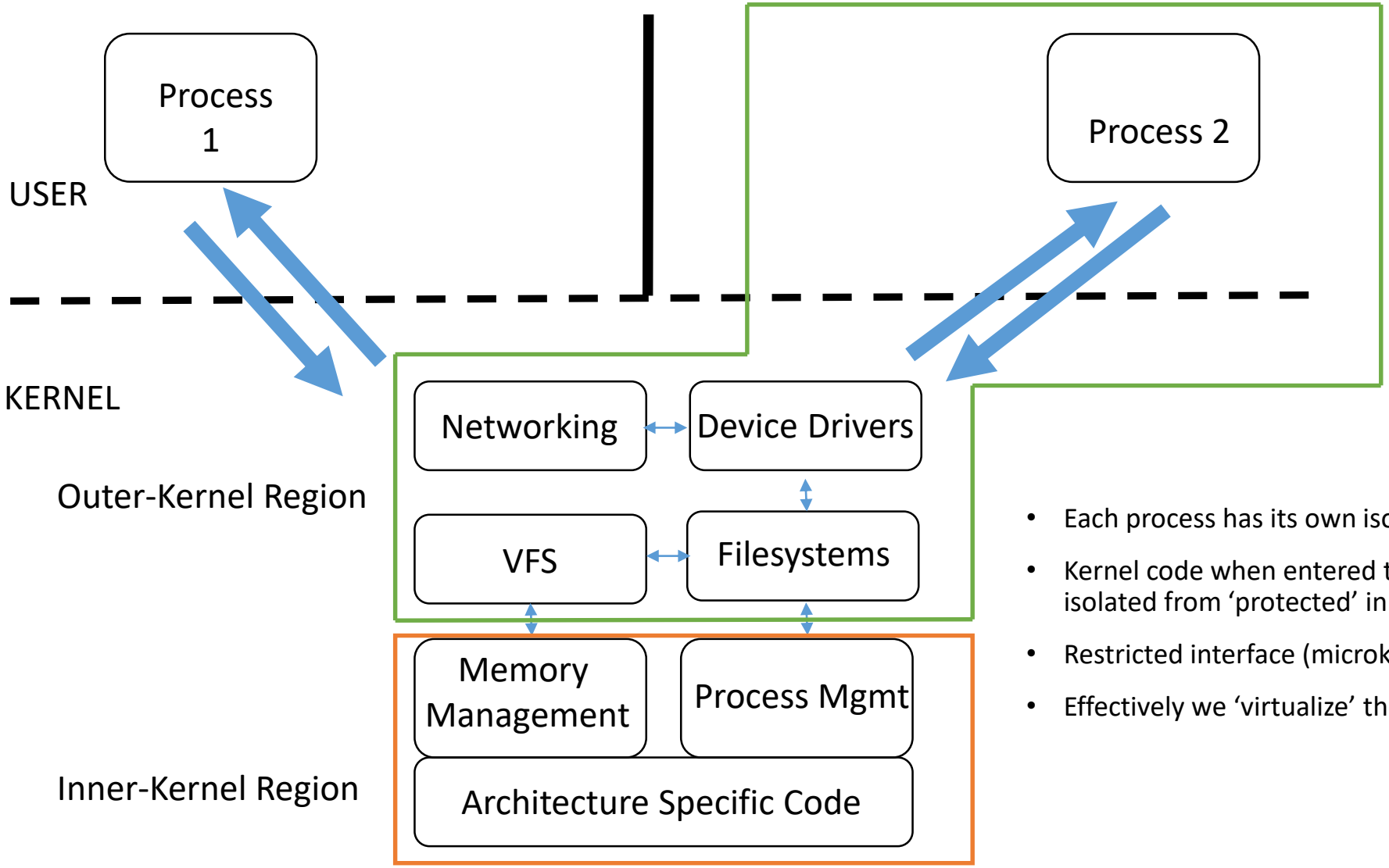


Separate Kernel Code & Data per VM

Linux Split-Kernel Design using Virtualization HW

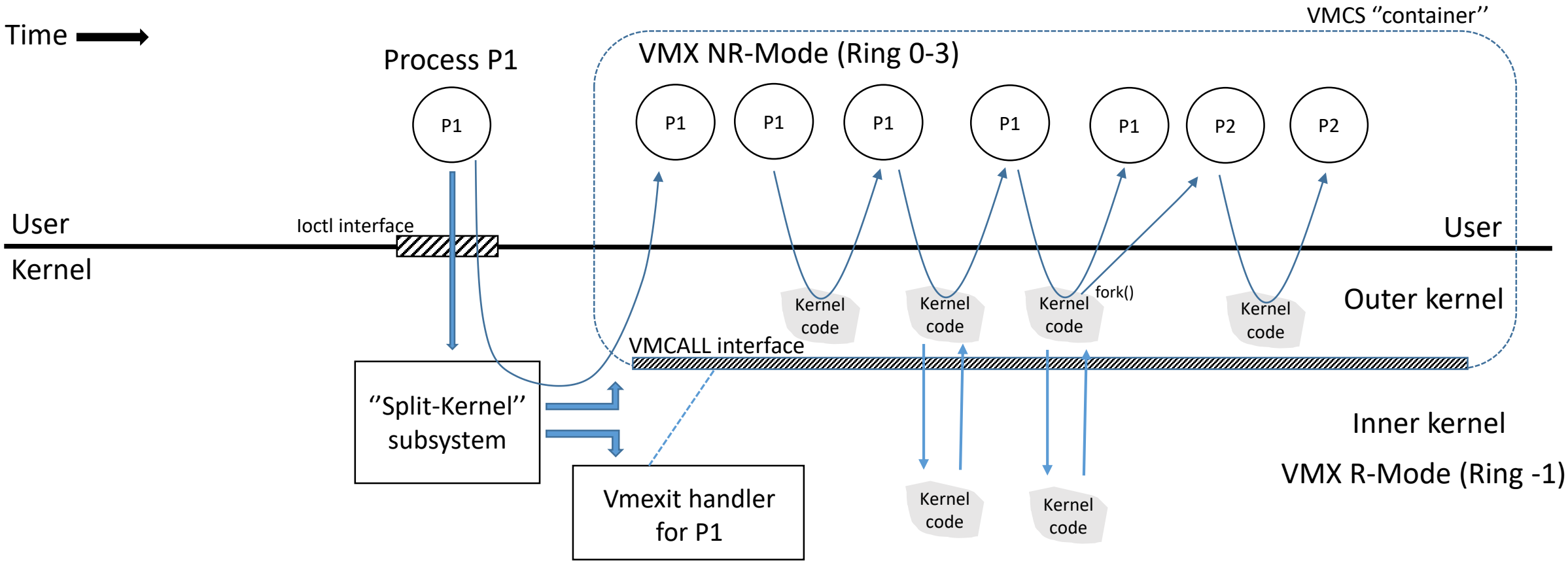


Split-Kernel (logical view)



- Each process has its own isolated user space
- Kernel code when entered through ‘containerized’ process is isolated from ‘protected’ inner kernel code & data
- Restricted interface (microkernel-like) to inner-kernel
- Effectively we ‘virtualize’ the kernel

Split-Kernel Process Lifecycle



Split-Kernel Architecture (x86_64)

- Use Intel VMX/EPT interposed on the normal linux kernel path
  - Allows vMMU interface for kernel entered through userspace-process to be enforced
  - Split the single shared kernel into an **Outer** region and an **Inner** region
  - Same kernel image but detects during execution whether in outer-kernel or inner-kernel mode
- X86: Each (container) process runs in vmx non-root mode within an EPT 'Container'
  - State defined by its own VMCS record
  - Each VMCS has its EPT pointer set to the same top-level table
  - Direct map 1:1 'host' physical memory except for 'protected' or 'private' memory regions
  - 'Protected' memory regions mapped RO, 'private' regions not mapped unless owned by that process
- do_schedule() from outer-kernel does VMCALL into inner-kernel
  - R-mode side of the process is scheduled / de-scheduled
- Inner-kernel needs to provide a VMEXIT handler
  - Need to maintain VMCS state across time-slices

Split-Kernel Architecture (x86_64)

- Use Intel VMX/EPT interposed on the normal linux kernel path
  - Allows vMMU interface for kernel entered through userspace-process to be enforced
  - Split the single shared kernel into an **Outer** region and an **Inner** region
  - Same kernel image but detects during execution whether in outer-kernel or inner-kernel mode
- X86: Each (container) process runs in vmx non-root mode within an EPT ‘Container’
  - State defined by its own VMCS record
  - Each VMCS has its EPT pointer set to the same top-level table
  - Direct map 1:1 ‘host’ physical memory except for ‘protected’ or ‘private’ memory regions
  - ‘Protected’ memory regions mapped RO, ‘private’ regions not mapped unless owned by that process
- do_schedule() from outer-kernel does VMCALL into inner-kernel
  - R-mode side of the process is scheduled / de-scheduled
- Inner-kernel needs to provide a VMEXIT handler
  - Need to maintain VMCS state across time-slices

Split-Kernel Architecture (x86_64)

- Use Intel VMX/EPT interposed on the normal linux kernel path
  - Allows vMMU interface for kernel entered through userspace-process to be enforced
  - Split the single shared kernel into an **Outer** region and an **Inner** region
  - Same kernel image but detects during execution whether in outer-kernel or inner-kernel mode
- X86: Each (container) process runs in vmx non-root mode within an EPT ‘Container’
  - State defined by its own VMCS record
  - Each VMCS has its EPT pointer set to the same top-level table
  - Direct map 1:1 ‘host’ physical memory except for ‘protected’ or ‘private’ memory regions
  - ‘Protected’ memory regions mapped RO, ‘private’ regions not mapped unless owned by that process
- do_schedule() from outer-kernel does VMCALL into inner-kernel
  - R-mode side of the process is scheduled / de-scheduled
- Inner-kernel needs to provide a VMEXIT handler
  - Need to maintain VMCS state across time-slices

Split-Kernel Architecture (x86_64)

- Use Intel VMX/EPT interposed on the normal linux kernel path
  - Allows vMMU interface for kernel entered through userspace-process to be enforced
  - Split the single shared kernel into an **Outer** region and an **Inner** region
  - Same kernel image but detects during execution whether in outer-kernel or inner-kernel mode
- X86: Each (container) process runs in vmx non-root mode within an EPT ‘Container’
  - State defined by its own VMCS record
  - Each VMCS has its EPT pointer set to the same top-level table
  - Direct map 1:1 ‘host’ physical memory except for ‘protected’ or ‘private’ memory regions
  - ‘Protected’ memory regions mapped RO, ‘private’ regions not mapped unless owned by that process
- do_schedule() from outer-kernel does VMCALL into inner-kernel
  - R-mode side of the process is scheduled / de-scheduled
- Inner-kernel needs to provide a VMEXIT handler
  - Need to maintain VMCS state across time-slices

What does this buy us again?

- Strong control over the integrity of kernel code and core data
  - No ‘un-authorized’ code gets to run in kernel mode
  - Can protect the integrity of data even against malicious kernel mode code
- Can offer confidentiality guarantees
  - Can protect application secrets even against malicious kernel mode code
- Guard against cross-process code + data compromise
  - Enhanced risk when running multiple ‘isolated’ containers

What does this buy us again?

- Strong control over the integrity of kernel code and core data
  - No ‘un-authorized’ code gets to run in kernel mode
  - Can protect the integrity of data even against malicious kernel mode code
- Can offer confidentiality guarantees
  - Can protect application secrets even against malicious kernel mode code
- Guard against cross-process code + data compromise
  - Enhanced risk when running multiple ‘isolated’ containers

What does this buy us again?

- Strong control over the integrity of kernel code and core data
  - No ‘un-authorized’ code gets to run in kernel mode
  - Can protect the integrity of data even against malicious kernel mode code
- Can offer confidentiality guarantees
  - Can protect application secrets even against malicious kernel mode code
- Guard against cross-process code + data compromise
  - Enhanced risk when running multiple ‘isolated’ containers

Some Code: Entering Outer-Kernel Mode

```
long ok_device_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
        switch(cmd)
        {
        case OKERNEL_ON_CMD:
                /* do okernel_enter() here... */
                HDEBUG("About to go into okernel on mode via okernel_enter() for pid (%d)\

                current->okernel_status = OKERNEL_ON;

                if(is_in_vmx_nr_mode()){
                        printk(KERN_CRIT "NR: Calling okernel_enter in  NR mode kernel...s
                }

                ret = okernel_enter(OKERNEL_ENTER_IOCTL);

                if(is_in_vmx_nr_mode()){
                        BXMAGICBREAK;
                        HDEBUG("Returning from okernel_enter (IOCTL_LAUNCH).\n");
                        BXMAGICBREAK;
                        goto nr_exit;
                }
                current->okernel_status = OKERNEL_OFF;

                if(ret){
                        printk(KERN_ERR "okernel_enter failed for pid <%d> ret (%d)\n", cu
                }
                HDEBUG("outer kernel off for <%d>\n", current->pid);
                unset_vmx_r_mode();
                break;
        default:
                printk(KERN_ERR "okernel invalid IOCTL cmd.\n");
                return -ENODEV;
}
nr_exit:
        return 0;
}
```

```
        HDEBUG("About to enter vmx handler while loop...\n");

        while(1){
                HDEBUG("Entering vmxit handler loop...\n");

                /* Use this instead of local_irq_disable() to save getting tangled in irq on/off
                native_irq_disable();

                /************************** GO FOR IT **************************/
                ret = vmx_run_vcpu(vcpu);
                /************************** GONE FOR IT! ***********************/

                if((vmcs_readl(GUEST_RFLAGS) & RFLAGS_IF_BIT)){
                        native_irq_enable();

                }

                HDEBUG("Returned from vmx_run_vcpu...handling exit condition...\n");

                if(ret==VMX_EXIT_REASONS_FAILED_VMENTRY){
                        HDEBUG("vmentry failed (%#x)\n", ret);
                        break;
                } else if (ret == EXIT_REASON_EPT_VIOLATION){
                        qual = vmcs_readl(EXIT_QUALIFICATION);
                        gp_addr = vmcs_readl(GUEST_PHYSICAL_ADDRESS);

                        HDEBUG("ept violation exit - qualification=%#lx gpa=%#lx\n",
                                qual, gp_addr);
                        /* Grant access to protected pages lazily */
                        if(__ok_protected_phys_addr(gp_addr)){
                                HDEBUG("ok: EPT vmexit on protected address:=(%#lx)\n", gp_addr);
                                if(ok_allow_protected_access(gp_addr)){
                                        (void)add_ept_page_perms(vcpu, gp_addr);

                                } else {
                                        /*
                                           Map in 'dummy' page for now  - need to be careful not
                                           create another vulnerability.
                                        */
                                        (void)remap_ept_page(vcpu, gp_addr, ok_get_protected_dumm
                                }
                                continue;
                        }
                HDEBUG("Done handling exit condition, looping...\n");
        }

        /* something went wrong - legitimate exit would have been through the
```

Some Code: Scheduling

```
asmlinkage __visible void __sched schedule(void)
{
        struct task_struct *tsk = current;
#if defined(CONFIG_OKERNEL)
#ifdef HPE_DEBUG
        volatile struct thread_info *ti;
        int cpu = smp_processor_id();
        struct tss_struct *tss = &per_cpu(cpu_tss, cpu);
        unsigned long fs;
        int orig_cpu = 0;
        int new_cpu = 0;
#endif

        if(is_in_vmx_nr_mode()){
                /* Return control to the original process running in root-mode VMX */
                /* shouldn't be holding locks at this point? */
#ifdef HPE_DEBUG
                ti = current_thread_info();
                rdmsrl(MSR_FS_BASE, fs);
                HDEBUG("called (pid=%d)  cpu_cur_tos (%#lx) flgs(%#x) MSR_FS_BASE=%#lx\n",
                        current->pid, (unsigned long)tss->x86_tss.sp0, ti->flags, fs);
                BXMAGICBREAK;
                HDEBUG("in_atomic(): %d, irqs_disabled(): %d, pid: %d, name: %s\n",
                       in_atomic(), irqs_disabled(), current->pid, current->comm);
                HDEBUG("preempt_count (%#x) rcu_preempt_depth (%#x)\n",
                       preempt_count(), rcu_preempt_depth());
#if defined(CONFIG_TRACE_IRQFLAGS) && defined(CONFIG_PROVE_LOCKING)
                HDEBUG("current->h_irqs_en (%d)\n",
                       current->hardirqs_enabled);
#endif /* CONFIG_TRACE_IRQFLAGS */
#endif /* HPE_DEBUG */
                sched_submit_work(tsk);
                (void)vmcall(VMCALL_SCHED);
        } else {
                sched_submit_work(tsk);
                do {
                        preempt_disable();
                        okernel_schedule(false);
                        sched_preempt_enable_no_resched();
                } while (need_resched());
        }
```

```
void vmx_handle_vmcall(struct vmx_vcpu *vcpu, int nr_irqs_enabled)
{
        struct vmx_vcpu *cpu_ptr;
        /* do_fork_fixup args */
        struct task_struct *p;

        cmd = vcpu->regs[VCPU_REGS_RAX];

        if(cmd == VMCALL_DO_EXEC_FIXUP_HOST){
                /* Next time we take a vmexit we will return using these page tables - should val
                h_cr3 = vcpu->regs[VCPU_REGS_RBX];
                HDEBUG("excec_fixup: Setting saved HOST CR3 to (%#lx) __pa (%#lx)\n",
                       (unsigned long)h_cr3, __pa(h_cr3));
                vmx_get_cpu(vcpu);
                vmcs_writel(HOST_CR3, __pa(h_cr3));
                vmx_put_cpu(vcpu);
                BXMAGICBREAK;
                ret = 0;
        } else if (cmd == VMCALL_SCHED){
                unset_vmx_r_mode();
                /* This is the only place we should be swapping CPUs */
                /* There is redunancy here: don't need to do all this flushing */
                vpid_sync_vcpu_global();
                ept_sync_global();
                barrier();

                schedule_r_mode();

                vpid_sync_vcpu_global();
                ept_sync_global();

                /* We may come back here on a different CPU...*/
                set_vmx_r_mode();

                cpu = smp_processor_id();
                tss = &per_cpu(cpu_tss, cpu);

                HDEBUG("ret schedule_r (pid %d) cpu_cur_tos (%#lx) tss.sp0 (%#lx) flgs (%#x)\n",
                       current->pid, current_top_of_stack(), (unsigned long)tss->x86_tss.sp0, r_t
                HDEBUG("ret from sched in_atomic(): %d, irqs_disabled(): %d, pid: %d, name: %s\n"
                       in_atomic(), irqs_disabled(), current->pid, current->comm);
                HDEBUG("ret from preempt_count (%d) rcu_preempt_depth (%d)\n",
                       preempt_count(), rcu_preempt_depth());
                BXMAGICBREAK;
        } else if(cmd == VMCALL_DOEXIT){
                code = (long)vcpu->regs[VCPU_REGS_RBX];
                HDEBUG("calling do_exit(%ld)...\n", code);
                vmx_destroy_vcpu(vcpu);
```

Performance and Invasiveness

- Still surprised it works at all…
  - Largest test machine 2 physical CPUS with 24 cores & 256 GB memory
  - Focus on functionality not performance or upstream optimization
  - Docker a good testcase for use of obscure Linux kernel features BTW
  - Initial debugging really,really hard
  - Can’t use printk(), etc but Bochs is really useful ☺
- Benchmarking
  - Linux kernel build, Apache Phoronix, etc.
  - Overhead around 2-5% depending upon number of processor cores
  - Depends on approach to handling process migration across CPU cores
- Invasiveness
  - Core code around 1000-2000 new lines in separate (static) kernel module
  - Plus maybe 100-200 lines of other Linux kernel modifications
  - Does still look like Linux!

Performance and Invasiveness

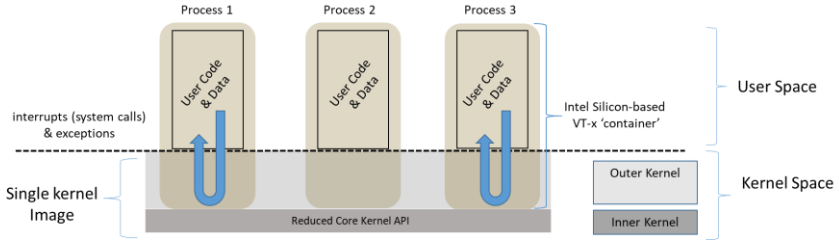
- Still surprised it works at all…
  - Largest test machine 2 physical CPUS with 24 cores & 256 GB memory
  - Focus on functionality not performance or upstream optimization
  - Docker a good testcase for use of obscure Linux kernel features BTW
  - Initial debugging really,really hard
  - Can’t use printk(), etc but Bochs is really useful ☺
- Benchmarking
  - Linux kernel build, Apache Phoronix, etc.
  - Overhead around 2-5% depending upon number of processor cores
  - Depends on approach to handling process migration across CPU cores
- Invasiveness
  - Core code around 1000-2000 new lines in separate (static) kernel module
  - Plus maybe 100-200 lines of other Linux kernel modifications
  - Does still look like Linux!

Performance and Invasiveness

- Still surprised it works at all…
  - Largest test machine 2 physical CPUS with 24 cores & 256 GB memory
  - Focus on functionality not performance or upstream optimization
  - Docker a good testcase for use of obscure Linux kernel features BTW
  - Initial debugging really,really hard
  - Can’t use printk(), etc but Bochs is really useful ☺
- Benchmarking
  - Linux kernel build, Apache Phoronix, etc.
  - Overhead around 2-5% depending upon number of processor cores
  - Processor scaling depends on approach to handling process migration across CPU cores
- Invasiveness
  - Core code around 1000-2000 new lines in separate (static) kernel module
  - Plus maybe 100-200 lines of other Linux kernel modifications
  - Does still look like Linux!

Current Status / Opportunities / Futures

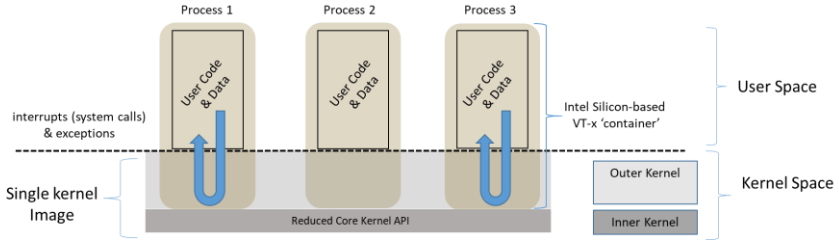
- Available at https://github.com/linux-okernel



- We do mention some of the concepts in our HOTOS paper
  - 'Separating Translation from Protection in Address Spaces with Dynamic Remapping', HOTOS 17
- Worth considering for general Linux use
  - Not just container deployments
  - Can run light-dm in outer-kernel mode (e.g. full desktop)
  - Need to fill-out ARM v8.1 implementation
- Good vehicle for kernel malware tracing on top of enhanced security
  - What else?

Current Status / Opportunities / Futures

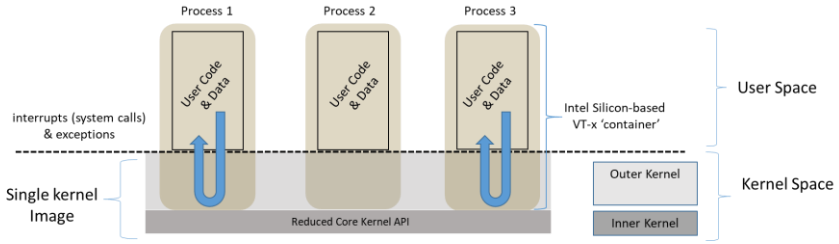
- Available at https://github.com/linux-okernel



- We do mention some of the concepts in our HOTOS paper
  - 'Separating Translation from Protection in Address Spaces with Dynamic Remapping', HOTOS 17
- Worth considering for general Linux use
  - Not just container deployments
  - Can run light-dm in outer-kernel mode (e.g. full desktop)
  - Need to fill-out ARM v8.1 implementation
- Good vehicle for kernel malware tracing on top of enhanced security
  - What else?

Current Status / Opportunities / Futures

- Available at https://github.com/linux-okernel



- We do mention some of the concepts in our HOTOS paper
  - 'Separating Translation from Protection in Address Spaces with Dynamic Remapping', HOTOS 17
- Worth considering for general Linux use
  - Not just container deployments
  - Can run light-dm in outer-kernel mode (e.g. full desktop)
  - Need to fill-out ARM v8.1 implementation
- Good vehicle for kernel malware tracing on top of enhanced security
  - What else?