Secure Computer Systems

Hardware Support for Protection of Resources

Mustaque Ahamad, Ph.D.

Professor & Associate Director: Educational Outreach, Institute for Information Security & Privacy

Georgia Tech - College of Computing

Protecting TCB from Untrusted Applications

Georgia Tech

Before We Begin

- Going back to trusted computing base (TCB) requirements
  - TCB must be isolated from untrusted code.
- How can we meet this requirement?
  - Hardware helps us with TCB isolation
  - We also get isolation of applications from each other.
- We will focus on hardware support that makes TCB isolation possible.

Processes & Address Spaces

Georgia Tech

How Do We Meet Isolation/Tamper-proof Requirement of TCB?

Use functionality supported by hardware

- We trust hardware (should we?)

Two Basic Mechanisms Provided by Hardware

- Processor execution modes (privilege levels)
- Privileged instructions

We will Explore these in the Context of the x86 Architecture

- We will gloss over many low-level details, but goal is to understand important high level ideas

• Address space is the Unit of Protection/Isolation

Process/program executes in an address space

Processor executes instructions

- EIP holds the address of the next instruction)
- Also operands point to data operated on by the instruction
- Logically, next instruction fetched/executed after execution of current one

Addresses are logical and must be translated to (or mapped to) actual physical addresses

TCB must control the translation/mapping process to ensure that executing code can only access memory made available to it

Highly simplified idea for memory protection

- Base and bounds register can limit when user code of a program runs
- These registers can only be loaded in system mode (privileged instructions)

Supporting Address Space on Modern Processors

Georgia Tech

Memory Management in Modern Processor Architectures (Intel x86)

We are going to gloss over details but learn the basic ideas

Real, protected and long/flat modes (32 and 64-bit architectures)

Logical address vs. physical adrress

Segmentation

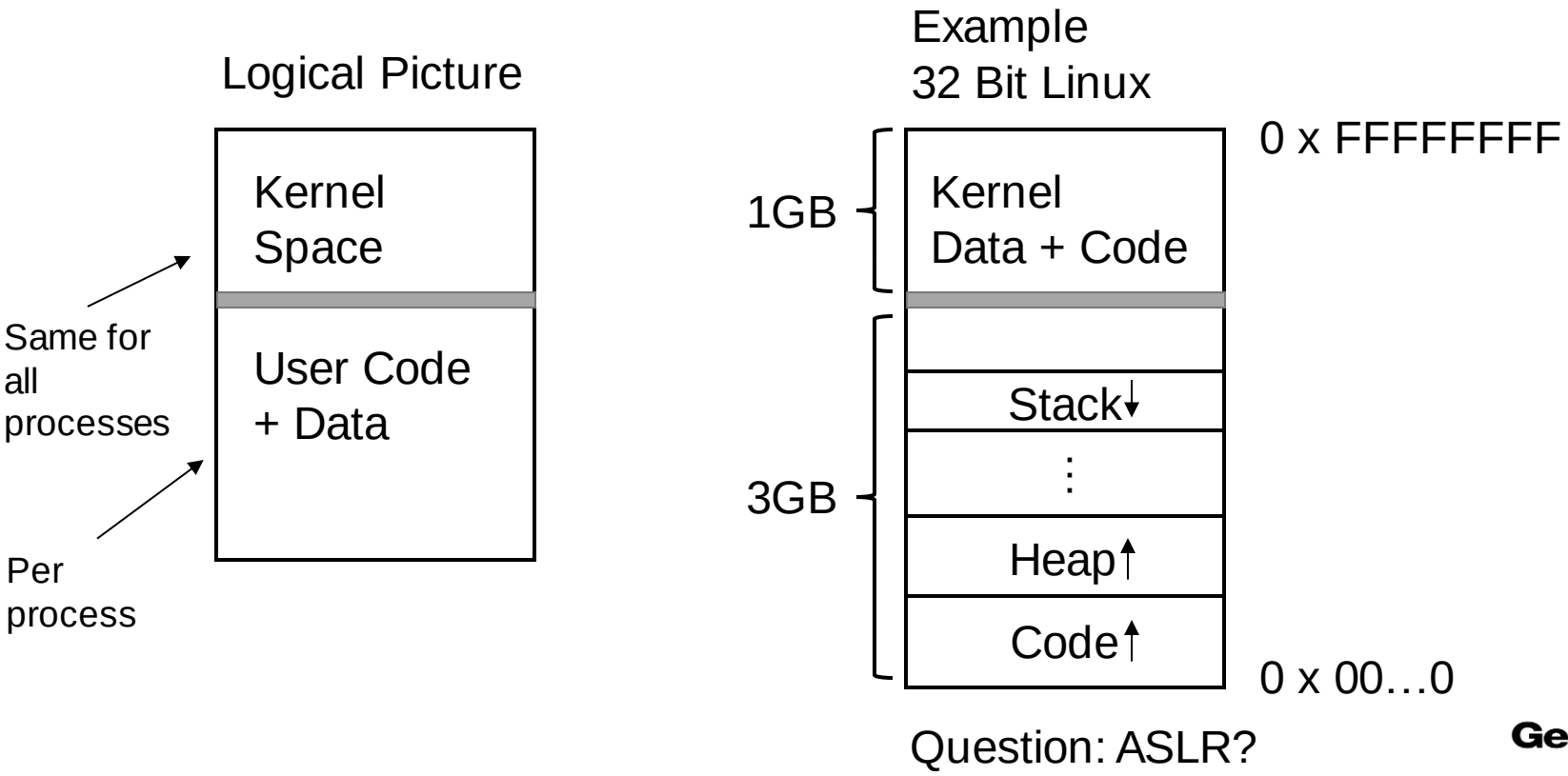
- Address space divided in variable size logical units called segments
  - Code, data, stack etc.
- Segment could be further divided pages
  - Pages are fixed size (4KB and big pages)

Logical address = (segment number, page number, page offset)

Global and local segment descriptor tables (GDT and LDT)

Georgia Tech

Process Address Space



Logical/Virtual Addresses & Their Translation

Georgia Tech

How will Address Translation Work?

Logical address = (segment number, displacement)

- Physical address = *(SGTBR+ STE * STE Size) + displacement
- Some obvious checks (e.g., displacement <= segment size)

Which descriptor table to use?

- User mode execution – LDT
- System mode execution – GDT

x86 architecture provides segment selectors

What if we also have paging?

- STE can point to page table base of the segment page table

x86 Address Translation

Georgia Tech

Segmentation + Paging

PROTECTED-MODE MEMORY MANAGEMENT

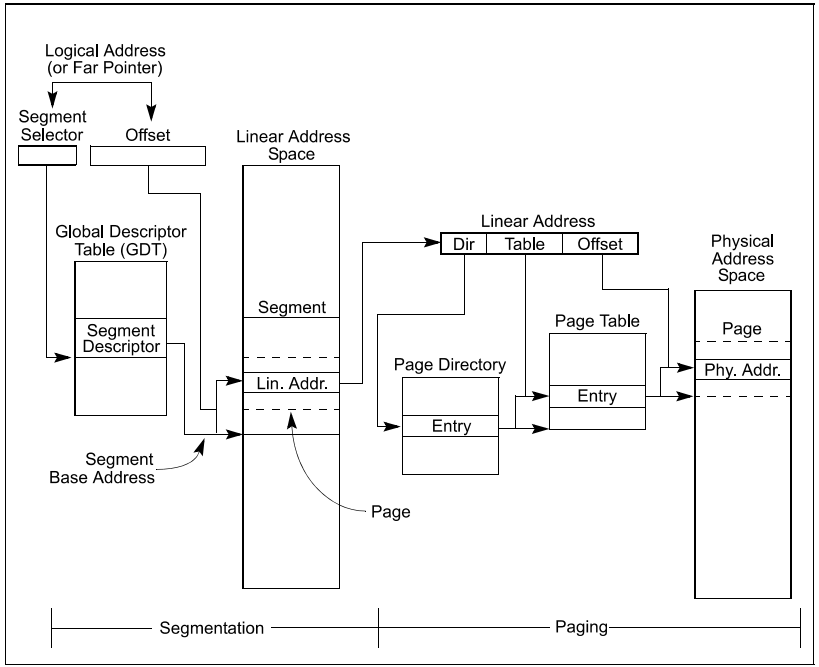


Figure 3-1. Segmentation and Paging

Page 3-2, Intel Software Developer Manual

Georgia Tech

X86 Address Translation (IA-32e)

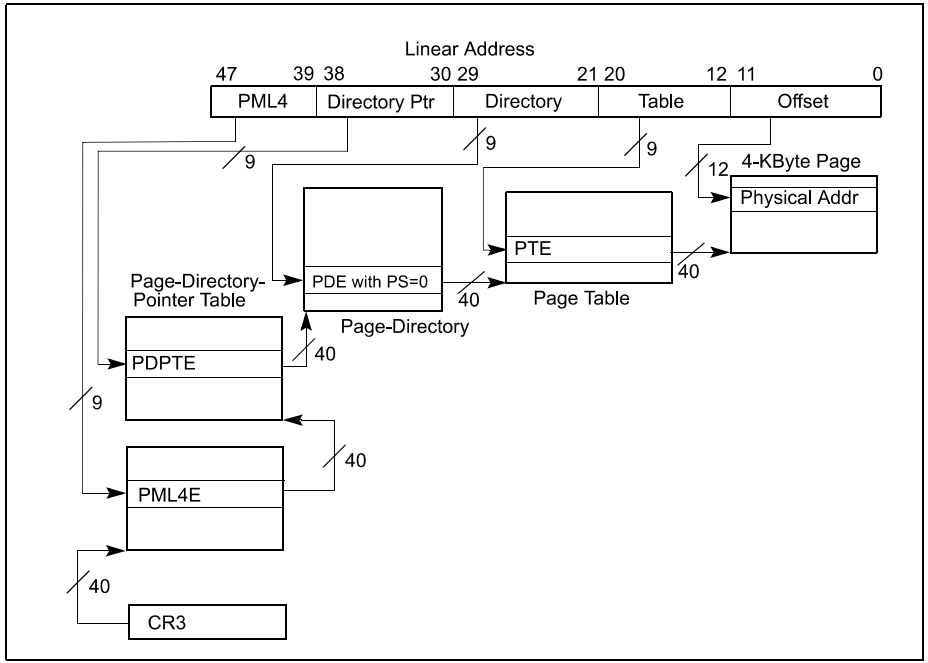


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

Page 4-20, Intel Software Developer Manual

Georgia Tech

Some Observations About Address Translation

Georgia Tech

Some Observations

Where in the address space do page/segment tables get allocated?

Who can load PTBR and STBR (e.g., CR3)?

- Special instructions that can only be executed by TCB (privileged instructions)

Do we only protect OS from user program or user programs from each other as well?

Does the system control the results of address translation?

Does translation slow memory access?

- Translation Lookaside Buffer (TLB)

- **In real-life, things are bit more complicated (e.g., segment selectors do not play an important role in address translation 64-bit architectures except few of them)**

Georgia Tech

Protecting Program Memory in the x86 Architecture

Georgia Tech

Memory Protection in the x86

Protection bits associated with segments
- Segment descriptor protection level (DPL)
- From 0 to 3, 0 is most privileged

Current protection level (CPL)
- DPL of the code segment being executed

Requestor privilege level (RPL)
- Specified in segment selector

Segment Selector

Index	TI (GDT or LDT)	RPL

Protection check
- Max (CPL, RPL) <= DPL of target

Why RPL?
- To avoid privilege escalation when kernel code executes on behalf of an application

Conforming and non-conforming code segments

Georgia Tech

Example (Section 5.6 of Intel Document)

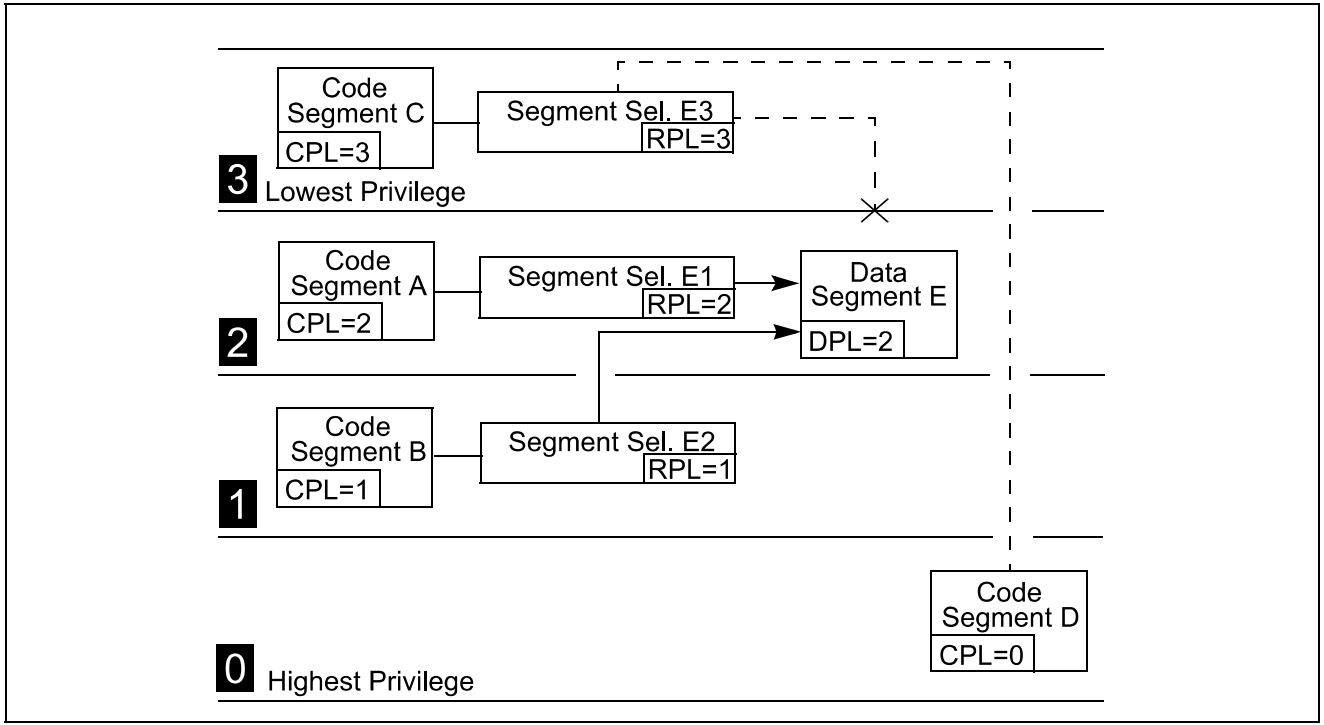


Figure 5-5. Examples of Accessing Data Segments From Various Privilege Levels

Georgia Tech

Some More Memory Protection Details

Conforming and non-Conforming Segments

Conforming Code Segments

- Transfer to more privileged code segment allows execution to continue at current privilege level
- Useful for system utilities that do not need protected system (exception handler for divide by zero)

Non-conforming Code Segments

- Transfer to different privilege segment generates a general protection fault unless call or task gate is specified
- Call gates can be used for transfer to different privilege levels (better ways to do this – system call instructions)
- All data segments are non-conforming

Page Level Protection

- **Page Protection Levels**
  - PPL of 0 and 1
  - CPL with 3 can only access PPL 1
  - Read-write protection
  - Execute disable protection
- **Segment & Page Protections can be Combined**
  - Selection for pages in a segment

Changing Privilege Level and System Calls

Georgia Tech

Call Gates

- Specifies code segment to be accessed
- Privilege level required for caller
- Entry point for procedure in called code segment
- Number and size of parameters when stacks are to be switched
- Privilege check

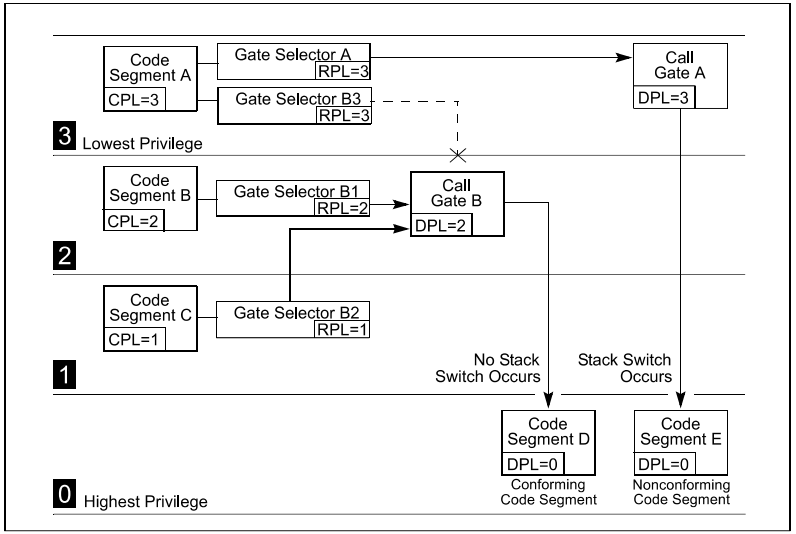


Figure 5-12. Example of Accessing Call Gates At Various Privilege Levels

Page 5-17, Intel Developer Doc

Georgia Tech

SYSENTER & SYSEXIT (SYSCALL & SYSRET)

Introduced for Low Overhead System Call Implementation

- SYSENTER used for transfer from level 3 to level 0
- SYSEXIT from level 0 to level 3
- Not paired instructions

For SYSENTER, target fields are generated using the following sources:

- **Target code segment** — Reads this from IA32_SYSENTER_CS.
- **Target instruction** — Reads this from IA32_SYSENTER_EIP.
- **Stack segment** — Computed by adding 8 to the value in IA32_SYSENTER_CS.
- **Stack pointer** — Reads this from the IA32_SYSENTER_ESP.

For SYSEXIT, target fields are generated using the following sources:

- **Target code segment** — Computed by adding 16 to the value in the IA32_SYSENTER_CS.
- **Target instruction** — Reads this from EDX.
- **Stack segment** — Computed by adding 24 to the value in IA32_SYSENTER_CS.
- **Stack pointer** — Reads this from ECX.

From page 5-21 of Intel System Developer Manual

Georgia Tech

Privileged Instructions

Georgia Tech

Privileged Instructions

- LGDT — Load GDT register.
- LLDT — Load LDT register.
- LTR — Load task register.
- LIDT — Load IDT register.
- MOV (control registers) — Load and store control registers.
- LMSW — Load machine status word.
- CLTS — Clear task-switched flag in register CR0.
- MOV (debug registers) — Load and store debug registers.
- INVD — Invalidate cache, without writeback.
- WBINVD — Invalidate cache, with writeback.
- INVLPG —Invalidate TLB entry.
- HLT— Halt processor.
- RDMSR — Read Model-Specific Registers.
- WRMSR —Write Model-Specific Registers.
- RDPMC — Read Performance-Monitoring Counter.
- RDTSC — Read Time-Stamp Counter.

Attacks Against Hardware and Operating Systems

Georgia Tech

Attacks Against TCB Isolation

- **Should we trust hardware?**
  - Row hammer attack exploits DRAM vulnerability
  - Bit flips can result in access to memory which should not be allowed (write access to own page table)

- **Software Attacks Targeting TCB**
  - Kernel rootkits run in kernel mode and can change OS memory
  - NVD CVE-2019-6218: …..a malicious application may be able to execute arbitrary code with kernel privileges (input validation issue in iOS, macOS etc.).

Memory Protection and Software Security

Georgia Tech

System-level Software Security Techniques

- **Address Space Layout Randomization (ASLR)**
  - Makes it hard for attacker to guess where some code or variable is located
  - Ways to get around it
- **Non-executable Stack**
  - NX (never execute) or disable execute (DX) bits
  - Eliminates vulnerabilities where malicious code is executed from stack
  - Buffer overflow problems go away?
    - See example: return-to-libc attack

Summary

Georgia Tech

Summary

Logical/virtual addresses get translated to physical addresses

Memory protection works to separate OS and user code execution

Importance of privileged instructions for memory protection

Should we really trust the hardware and feel confident about memory protection?

- Flipping bits in memory (Row hammer attack paper & Google Projectzero blogpost)
- Intel hardware vulnerabilities (ZombieLoad ??)