User Mode and Interrupt Handling

Advanced Operating Systems

Overview

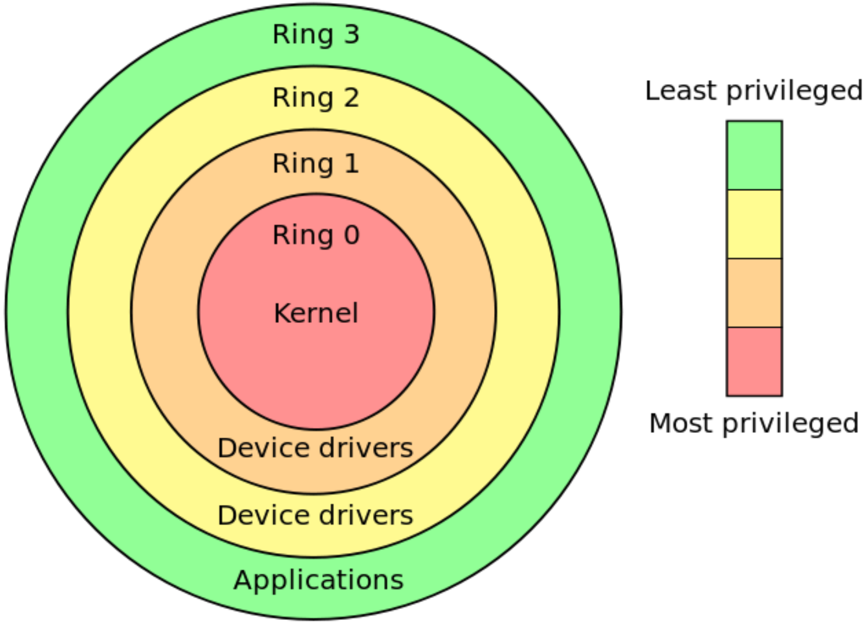
- User mode support
- User/kernel mode security
- Interrupts
- System calls

Operating System’s Task

Safe and efficient multiplexing of system resources to competing applications

- CPU provides support for management and safety through privileged instructions
- OS uses this support to manage user applications

Privilege Separation on x86



Typical Ring Usage

- Ring 0: kernel
- Ring 1: unused
- Ring 2: unused
- Ring 3: user processes

Differences Between Rings

- Ring 0 can use any instruction
- Rings 1-3 cannot
  - access supervisor-only pages
  - read/write most system registers
  - use memory management instructions
- Rings allowed to do I/O configurable

Segmentation

- Memory is divided in several **segments**
- Each pointer dereference has a segment associated with it
  - Code pointer: code segment (CS)
  - Heap pointer: data segment (DS)
  - Local variable pointer: stack segment (SS)
- CS:0x1000 can be different than DS:0x1000

NOW IRRELEVANT

or is it?

Segmentation in x86

- x86: segments introduced 42 years ago
  - Pointers were 16 bits
  - 64K memory was no longer enough
- x86_64: pretends not to have segments
  - All segments point to same memory*
  - However, low-order 2 bits of CS register determine current ring

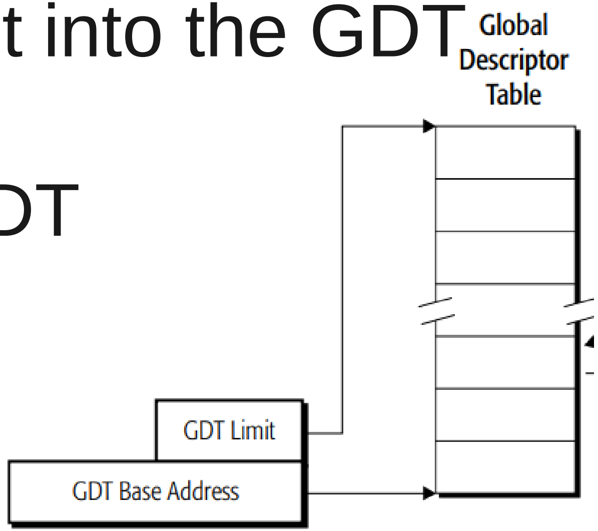
*more or less

Switching x86 Rings

- There is no instruction to set CS directly
- Traditional approach
  - From user mode to the kernel
    - Interrupt CPU (“int” instruction)
    - CPU switches to interrupt handler in kernel
  - From kernel to user mode
    - Kernel interrupt handler returns (“iret” instruction)
    - CPU restores user mode state, including CS

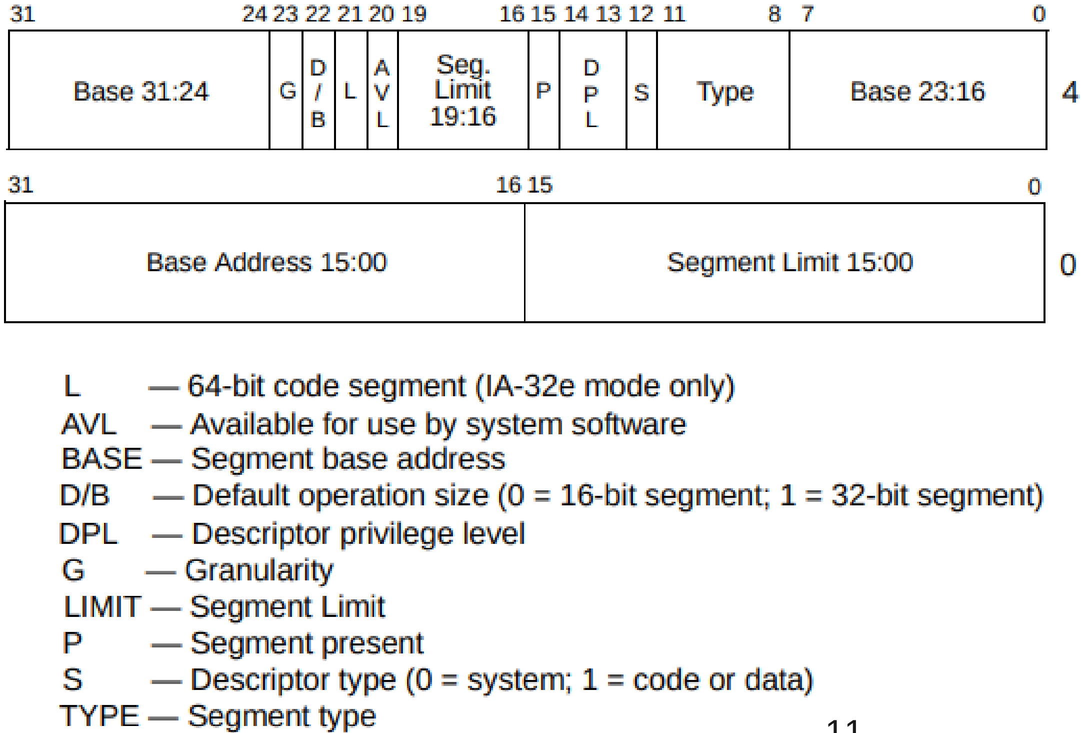
Global Descriptor Table

- Defines segments (among other things)
- Your code segment is an offset into the GDT
- So is your data segment
- GDTR register points to the GDT
- boot/boot2.S for inspiration



Global Descriptor Table Register

64-bit GDT Entries



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

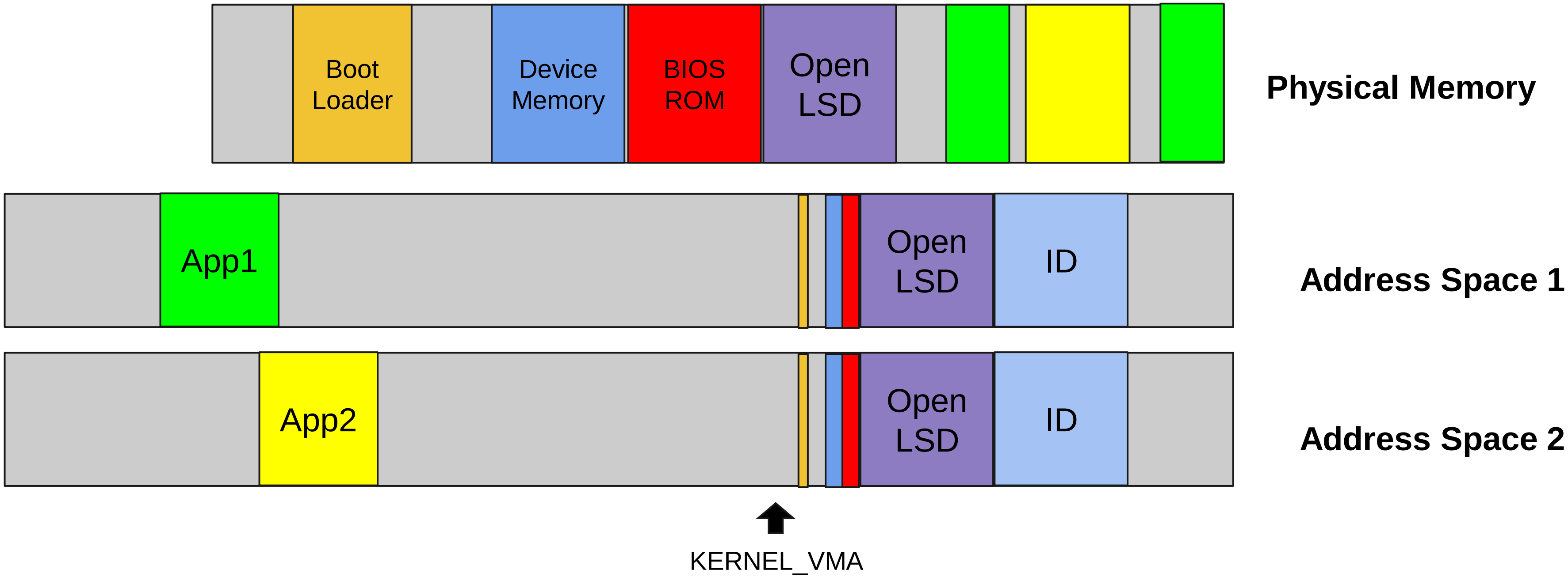
Lots of legacy

- Segment limits: all set to 1
- Base address: all set to 0
- DPL decides which rings can access

Address Spaces and User Processes

- Different page tables create different address spaces
- Address spaces isolate processes from each other
- Most modern OSes provide a one-to-one mapping between address spaces and user processes
    - E.g., Windows, MacOS, and Linux
    - Possible to provide protection with managed languages instead (e.g., Singularity)

OpenLSD
Basic Virtual Address Space



Process Management

- Multiple live processes in the system
- Each of these processes may hold a number of resources (e.g., memory, files, etc.)
- Some processes may cooperate to build applications while distrusting other processes
- Some processes have higher privileges
- More about process management in Lab 5

Example: OpenLSD task (inc/task.h)

```
struct task {
  /* The saved registers. */
  struct int_frame task_frame;

  /* The task this task is waiting on. */
  struct task *task_wait;

  /* The process ID of this task and its parent. */
  pid_t task_pid;
  pid_t task_ppid;

  /* ... */

  /* The virtual address space. */
  struct page_table *task_pml4;
};
```

Example: Linux task_struct (selection of include/linux/sched.h)

```
struct task_struct {
  /* -1 unrunnable, 0 runnable, >0 stopped: */
  volatile long state;

  void *stack;

  /* Current CPU: */
  unsigned int cpu;

  const struct sched_class *sched_class;
  struct mm_struct *mm;
  pid_t pid;

  /* Real parent process: */
  struct task_struct __rcu *real_parent;

  /* Objective and real subjective task credentials (COW): */
  const struct cred __rcu *real_cred;

  /* Open file information: */
  struct files_struct *files;
};
```

User Mode in OpenLSD

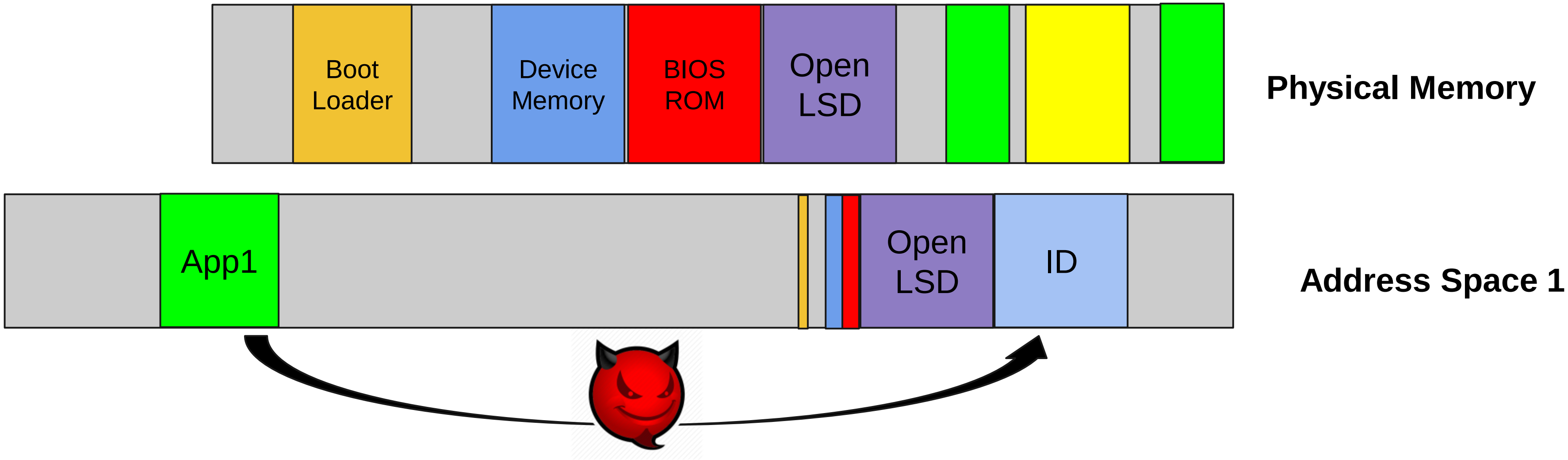
Lab 3: Three rings under
(user mode support and interrupts)

Will run a user space app at the end

Kernel Security

Bad guys are leaking your kernel secrets!

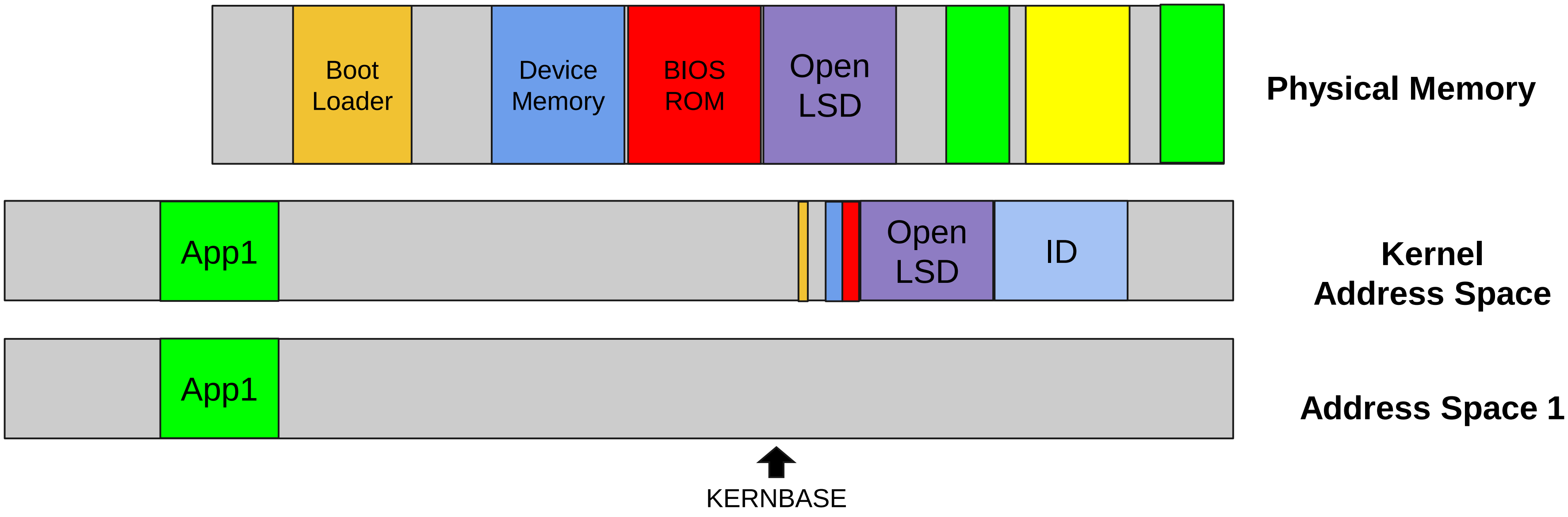
Intel CPU Vulnerability



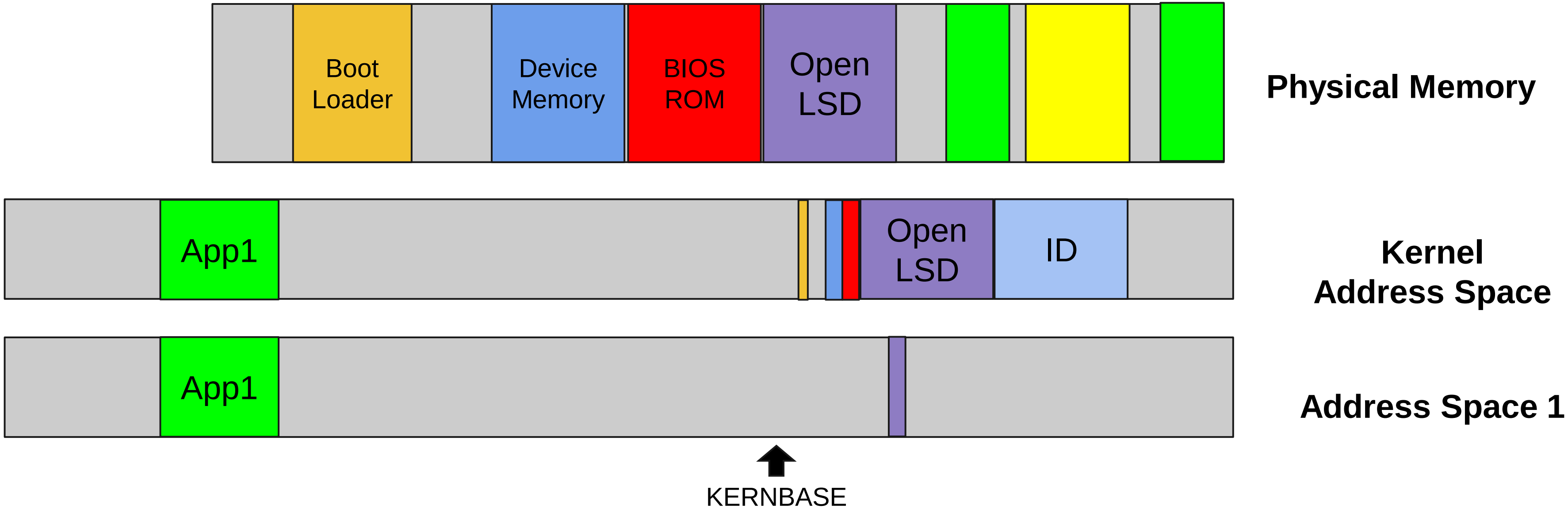
Rogue Data Cache Load - branded as Meltdown

Bypasses the supervisor bit in PTEs → Arbitrary read

OpenLSD Bonus:
Kernel Page Table Isolation (KPTI)

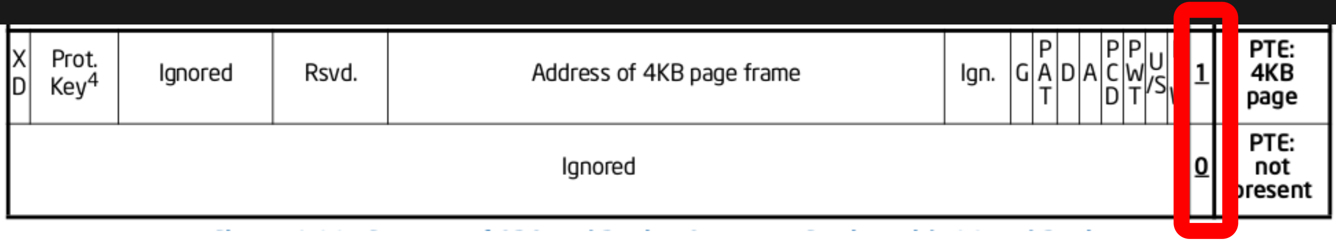


OpenLSD Bonus: KPTI



What is the minimum you need to map?

(Partially) Mitigating Foreshadow

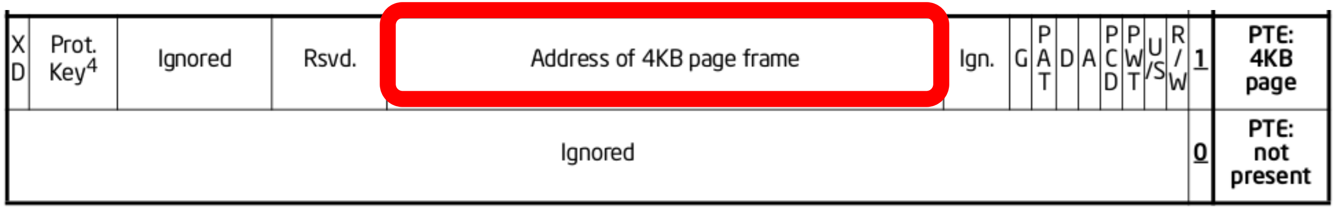


XD	Prot. Key^4	Ignored	Rsvd.	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S		1	PTE: 4KB page
Ignored														0	PTE: not present

L1TF (dubbed Foreshadow) bypasses the present bit

PTE address bits must be cleared.

(Partially) Mitigating RIDL



XD	Prot. Key[4]	Ignored	Rsvd.	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	**PTE: 4KB page**
Ignored														0	**PTE: not present**

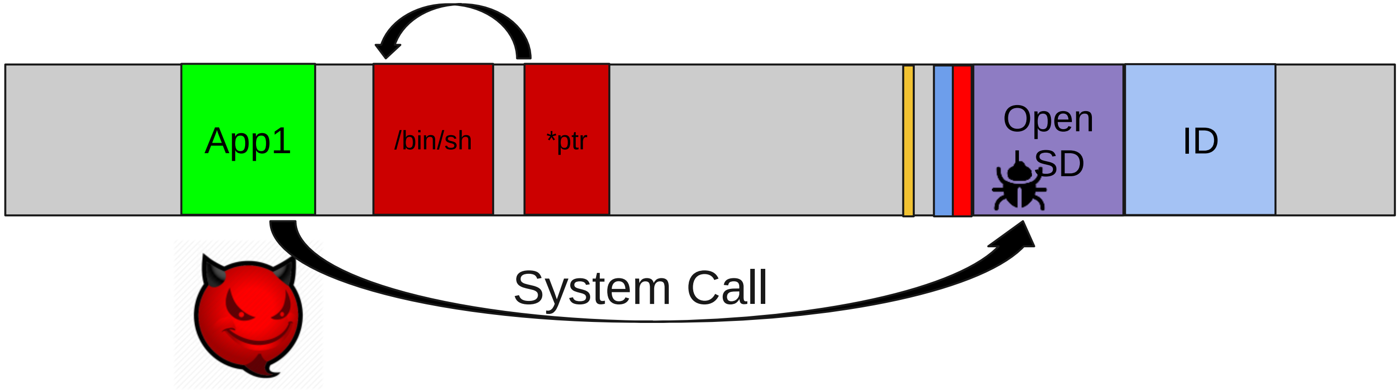
MDS (dubbed RIDL) bypasses the address bits

Flush CPU buffers before iretq with verw

Kernel Security

Bad guys are exploiting your kernels!

Defending Kernel Attacks



SMEP: Supervisor Mode Execution Protection (just enable)

SMAP: Supervisor Mode Access Protection (copy_to/from_user)

User Mode Security

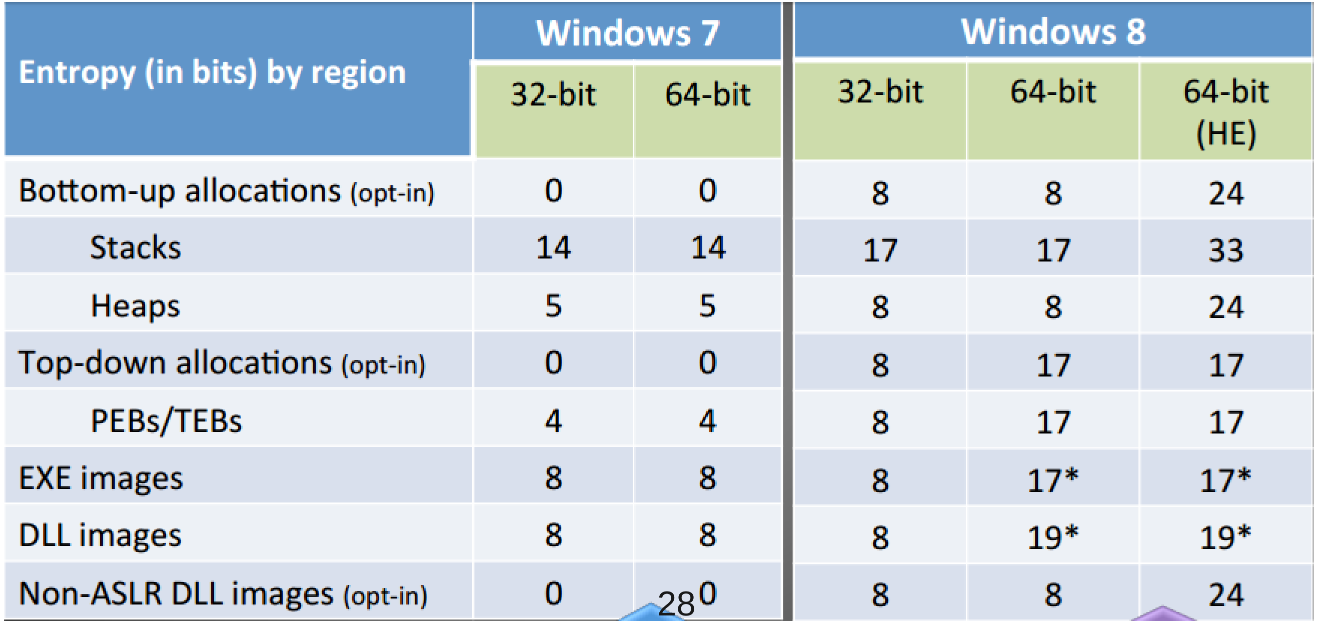
Bad guys are exploiting your user applications!

Address Space Layout Randomization (ASLR) to the Rescue

- Attackers use software (?) vulnerabilities to divert control flow or perform data attacks
  - Famous hacks: Trinity’s SSH CRC-32 exploit
  - More real: RPC DCOM interface overflow (Blaster)
- ASLR: map user mode code and data in random locations in the virtual address space
  - Makes it significantly harder to write exploits
  - But see: Dedup Est Machina and AnC

ASLR Entropy in Popular OSes

Linux mmap provides 28 bits of entropy



Entropy (in bits) by region	Windows 7		Windows 8		
	32-bit	64-bit	32-bit	64-bit	64-bit (HE)
Bottom-up allocations (opt-in)	0	0	8	8	24
Stacks	14	14	17	17	33
Heaps	5	5	8	8	24
Top-down allocations (opt-in)	0	0	8	17	17
PEBs/TEBs	4	4	8	17	17
EXE images	8	8	8	17*	17*
DLL images	8	8	8	19*	19*
Non-ASLR DLL images (opt-in)	0	0	8	8	24

Lab 3 Bonuses

Lots of goodies:

Hardening: user mode ASLR, SMEP/SMAP

Mitigations: KPTI, Foreshadow, RIDL

Interrupt Handling

What Are Interrupts

- Event that “interrupts” execution flow
  - Kernel handles interrupt before program execution can continue
- Can be external
  - key presses, network packets
- Can be internal
  - divide by zero, page faults

Terminology

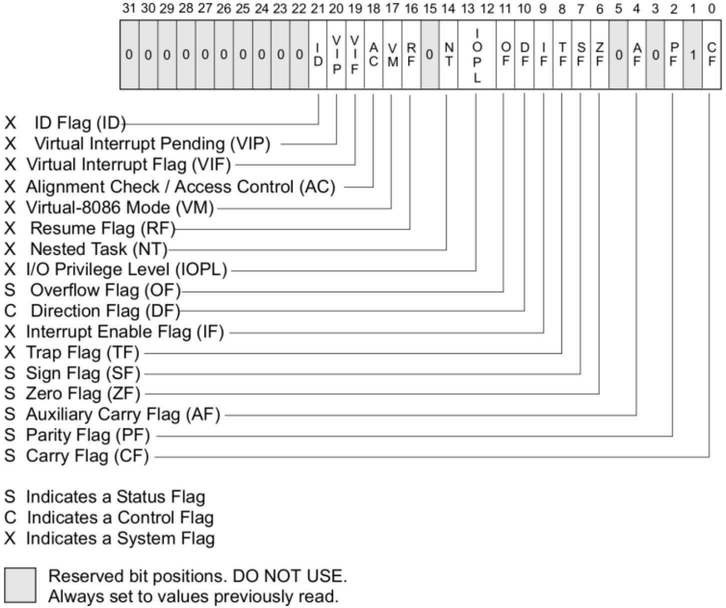
- Interrupt
  - External interrupt
    - Maskable hardware interrupt
    - Non-maskable interrupt
  - Software-generated interrupt
- Exception
  - Fault
  - Trap
  - Abort

Note: despite the terminology, the interrupt handling system applies to both interrupts and exceptions

Note 2: these terms are officially defined by Intel, but often used loosely in practice

External Interrupts

- Hardware-generated (example: Lab 5)
  - Device wants the kernel to respond to something
  - Example: key press, network packet, …
  - Device signals CPU by setting a pin
- Most hardware interrupts can be masked
  - Interupt handling postponed if IF in EFLAGS is not set


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check / Access Control (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Reserved bit positions. DO NOT USE. Always set to values previously read.

Software Interrupts

- Software-generated (example: Lab 3)
  - Software needs some service from the kernel
  - Examples: opening a file, allocating memory, …
- Generated by executing “int *n*” instruction

Exceptions

- Fault
  - Error condition (divide by zero, page fault, …)
  - Instruction can be retried after addressing error
- Trap
  - Typically for debugging (breakpoint, overflow, …)
  - Program can resume at next instruction
- Abort
  - Serious error condition (fault handling interrupt, unrecoverable hardware error)
  - Program cannot continue

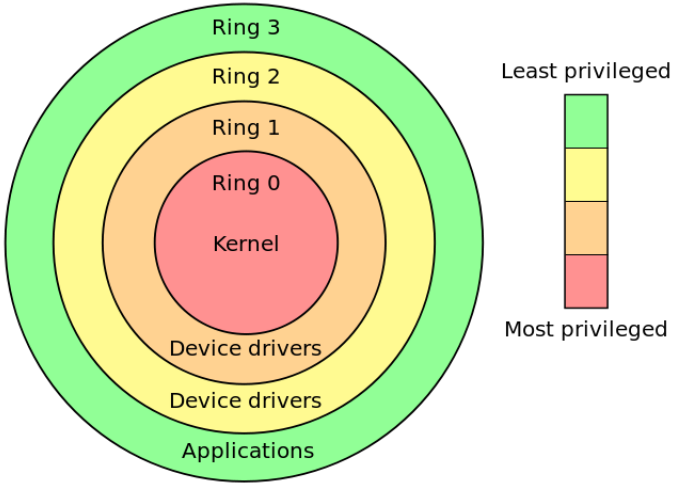
Note: the terms trap and fault are often used to refer to *all* exceptions

(A)synchronous interrupts

- Most software interrupts are synchronous
    - Directly before (fault) or after (trap) specific instruction
    - Kernel can do handling specific to the instruction (example: map missing page)
- Most hardware interrupts are asynchronous
    - Can come at any time, especially the most inconvenient ones
    - Proper masking is important,
      for example when handling interrupt

Interrupts and Privilege Levels

- Interrupts often result in a change in the state of the system
- Interrupt handlers typically run at
  a high privilege level



What Happens During an Interrupt

Safe transfer of execution to kernel mode

1. CPU elevates privilege level and switches to the kernel stack
2. Some user context (e.g., rip) is saved
3. A function is called to handle the interrupt
   - AKA Interrupt service routine
4. CPU restores some of the user context and drops the privilege level

Programming Interrupts

- Interrupts can come from different sources
  - More advanced information on interrupts → Lab 6
- CPU needs to know what to do when interrupted
- Kernel programs CPU to call interrupt service routine
  - On x86 this is done using interrupt descriptor table (IDT)

Some Notes on IDT

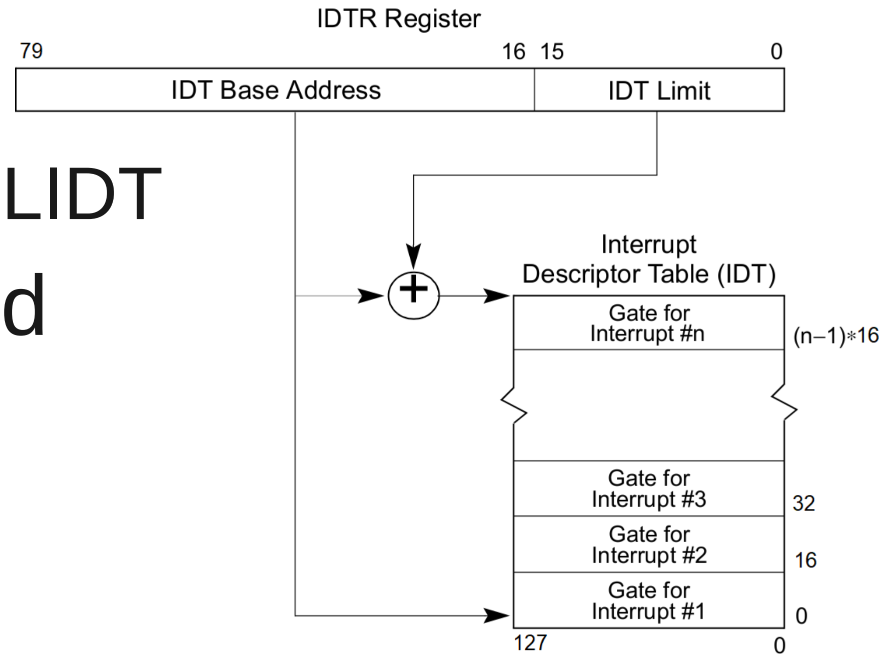
- Max 256 entries
- First 32 entries reserved for exceptions (e.g. page fault 14, double fault 8)
- 16 external (hardware) interrupts can be (re)mapped using the APIC unit (lab 5)

Calling Interrupt Handler: Overview

- Interrupt vector identifies type of interrupt
  - $0 \leq$ interrupt vector < 256
- Look up interrupt gate in IDT using vector
- Jump to interrupt handler and ring
- Mask further interrupts
- Switch stack
- Store calling context

Calling Interrupt Handler: Look Up Interrupt Gate

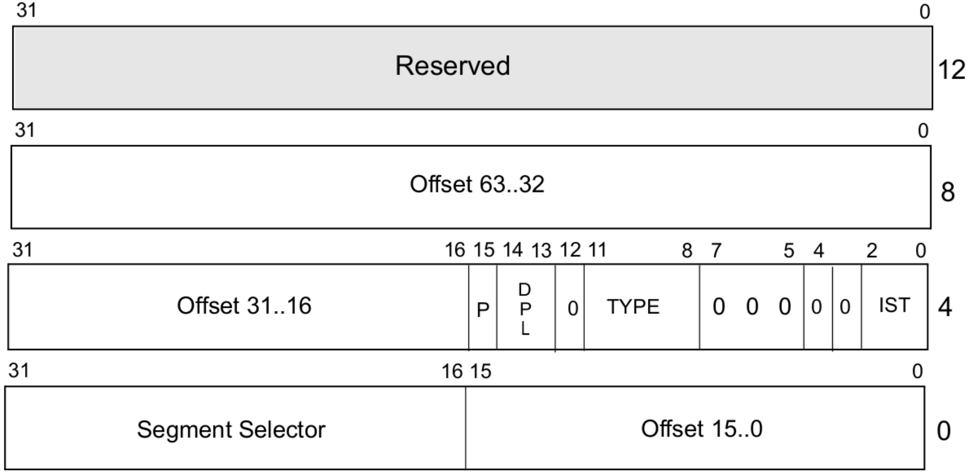
- IDTR points to IDT
  - Previously set with LIDT
- Interrupt vector used as index into IDT
  - Vector depends on cause of interrupt



Calling Interrupt Handler: Jump to Interrupt Handler

- Set CS to Segment Selector
  - This changes ring
- Set RIP to Offset
  - This jumps to interrupt handler

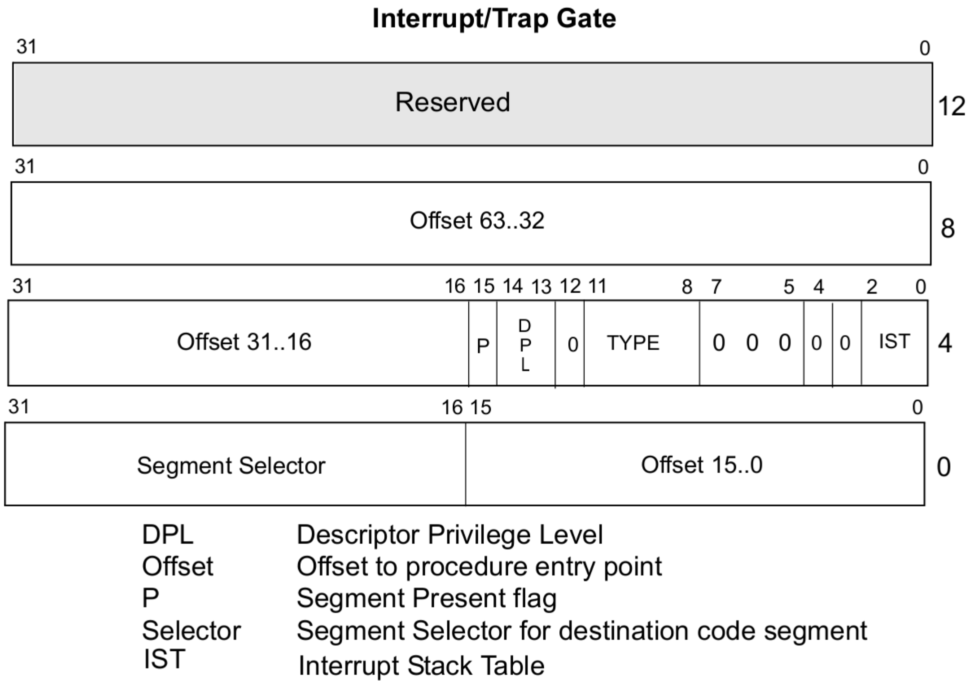
Interrupt/Trap Gate



DPL	Descriptor Privilege Level
Offset	Offset to procedure entry point
P	Segment Present flag
Selector	Segment Selector for destination code segment
IST	Interrupt Stack Table

Calling Interrupt Handler: Mask Further Interrupts

- Type can be interrupt gate or trap gate
- Interrupt gate clears IF in EFLAGS to mask further interrupts



Interrupt/Trap Gate

DPL	Descriptor Privilege Level
Offset	Offset to procedure entry point
P	Segment Present flag
Selector	Segment Selector for destination code segment
IST	Interrupt Stack Table

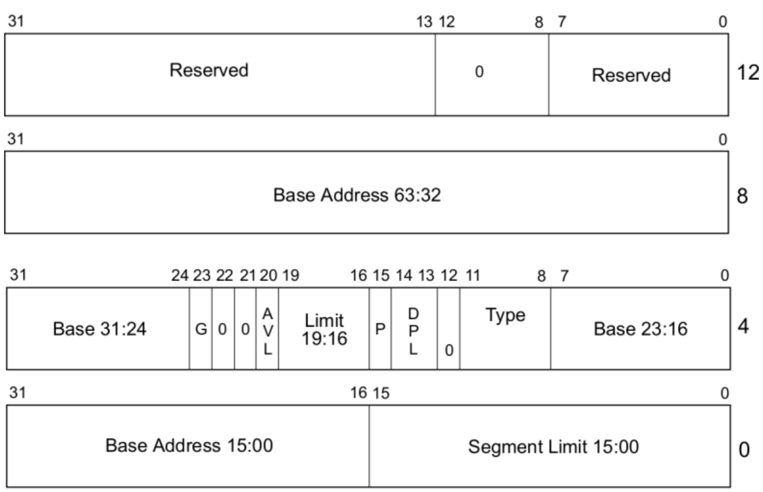
Calling Interrupt Handler: Switch Stack

- Kernel stack pointer stored in legacy data structure: Task State Segment (TSS)
- Task Register (TR) contains the index in the GDT that specifies where TSS is
    - Specify with LTR instruction
- TSS contains stack pointers for each ring
- Stack segment set to null

Calling Interrupt Handler: Switch Stack – Finding TSS

- TSS descriptor uses two 64-bit entries in GDT, indexed by TR
- Glue pieces of Base Address together to find location of TSS

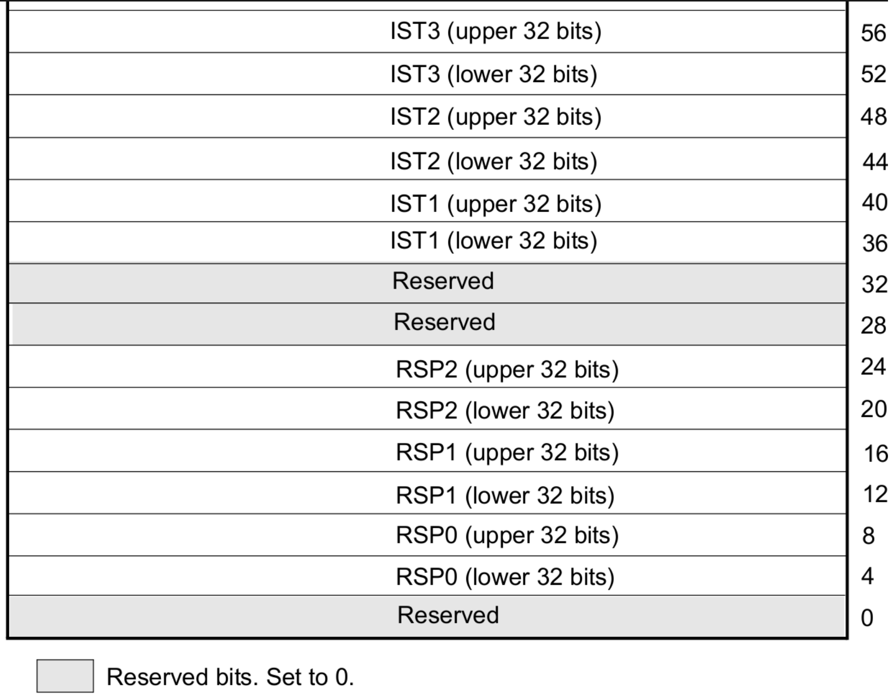
TSS (or LDT) Descriptor



AVL	Available for use by system software
B	Busy flag
BASE	Segment Base Address
DPL	Descriptor Privilege Level
G	Granularity
LIMIT	Segment Limit
P	Segment Present
TYPE	Segment Type

Calling Interrupt Handler: Switch Stack – TSS

- Load RSP0 from the TSS into RSP
- Set SS to null



IST3 (upper 32 bits)	56
IST3 (lower 32 bits)	52
IST2 (upper 32 bits)	48
IST2 (lower 32 bits)	44
IST1 (upper 32 bits)	40
IST1 (lower 32 bits)	36
Reserved	32
Reserved	28
RSP2 (upper 32 bits)	24
RSP2 (lower 32 bits)	20
RSP1 (upper 32 bits)	16
RSP1 (lower 32 bits)	12
RSP0 (upper 32 bits)	8
RSP0 (lower 32 bits)	4
Reserved	0

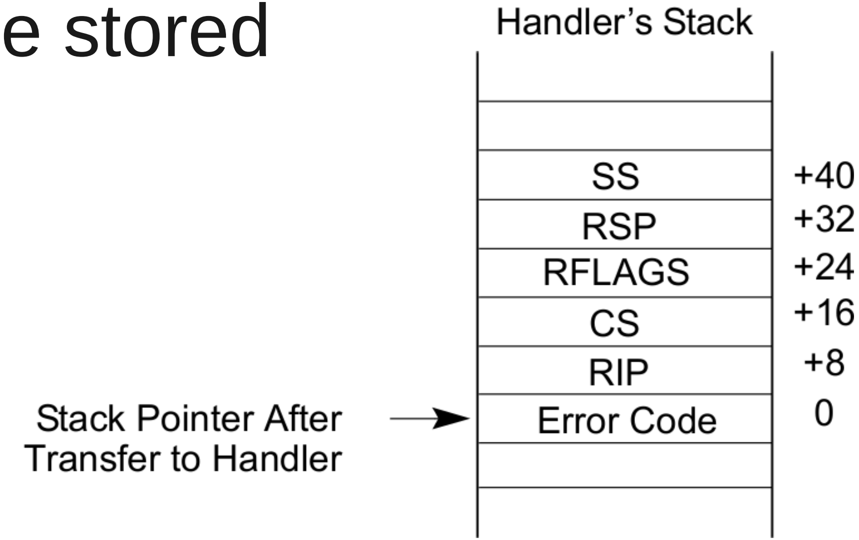
▨ Reserved bits. Set to 0.

TSS in OpenLSD

- Interrupt stack = top of the kernel stack
- OpenLSD loads correct TSS for this lab
  - You need to take care of it for multicore support

Calling Interrupt Handler: Store Calling Context

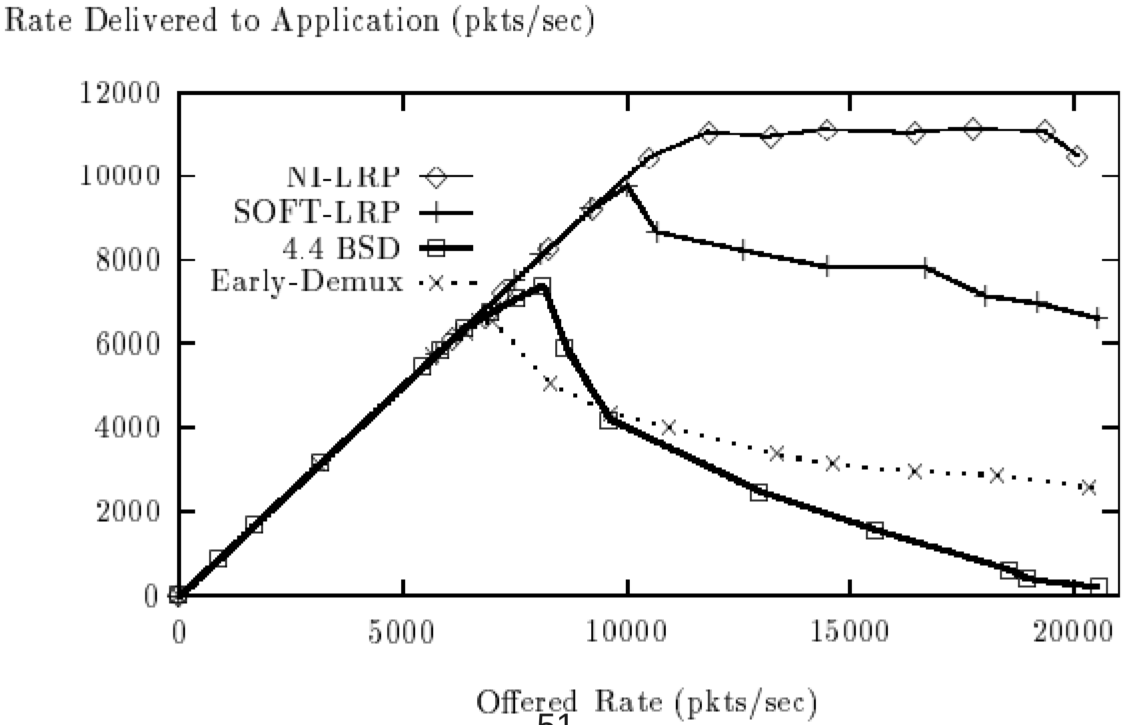
- Old register values are stored on the kernel stack to be restored later



Returning from Interrupts

- When handler is finished, IRET returns back to the location at the time of interrupt
- IRET does quite a bit
  - Pops the right amount from the stack
  - Restore the stack
  - Returns to last RIP, dropping privilege

Problem with Interrupts



Dealing with Livelocks

- Do as little as you can in the interrupt handler
  - Schedule work for later
  - Demux events early and drop if needed
- Reduce number of interrupts
  - Make use of hardware demux-ing
  - Polling instead of large numbers of interrupts
  - Transfer data directly to user applications (RDMA)

Interrupts in Linux: Setting Up the IDT (arch/x86/kernel/idt.c)

```
static const __initconst struct idt_data def_idts[] = {
  INTG(X86_TRAP_DE,       asm_exc_divide_error),
  INTG(X86_TRAP_NMI,      asm_exc_nmi),
  INTG(X86_TRAP_BR,       asm_exc_bounds),
  INTG(X86_TRAP_UD,       asm_exc_invalid_op),
  INTG(X86_TRAP_NM,       asm_exc_device_not_available),
  INTG(X86_TRAP_OLD_MF,   asm_exc_coproc_segment_overrun),
  INTG(X86_TRAP_TS,       asm_exc_invalid_tss),
  INTG(X86_TRAP_NP,       asm_exc_segment_not_present),
  INTG(X86_TRAP_SS,       asm_exc_stack_segment),
  INTG(X86_TRAP_GP,       asm_exc_general_protection),
  INTG(X86_TRAP_SPURIOUS, asm_exc_spurious_interrupt_bug),
  INTG(X86_TRAP_MF,       asm_exc_coprocessor_error),
  INTG(X86_TRAP_AC,       asm_exc_alignment_check),
  INTG(X86_TRAP_XF,       asm_exc_simd_coprocessor_error),
  /* ... */
};

void __init idt_setup_traps(void)
{
  idt_setup_from_table(idt_table, def_idts, ARRAY_SIZE(def_idts), true);
}
```

```
static __init void set_intr_gate(
  unsigned int n, const void *addr)
{
  struct idt_data data;

  BUG_ON(n > 0xFF);

  memset(&data, 0, sizeof(data));
  data.vector  = n;
  data.addr  = addr;
  data.segment  = __KERNEL_CS;
  data.bits.type  = GATE_INTERRUPT;
  data.bits.p  = 1;

  idt_setup_from_table(idt_table, &data, 1, false);
}
```

Interrupts in Linux: Entering the Kernel

```
/* arch/x86/include/asm/idtentry.h */
DECLARE_IDTENTRY(X86_TRAP_DE,       exc_divide_error);

#define DECLARE_IDTENTRY(vector, func) \
  idtentry vector asm_##func func has_error_code=0

/* x86/entry/entry_64.S */
.macro idtentry vector asmsym cfunc has_error_code:req
SYM_CODE_START(\asmsym)
  ASM_CLAC
  .if \has_error_code == 0
    pushq  $-1      /* ORIG_RAX: no syscall to restart */
  .endif
  idtentry_body \cfunc \has_error_code
SYM_CODE_END(\asmsym)
.endm

.macro idtentry_body cfunc has_error_code:req
  call  error_entry
  UNWIND_HINT_REGS
  movq  %rsp, %rdi      /* pt_regs pointer into 1st argument*/
  .if \has_error_code == 1
    movq  ORIG_RAX(%rsp), %rsi  /* get error code into 2nd argument*/
    movq  $-1, ORIG_RAX(%rsp)  /* no syscall to restart */
  .endif
  call  \cfunc
  jmp  error_return
.endm
```

Interrupts in Linux: Setting Up Kernel Environment

```
/* x86/entry/entry_64.S */
SYM_CODE_START_LOCAL(error_entry)
  UNWIND_HINT_FUNC
  cld
  PUSH_AND_CLEAR_REGS save_ret=1
  ENCODE_FRAME_POINTER 8
  testb  $3, CS+8(%rsp)
  jz  .Lerror_kernelspace

  SWAPGS
  FENCE_SWAPGS_USER_ENTRY
  /* We have user CR3.  Change to kernel CR3. */
  SWITCH_TO_KERNEL_CR3 scratch_reg=%rax

  /* Put us onto the real thread stack. */
  popq  %r12        /* save return addr in %12 */
  movq  %rsp, %rdi      /* arg0 = pt_regs pointer */
  call  sync_regs
  movq  %rax, %rsp      /* switch stack */
  ENCODE_FRAME_POINTER
  pushq  %r12
  ret

  FENCE_SWAPGS_KERNEL_ENTRY
  ret
```

Interrupts in Linux: the C Side

```
/* arch/x86/include/asm/idtentry.h */
#define DECLARE_IDTENTRY(vector, func)      \
  asmlinkage void asm_##func(void);         \
  asmlinkage void xen_asm_##func(void);     \
  __visible void func(struct pt_regs *regs)

DECLARE_IDTENTRY(X86_TRAP_DE, exc_divide_error);

#define DEFINE_IDTENTRY(func)                               \
static __always_inline void __##func(struct pt_regs *regs); \
                                                            \
__visible noinstr void func(struct pt_regs *regs) {         \
  bool rcu_exit = idtentry_enter_cond_rcu(regs);            \
  instrumentation_begin();                                  \
  __##func (regs);                                          \
  instrumentation_end();                                    \
  idtentry_exit_cond_rcu(regs, rcu_exit);                   \
}

/* arch/x86/kernel/traps.c */
DEFINE_IDTENTRY(exc_divide_error) {
  do_error_trap(regs, 0, "divide_error", X86_TRAP_DE, SIGFPE,
          FPE_INTDIV, error_get_trap_addr(regs));
}
```

Interrupts in Linux:
Returning To User Mode

```
/* arch/x86/entry/entry_64.S */
SYM_CODE_START_LOCAL(error_return)
  UNWIND_HINT_REGS
  DEBUG_ENTRY_ASSERT_IRQS_OFF
  testb  $3, CS(%rsp)
  jz  restore_regs_and_return_to_kernel
  jmp  swapgs_restore_regs_and_return_to_usermode
SYM_CODE_END(error_return)

SYM_INNER_LABEL(
  swapgs_restore_regs_and_return_to_usermode,
  SYM_L_GLOBAL)
  POP_REGS pop_rdi=0
  /*
   * The stack is now user RDI, orig_ax, RIP, CS,
   * EFLAGS, RSP, SS.
   * Save old stack pointer and switch to
   * trampoline stack.
   */
  movq  %rsp, %rdi
  movq  PER_CPU_VAR(cpu_tss_rw + TSS_sp0), %rsp
  UNWIND_HINT_EMPTY
```

```
/*Copy the IRET frame to the trampoline stack*/
pushq  6*8(%rdi)  /* SS */
pushq  5*8(%rdi)  /* RSP */
pushq  4*8(%rdi)  /* EFLAGS */
pushq  3*8(%rdi)  /* CS */
pushq  2*8(%rdi)  /* RIP */

/* Push user RDI on the trampoline stack. */
pushq  (%rdi)

/*
 * We are on the trampoline stack.
 * All regs except RDI are live.
 * We can do future final exit work right here.
 */
STACKLEAK_ERASE_NOCLOBBER

SWITCH_TO_USER_CR3_STACK scratch_reg=%rdi

/* Restore RDI. */
popq  %rdi
SWAPGS
INTERRUPT_RETURN
```

System Calls

System Calls

- Kernel support for servicing user applications
- Software generated through interrupt instruction
  - “int 0x80” on OpenLSD and Linux
- More modern variants
  - Sysenter/sysexit (x86), syscall/sysret (x86_64)

Syscalls - `int`

- Originally, only issued using `int` instruction
- Dispatch routine just an interrupt handler
- Syscalls often arranged in a table, e.g.:

```
asmlinkage const sys_call_ptr_t sys_call_table[
  __NR_syscall_max+1] = {
  [0 ... __NR_syscall_max] = &__x64_sys_ni_syscall,
#include <asm/syscalls_64.h>
};
```

- Kernel-user communication dictated by calling convention

Syscalls - Linux

- Each OS specifies its own calling convention
- x86 Linux calling convention:
  - `int 0x80` to issue a system call
  - Syscall number is specified in %rax
  - Arguments are specified in other registers:
    - %rdi, %rsi, %rdx, %r10, %r8 and %r9
  - Kernel places return value in %rax

Syscall Performance

- Modern processors are deeply pipelined
- Cache misses can cause pipeline stalls
- Pipeline stalls are very expensive
    - IDT entry may not be in the cache
- This became noticeable around the P4 era:
    - Syscalls took twice as long from P3 to P4
    - High impact for syscall-intensive workloads

Syscalls – `syscall`

- **Idea**: What if we cache the IDT entry for a system call in a special CPU register?
  - No more cache misses for the IDT!
- `syscall`/`sysret` instructions
  - Dedicated registers
  - Other optimizations are also possible
- **Assumption**: System calls are frequent enough to be worth the transistor budget

Syscalls – `syscall` MSR Setup

- Requires setup through MSR (model-specific registers) via `rdmsr` / `wrmsr` instructions
- `syscall`
    - IA32_EFER (0x80): E-Feature Register (SCE bit set)
    - IA32_LSTAR (0x82): Ring-0 `%rip`
    - IA32_FMASK (0x84): Mask with bits to clear in RFLAGS
    - IA32_STAR (0x81): Ring-0 CS/SS selectors (47:32)
- `sysret`
    - IA32_STAR (0x81): Ring-3 CS/SS selectors (63:48)

Syscalls – `syscall` Behavoir

- `syscall:`
  - Saves RFLAGS in %r11 (and masks RFLAGS)
  - Saves `%rip` in `%rcx` (and switches `%rip` to handler)
  - Switches CS, SS, and privilege level (ring-0)
- `sysret:`
  - Restores RFLAGS from %r11
  - Restores `%rip` from %rcx
  - Switches CS, SS, and privilege level (ring-3)
- `%rsp` saved/restored by software (ring-0/3)

Syscalls – syscall MSR Setup

```
/* arch/x86/kernel/cpu/common.c */

void syscall_init(void)
{
  wrmsr(MSR_STAR, 0, (__USER32_CS << 16) | __KERNEL_CS);
  wrmsrl(MSR_LSTAR, (unsigned long)entry_SYSCALL_64);

  /* ... */

  /* Flags to clear on syscall */
  wrmsrl(MSR_SYSCALL_MASK,
         X86_EFLAGS_TF|X86_EFLAGS_DF|X86_EFLAGS_IF|
         X86_EFLAGS_IOPL|X86_EFLAGS_AC|X86_EFLAGS_NT);
}
```

VDSO

- In 32-bit x86, fast system call instruction (`sysenter`) was optional in CPU
  - Need to retain legacy `int` support
- Solution
  - Kernel figures out what CPU supports using `cpuid` instruction
  - Virtual syscall page with optimal system call instruction

VDSO

- Virtual syscall page is now called `vdso` (Virtual Dynamic Shared Object)
- Originally mapped at fixed user address
- Replaces `int 0x80` with a `call <addr>`
- Allows for a fixed return point (needed for sysenter)
- Looks like a DSO (program transparent)
- Now includes “goodies” (`gettimeofday`)

Syscalls - sysenter: vdso

```
$ cat /proc/self/maps
5628c2b38000-5628c2b40000 r-xp 00000000 103:02 5505049    /bin/cat
5628c2d3f000-5628c2d40000 r--p 00007000 103:02 5505049    /bin/cat
5628c2d40000-5628c2d41000 rw-p 00008000 103:02 5505049    /bin/cat
5628c4b73000-5628c4b94000 rw-p 00000000 00:00 0           [heap]
... (libraries)
7eff82b7d000-7eff82b7e000 rw-p 00000000 00:00 0
7fff67771000-7fff67792000 rw-p 00000000 00:00 0           [stack]
7fff6779f000-7fff677a2000 r--p 00000000 00:00 0           [vvar]
7fff677a2000-7fff677a3000 r-xp 00000000 00:00 0           [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0   [vsyscall]
```

References

1. Intel® 64 and IA-32 Architectures Optimization Reference Manual
2. 80386 Programmer’s manual, https://sipb.mit.edu/iap/6.828/readings/i386/toc.htm
3. MIT caffeinated 6.828, https://sipb.mit.edu/iap/6.828/lab/lab3/
4. x86 Protected Mode Exceptions, https://support.microsoft.com/en-us/kb/117389
5. Peter Druschel and Gaurav Banga, Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *SOSP*, 1996
6. Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, Edouard Bugnion, IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *OSDI*, 2014
7. Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson and Miguel Castro, FaRM: Fast Remote Memory. In *NSDI*, 2014.
8. Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. R2C2: A Network Stack for Rack-scale Computers, In *SIGCOMM*, 2015.
9. Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P*, 2016.
10. Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. “ASLR on the Line: Practical Cache Attacks on the MMU.” In NDSS, 2017.