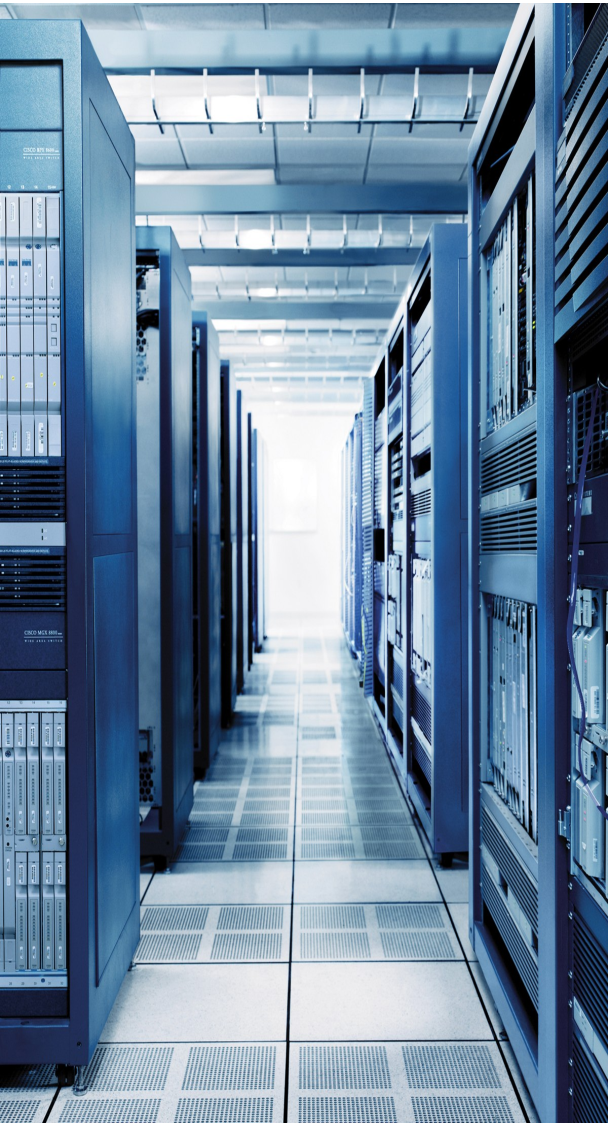


# Linux® KVM as a Learning Tool

**Duilio J. Protti**
Intel Corporation

October 24, 2008

# System Programming is hard

**But Linux KVM can make it easier (to learn)**

- Learning interrupt handling, memory segmentation, paging, etc. in a native host could be a dangerous and time-consuming process
- A pre-existent kernel makes things harder
  - Understand the kernel + understand the platform
- Solution: build your own kernel, from scratch
  - Again, this is time-consuming

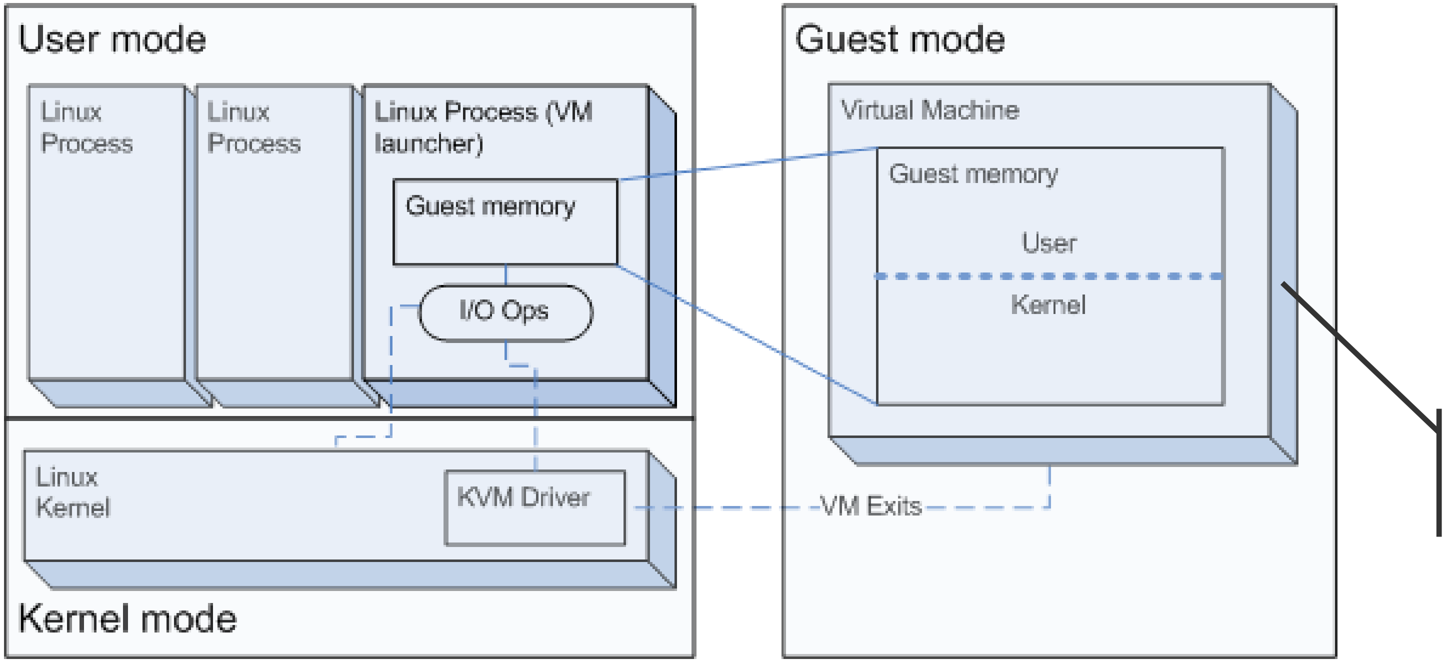
***Unless you use Linux KVM!***

# What is KVM?

- Turns the Linux host into a Virtual Machine Monitor (VMM)
  - A VMM presents guest software with an abstraction of a virtual processor…
  - … while is able to retain selective control of real processor resources, physical memory, and I/O
  - Included in mainline Linux as of version 2.6.20
- Guest software could be a stack consisting of operating system and application software
- KVM allows guest code to execute almost any instruction **directly on the real processor**, except for a few ones (I/O mainly)

# How it works?

- Linux processes have 2 execution modes: kernel and user
  - KVM adds a 3rd one: **guest mode**
- A KVM virtual machine will be "seen" as a normal process
  - A portion of code runs non-I/O guest code (guest mode)
  - A portion of code performs I/O on behalf of the guest (user mode)




**Its own 4 rings**

# What gets virtualized?

- Computer machines have
  - Memory
  - 1 or more CPU's
  - 1 or more I/O devices
- Virtual machines should have these three artifacts
- KVM controls both memory and virtual CPU's (using hardware support)
  - 3rd ingredient, I/O, is left to the programmer (e.g. **qemu-kvm**)

# How to create a VM launcher

- We don’t use qemu-kvm, we create our own (tiny one)
- KVM driver creates the node /dev/kvm for interaction with user space
  - Access this device through libkvm (API in <libkvm.h>)
- Use 5 libkvm functions
  - kvm_init
  - kvm_create
  - kvm_create_vcpu
  - kvm_create_phys_mem
  - kvm_run

# Code Walkthrough
# (VM launcher)

# 16-bit Real Mode

- Legacy mode inherited from the Intel® 8086 processor
  - Memory up to 1Mb
  - (1Mb = 2^20 bytes) => addresses requires 20 bit
  - 8086's registers are only 16-bit wide…
  - … so addresses are built by pairing two values (selector:offset) with the formula: **16 * selector + offset**
- Example, DEADH:BEEFH = EA9BFH

```
  DEAD0H
+ BEEFH
-----------
  EA9BFH
```

- A given selector value can only reference 64Kb of memory
  - Programs bigger than 64Kb must use multi-segment code

# Our first kernel

- We will keep our kernel simple and make it fit into a single segment
- x86 processors starts in real-address mode after a power-up or reset and jumps to **FFFF0H**
  - The first instruction of our kernel will be there
- Our launcher will put the kernel image in the last segment
  - Contains the FFFF0H entry point
  - Last segment at F0000H:

    Start of the last segment =

    (Maximum 8086’s memory) – (Size of a segment) =

    1Mb - 64Kb = 100000H - 10000H = F0000H

# Code Walkthrough
## (16-bit kernel)

# 32-bit Protected Mode

- The native mode of the processor
  - Provides a rich set of features
- Switch to protected-mode. Minimum data structures:
    - > IDT
    - > GDT (LDT optional)
    - > TSS (Optional for virtual machines)
    - > If paging, at least one page directory and one page table
    - > A code segment for protected mode
    - > Code modules with necessary interrupt and exception handlers
  - Minimum initialization:
    - > Load GDT (optionally also IDT)
    - > Set CR3 and CR4
    - > (Pentium® 4, Xeon®, and P6 family processors only) Set (MTRRs)
- Then set the PE flag (bit 0) in CR0 to enter protected mode

# PM Memory Management

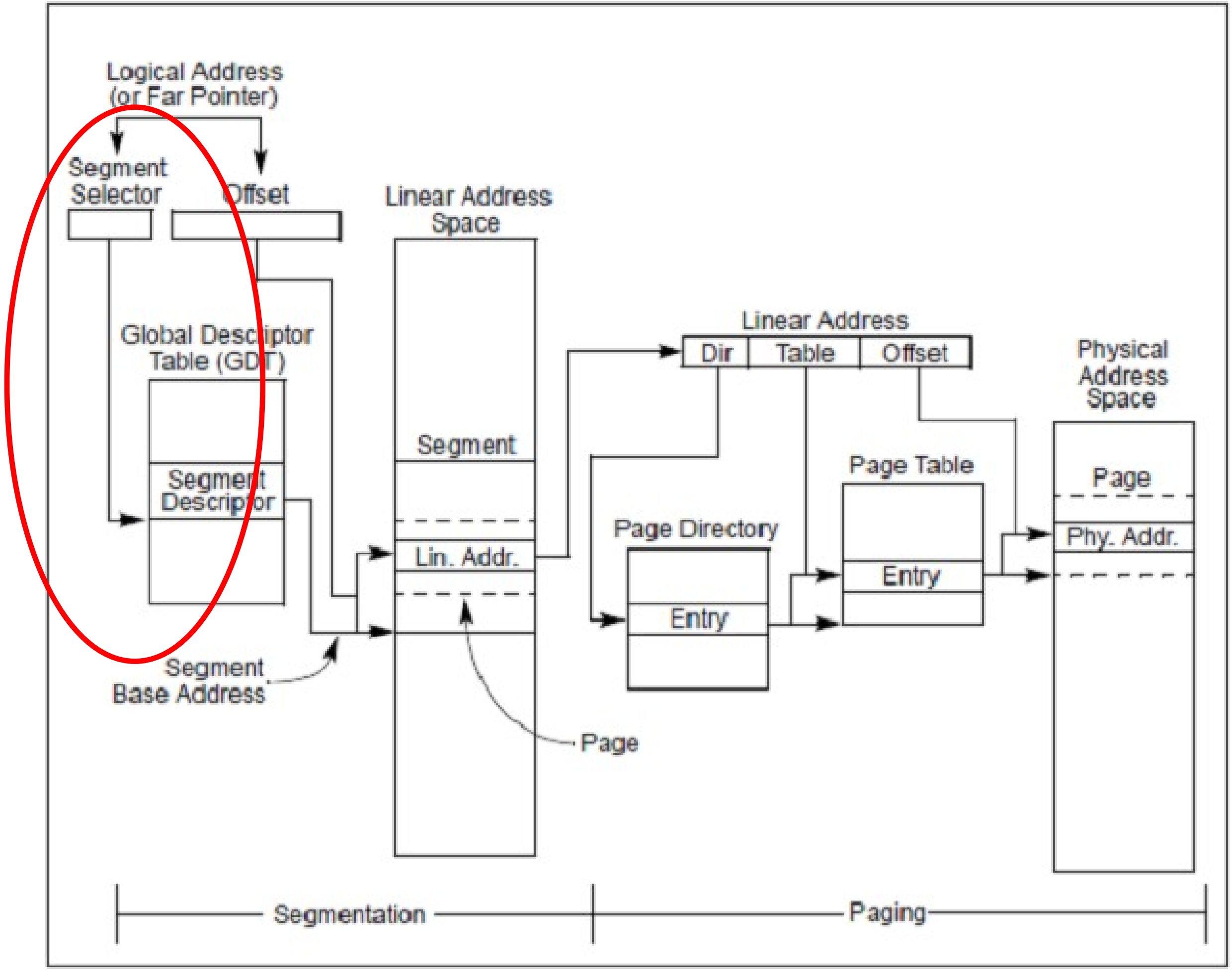



Figure 3-1, Intel® 64 and IA-32 Architectures Software Developer's Manual, Vol. 3A

- Content of segment registers are interpreted differently in PM
- Now selectors are **offsets into the GDT** (or else LDT)

# We use a Flat Memory Model

- Minimal GDT with 3 descriptors (code, data and null)
- Our code and data segments will overlap and will take the entire 4Gb

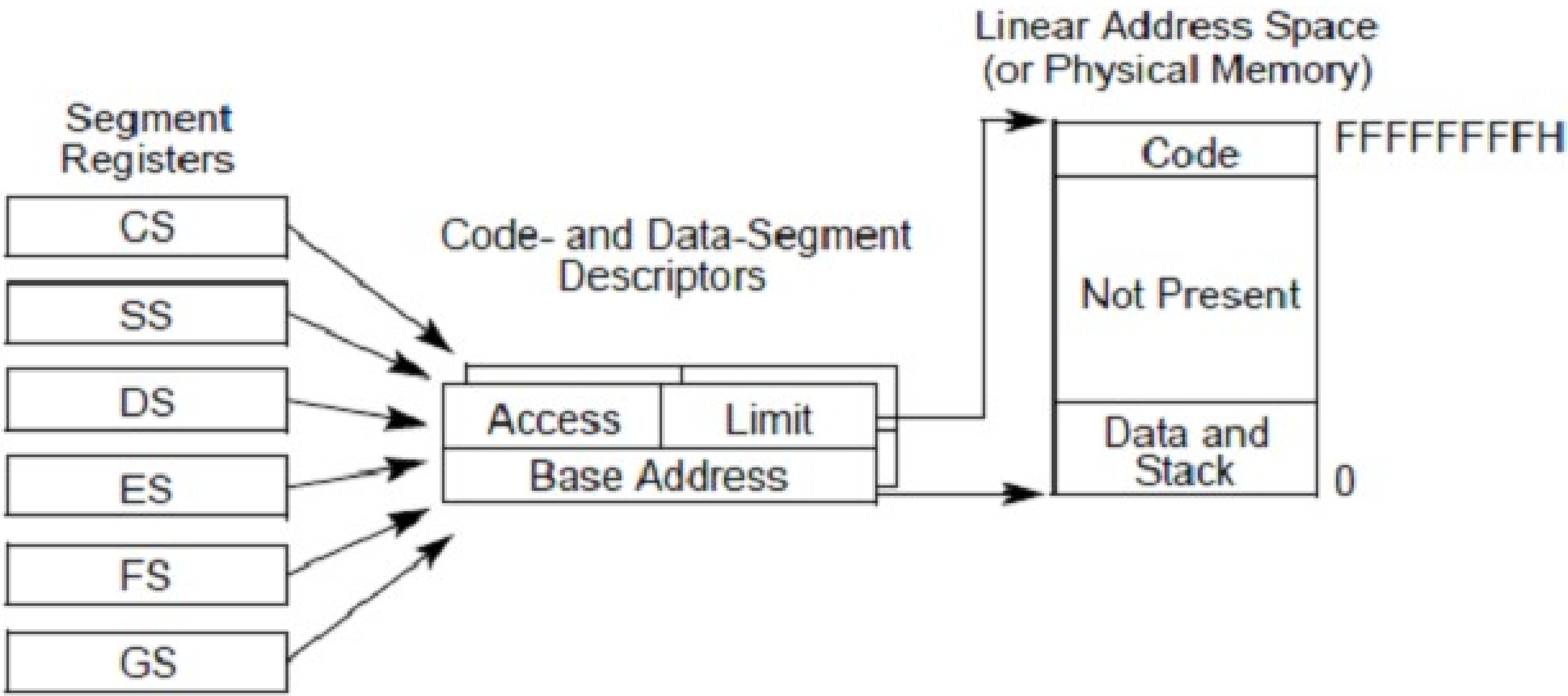



Figure 3-2, Intel® 64 and IA-32 Architectures Software Developer's Manual, Vol. 3A

# Segment Descriptors

- Each GDT entry is an 8-byte segment descriptor

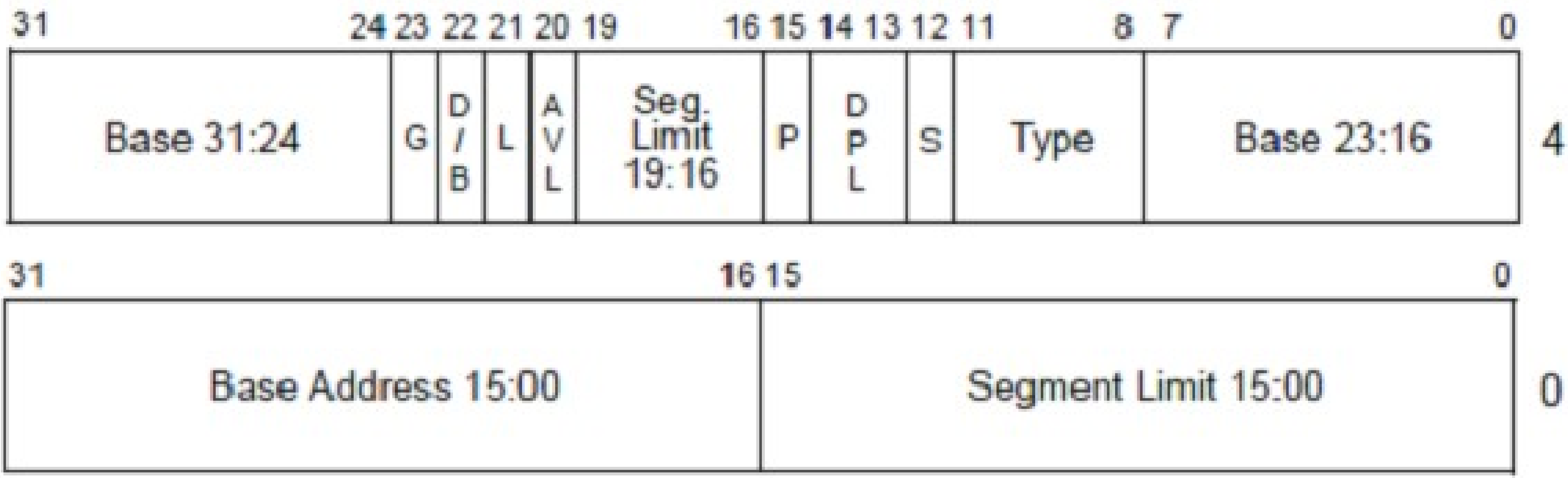



Figure 3-8, Intel® 64 and IA-32 Architectures Software Developer's Manual, Vol. 3A

# Code Walkthrough
## (32-bit kernel)

# Conclusions

- Programming in a "raw" processor
  - Realistic
  - Well documented
  - A lot of code out there (allow comparisons)
- Micro-managed
  - Micro-kernels to test specific features
- A good way to learn (by doing)

intel®