Secure Operating System Design and Implementation x86 architecture

Jon A. Solworth

Dept. of Computer Science
University of Illinois at Chicago

April 3, 2012

Part I

X86 architecture overview

X86 architecture overview

The x86 architecture components that most effect OS programming

- Privileged instructions
- Traps and interrupts
- Time
- Data layout (e.g., page tables)
- Memory semantics
- Virtualization

x86 processor modes

mode	characteristic
real	the original 8086 instruction set with 2^{20} memory addresses
protected	the mode for the 80286, supporting 2^{24} memory addresses
protected	for the 80386, supporting 2^{32} memory addresses and virtual memory
system management	handle system errors, provide power management facilities.
long	AMD64 bit architecture supports 64-bit virtual addresses and 64-bit word size.

Registers

- Control Registers: CR0, CR2, CR3 CR4, CR8 Control system function and some system features
- System-Flags Register: system status and masks
- Descriptor Table Registers: used for segmentation
- Task Register: task state segment

Current Privilege Level

- The Current Privilege Level (CPL) is contained in the low order two bits of the Current Code Segment (CS).
- Need to be at ring 0 to execute privileged instructions
- Need to be at ring 0 to access privilege resources, such as
  - CR0 enables paging, caching, alignment checks and other processor behaviors
  - CR2 contains the page fault linear address
  - CR3 points the current page table (if paging selected)
  - CR4 used in protected mode for PAE, Debugging extensions,

Part II

Memory Protection

Memory protection overview

- Processes need to be isolated from each other
- The kernel must be isolated from processes
- **segmentation** enable memory regions to be exactly specified
- **paging** enables memory to be split up and protected in fixed sized pages
- each of paging and segmentation provides memory protection for read, write, and execute

x86 memory architecture

- Relevant memory architectures
  - 32-bit protected: 32-bit virtual memory and word size
  - 32-bit PAE: 36-bit physical memory, with a 32-bit virtual memory PAE—Physical Address Extensions
  - x86-64: 64-bit virtual memory and word size (Also called AMD64 and Intel64)
- Current Intel (non-atom) processors and AMD processors support all these modes.
- x86 architectures support older modes, including **real mode**, which is the initial mode for all x86 architectures
- But Xen brings up the OS into protected mode or higher.
- Xen supports only 32-bit PAE or 64-bit

X86-64

- 64-bit integer registers and pointers
- Doubled the number of integer registers to 16
- Virtual address space is currently 2^{48} (eventually 2^{64})
- Physical address space is currently 2^{40} (eventually 2^{64})
- Addresses are sign extended, facilitating locating the kernel in high memory and user programs in low memory.
- Instruction pointer relative address accesses (efficient support for Position Independent Code) in libraries.
- NX (no execute bit) to prevent page from being used to execute from.

X86-64 (cont'd)

Long mode can be either

- long mode: 64-bit address; CS.L = 1
- compatibility mode: 32-bit address; CS.L = 0

Memory translation

- Modern OSs use either protected mode in 32-bit
- Or long mode in 64-bit (which is an extension to protected)
- Every memory access is first mapped through segmentation
- Segmentation adds an offset to the program generated address
- Segmentation cannot be turned off, but by setting the base address to 0 (and the limits address to the maximum memory address), segmentation performs the identity mapping.
- And then if paging is turned on, the resulting address is mapped through the page table

Address types

Logical Address consisting of segmentSelector : offset

Segment selector usually implicitly chosen, e.g.,

- CS: (code segment) for instructions
- SS: (stack segment) for stack access
- DS: (data segment) for non-stack data accesses

Linear Address (or virtual address) is

segmentBaseAddress + effectiveAddress

Physical Address is the result of page mapping the Linear Address

Computing an effective address

- Effective Address, or segment offset, is part of logical address
- The effective address is what is in a C pointer
- In assembly language it is computed as follows
  - **base** + **scale** * **index** + **displacement**
  - Where **base** is stored in a general purpose register;
  - **scale** is one of 1,2,4,8;
  - **index** is in a general purpose register;
  - **displacement** is part of the instruction

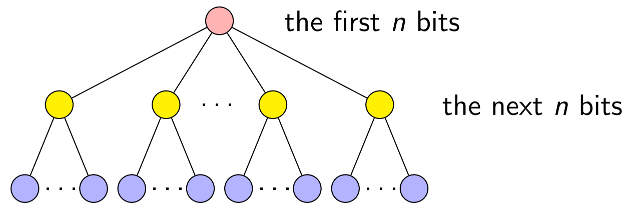
Segmentation

- In segmentation, each segment contains a **base** and **limits**
- segment number is the high order bits of the logical address
- segment number accesses the segment's **base** and **limits**
- the effective address is added to the **base**
- the effective address must be less than **limits**
- segmentation is not extensively used in modern OSs
- Segmentation is effectively turned off by using a single segment for everything
- (Actually, since segments include privilege level, we need a different segment for each ring).
- 64-bit disables segmentation

Paging

- x86 supports 4KByte, 2Mbyte, and 4Mbyte pages.
- Ethos uses on 4KByte pages
- paging must be enabled in long mode
- paging is controlled through the CRs
- but CRs can only be updated through Xen
- Hence the primary issue is the page table formation
- Xen requires that the page table in use is read-only because it needs to ensure OSs are isolated from one another

Virtual address tree mapping

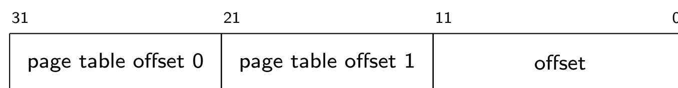


- Page tables are built at as a multiway tree
- Each level maps n-bits of virtual address
- Takes more time to do the mapping (when not in the TLB)
- Enables efficient sparse use address space
  - Kernel mapping is shared (only root level kernel entries need to be copied)
  - Userspace address can have holes in it to allow heap and stack room to grow

Part III

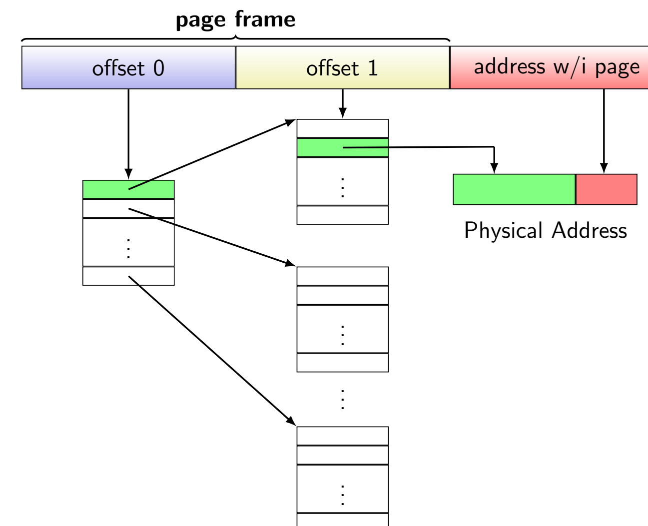
32-bit paging

32-bit paging (virtual memory addresses)



- in 32-bits, using 4Kbyte pages, using 32-bit page table entries
- each page can hold 1K page table entries (10 bits)
- So, the page access is split into 10 + 10 + 12 bits
- Two levels of page tables
- Offset is 12 bits, supporting 4K pages
- upper 20 bits of CR3 specify the upper 20 bits of the root of the page table

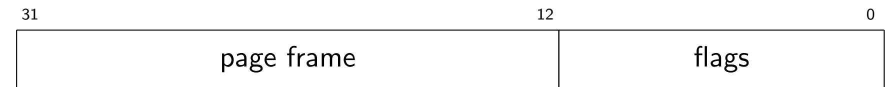
Virtual address tree mapping



Pseudo code

```
typedef pte_t PageTableT[512];
tableWalk(pageTable, virtualAddress) {
  PageTableT *pt[levels+1];
  pt[0] = pageTable;
  for (i=0; i<levels; i++)
  {
    // get appropriate bits from
    // the virtual address
    offset[i] = select(virtualAddress, i);
    // index into the pt using offset, convert
    // the page table entry to a virtual address
    pt[i+1]  = pteToVirtual(pt[i][offset[i]])
  }
  vaddr = (char *) pt[levels]
       + offset(virtualAddress);
}
```


PD/PT

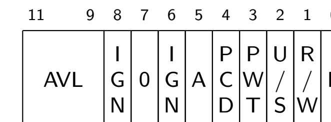


- All pages are page-aligned (low order 12-bits are zero)
- That allows the low order 12-bits to be used for flags
- Page Directory (PD) is for the root of the page table
- Page Table (PT) is for non-root of the page table
- Both PD and PT are an array of entries called PDEs and PTEs
- Since both PDE and PTE have the same basic form, we'll often use the term PTE

Flags

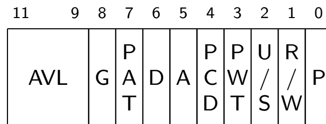
- Both PTEs and PDEs have flags
- Some of these are for software use (AVL)
- Some are for memory behavior, useful for device register manipulation (PCD/PWT)
- The three primary ones have to do with
  - Whether the page is accessible from ring 3 (U)
  - Whether the page is writable (R/W)
  - Whether the page is mapped (P)
- When translating, each PDE/PTE used in the translation must have the necessary permissions

32-bit Page Directory Entry (PDE) flags



- **AVL** Available to software (not used by hardware)
- **A** 1 if accessed by processor (used for LRU paging)
- **PCD** Page-level Cache Disable 0–page table cachable; 1–page table not cachable
- **PWT** Page level write-through 0–write back; 1–write through
- **U/S** 1–User mode (alway accessible); 0–Supervisor mode (accessible only in kernel). Applies to all pages reachable from the entry
- **R/W** 0–Read only; 1–Read/Write access
- **P** Page is present in memory

32-bit Page Table Entry (PTE) flags



- the following are applicable only on leaves of the page table tree.
  - G Global page (ignored)
  - PAT Page attribute table (caching behavior)
  - D Dirty bit. Set when writing the page
- other fields are the same as in the PDE

Permissions

- 32 bit mode is either read or read and write
- execute permission is the same as read
- so there is no way to turn off execute permissions and specify read or write permissions
- can specify no permissions by having $P = 0$
- typically, kernel pages would have $U = 0, R/W = 1, P = 1$
- typically, user space pages would have $U = 1, R/W = 1, P = 1$

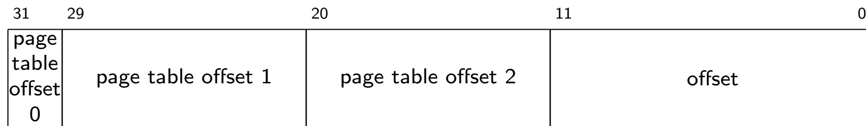
Part IV

PAE

PAE vs. non-PAE

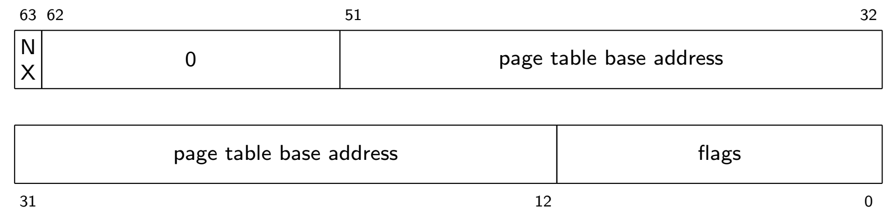
- pte_t is 64-bit in PAE vs. 32-bit in non-PAE
- So each page holds 1/2 then number of page table entries in PAE as non-PAE
- This results in a three level page table
- Same low-order 12 bit flags in both
- In addition, PAE has a No Execute (NX) as bit 63 of the PTE specifying

32-bit PAE (virtual memory address)



- 16 times the amount of physical memory (64GB max)
- but 4 GB virtual, so no longer a flat memory address space
- fits half the PTEs per page since PAE uses 64-bit PTEs
- Three levels of page tables
- Offset is 12 bits, supporting 4K pages

32-bit PAE PDE and PTE



Same as 32-bit, but adds another 32-bits to support larger address space and the NX (no execute) bit.

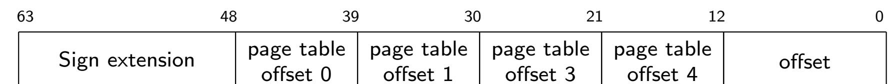
NX Bit 63, if set cannot fetch/execute instruction from this page

32-51 another 20-bits of address (giving 52 bits of physical address)

Part V

AMD64 paging

64-bit paging (virtual memory addresses)



- Sign extension enables
  - kernel to be located in addresses starting with 1,
  - userspace addresses start with 0
- Sign extension is forced, to prevent software from using the fields to encode information–i.e., pointers must have the top 17 bits either all 0s or all 1s
- Four levels of page tables
- Offset is 12 bits, supporting 4K pages

Part VI

Privileged Instructions

Privileged instructions

LLDT	Load Local Descriptor Table (286+)
LGDT	Load Global Descriptor Table (286+)
LTR	Load Task Register (286+)
LIDT	Load Interrupt Descriptor Table (286+)
MOV CRn	load and store control registers
LMSW	Load Machine Status Word (286+)
CLTS	Clear Task Switched Flag (286+).
MOV DRn	load and store debug registers
INVD	invalid cache (no write back)
WBINVD	invalid cache (with write back)
INVLPG	invalidate TLB entry
HLT	halt

Privileged instructions

RSM	return from system management mode
RDMSR	read model-specific registers
WRMSR	write model-specific registers
RTPMC	read performance modeling counter
RDTSC	read time stamp counter
RDTSCP	read serialize time stamp counter
XSETBV	enable one or more processor extended states

Part VII

X86 interrupt handling

X86 interrupt handling

- x86 has 256 interrupts
- any of these can be generated with a software interrupt instruction (`int`)
- interrupts 0-31 are hardware defined (and hardware generated)
- interrupts 32-255 are software defined interrupts
- some hardware interrupts are non-maskable (cannot ignore)
- e.g., such as a power failure
- maskable interrupts are deferred when $IF = 0$

x86 device interrupts

- PIC: Programmable Interrupt Controller (8259A)
- 16 wires, one for each device which it supports
- Each device maps to an interrupt number, INTR
- which is sent to the CPU (x86)
- use `cli` to set $IF = 0$, `sti` to set $IF = 1$
- IF also affected by: interrupt/task gates, POPF, and IRET
- and there is a non-maskable interrupt, from the NMI line of the 8259A
- immediately handled as interrupt 2 and must complete before any other interrupt is handled

Interrupts from program being executed

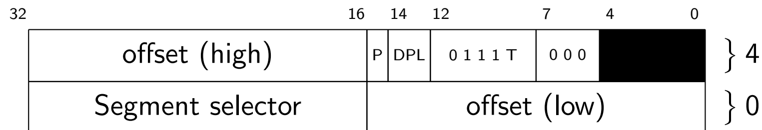
- Error condition in an instruction (e.g., divide by zero)
- Invalid address (e.g., page fault or segmentation violation)
- General Protection Fault if branched to unmapped segment

What has to happen on an interrupt?

- Can enter a different ring (typically 0, using change code segment (CS))
- Has to save registers that would be modified before software saves them
- Has to determine whether the interrupt is allowed.
- Has to ensure that interrupt handler is determined by privileged software.

Interrupt Descriptor Table (IDT)

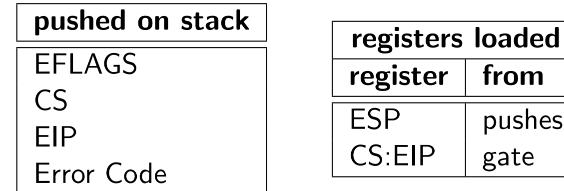
- IDT contains at most 256 entries, each entry 8 bytes.
- Each entry is called an **interrupt/trap gate**.
- `lidt` (privileged) loads the `IDTR` with [startingAddress, size]
- Interrupt number, n uses gate at address $IDTR.startingAddress + 8n$
- Each gate is as follows



- where $P = 1$ means the segment is present, DPL is descriptor privilege level (of the invoking ring), T is 1 for trap and 0 for interrupt

Interrupt (from kernel space)

- Interrupts can occur within the kernel or from user space
- if it occurs in kernel space, can use existing kernel space stack
- hardware saves registers which would otherwise change before software can save them

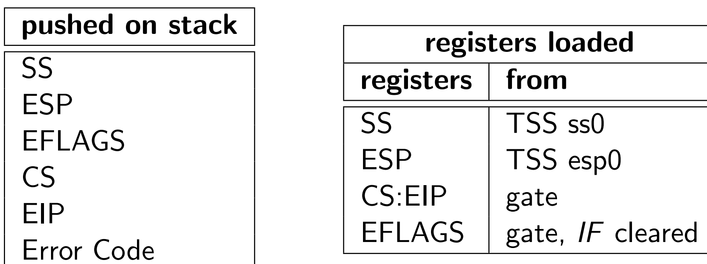


pushed on stack
EFLAGS
CS
EIP
Error Code

registers loaded	
register	**from**
ESP	pushes
CS:EIP	gate

Interrupt (from user space)

- From user space its more involved
- Need to store (userspace) ESP and SS
  (in addition to registers saved from kernel space)
- Need to set (kernel) ESP and SS



pushed on stack
SS
ESP
EFLAGS
CS
EIP
Error Code

registers loaded	
registers	**from**
SS	TSS ss0
ESP	TSS esp0
CS:EIP	gate
EFLAGS	gate, *IF* cleared

Returning from an interrupt

- uses a return from interrupt instruction `rti`
- return uses the stack, popping off CS:EIP
- where does this return to?
  - hardware interrupt: instruction after last completed instruction
  - trap: instruction after last completed instruction
  - fault: to the instruction causing the fault
  - aborts: unreliable contents

Part VIII

Clocks

Clocks

- At boot time, wall clock time is read from a clock chip
- After that, to updated the number of ticks is read since boot
- Ticks run at processor clock rate
- Tick rate slows when power management drops processor speed
- Ticks also gets messy with VMs, in which the clock is running only during part of the time the processor is running
- Finally, time keeping is not so accurate and so `ntp` is used to correct time
- RDTSC instruction is used to read the time stamp counter
- Xen provides this information through shared info page

Clock characteristics

- We want the clock to be
  - Accurate (wall time),
    so that things happen at the right time
  - Accurate (time elapse), and
    so that you can measure cost, schedule
  - Monotonically increasing
    to make it easier to reason about programs involving time
- Unfortunately, can't have it all so
  - Make small changes in time smooth
  - Have separate clocks, one monotonic and one accurate
  - Use cycle counts for performance reasons

Part IX

Memory Semantics

Barriers and Memory Semantics

- **memory semantics** ensures that the value read for a memory address A is the last value written to A.
- On Uniprocessors, **memory semantics** holds
- On Multiprocessors, because of separate caches, it does not
- Thus, when using multiprocessors (such as multicore) barriers are needed when communicating between processors
- Thus, when writing a flag to multi-core shared memory saying "Info available" an MFENCE is needed between writing (a) the info and (b) the "info available"