Operating Systems

Virtual Memory Basics

Daniel Gruss

2023-10-08

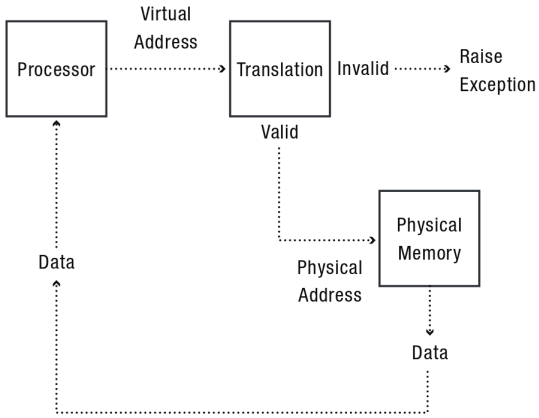
Table of contents

Address Translation

Address Translation

- OS in control of address translation
- enables number of advanced features
- programmers perspective:
  - pointers point to objects etc.
  - transparent: it is not necessary to know how memory reference is converted to data

Address Translation - Idea / Overview



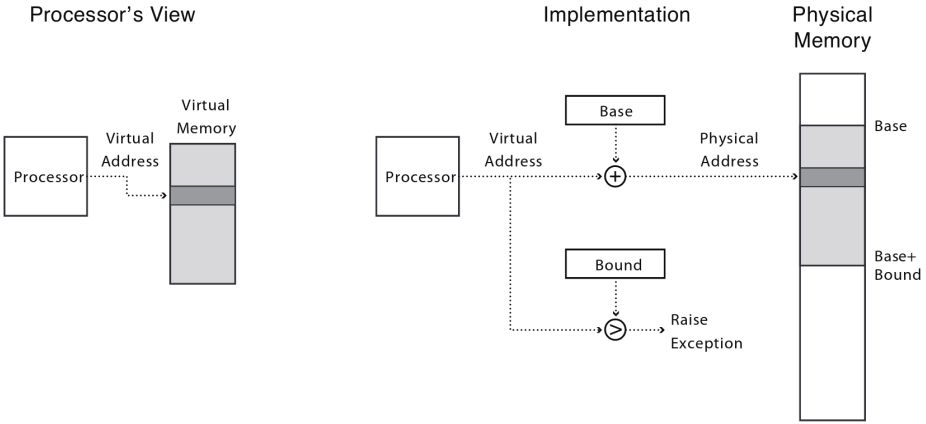
Address Translation Concepts

- Memory protection
- Memory sharing
  - Shared libraries, interprocess communication
- Sparse address space
  - Multiple regions for dynamic allocation (heaps/stacks)

Address Translation Concepts

- Efficiency
  - Flexible Memory placement
  - Runtime lookup
  - Compact translation tables
- Portability

Base-Limit or Base and bounds



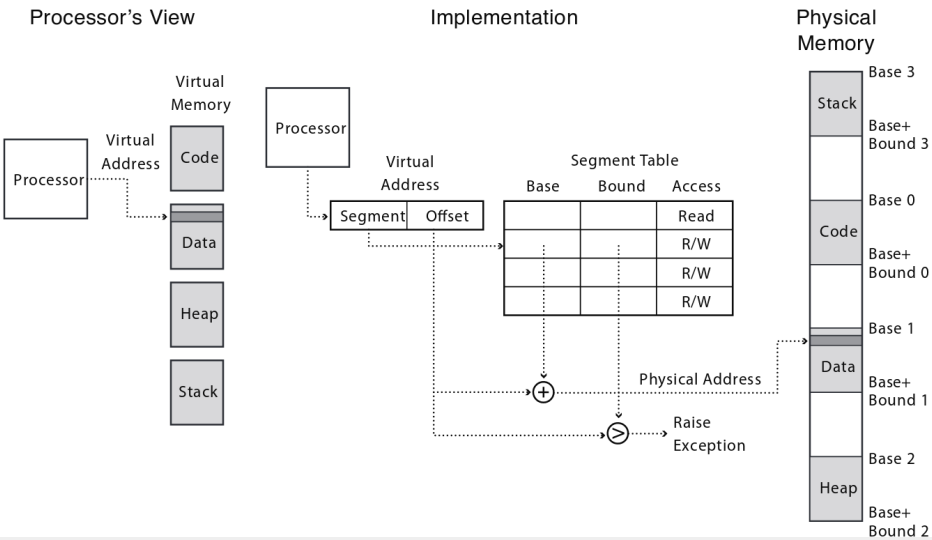
Virtually Addressed Base and Bounds

- Virtual Address: from **0** to an upper **bound**
- Physical Address: from **base** to **base + bound**
- what is saved/restored on a process context switch?

Segmentation

- Small Change: multiple pairs of base-and-bounds registers
- Segmentation
- Each entry controls a portion of the virtual address space

Segmentation



Segmentation

- Segment is a contiguous region of virtual memory
- Each process has a segment table (in hardware)
  - Entry in table = segment
- Segment can be located anywhere in physical memory
  - Each segment has: start, length, access permission

Segmentation

- Segmented Memory has gaps!
- no longer contiguous region - set of regions
- code and data not adjacent - neither in virtual nor in physical address space
- What if: program tries to load data from gap?
- Segmentation Fault (trap into OS)
  - correct programs will not generate references outside valid memory
  - trying to read or write data that does not exist: bug-indication

Shared Memory

- Processes can share segments
  - Same start, length, same/different access permissions
- Usage:
  - sharing code (shared libraries)
  - interprocess communication
  - copy on write

Copy on Write

Special kind of shared memory (after fork)

- two processes, both running the same program and almost same data
- makes sense not to copy everything
- we just need to be made aware if a process writes to a segment and changes the content
- reading does not present any problems
- how do we know when a process writes to a segment?

$\rightarrow$ set segment read only

Fork and Copy on Write

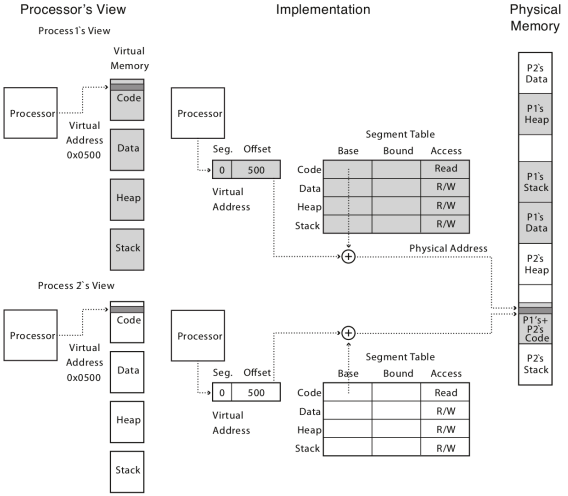
Fork:

- Copy segment table into child
- Mark parent and child segments read-only
- Start child process; return to parent

Parent/Child try to write:

- trap into kernel
- make a copy of the segment and resume

Copy on Write



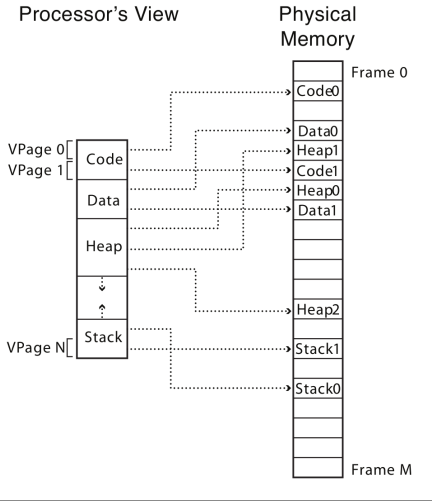
Zero-on-Reference

- How much physical memory needed for stack or heap?
  - Only what is currently in use
- When program uses memory beyond end of stack
  - Segmentation fault into OS kernel
  - Kernel allocates some memory
    - How much?
  - Zeros the memory
    - avoid accidentally leaking information!
  - Modify segment table
  - Resume process

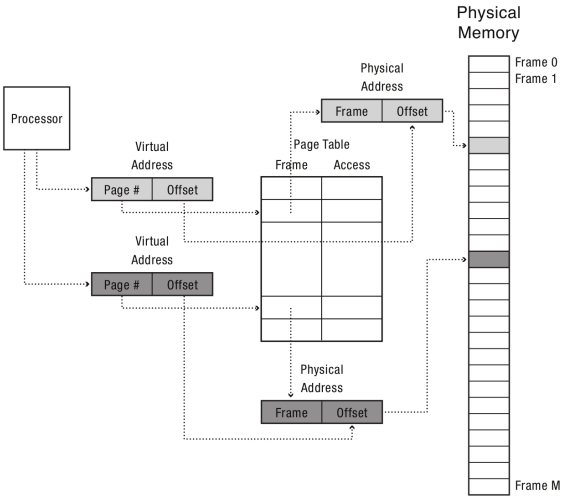
Paged Translation

- Manage memory in fixed size units, or pages
- Finding a free page is easy
  - Bitmap allocation: 0011111100000001100
  - Each bit represents one physical page number / one physical page frame
- Each process has its own page table
  - Stored in physical memory
  - Hardware registers
    - pointer to page table start
    - page table length

Logical View of Page Table Address Translation



paging - implementation



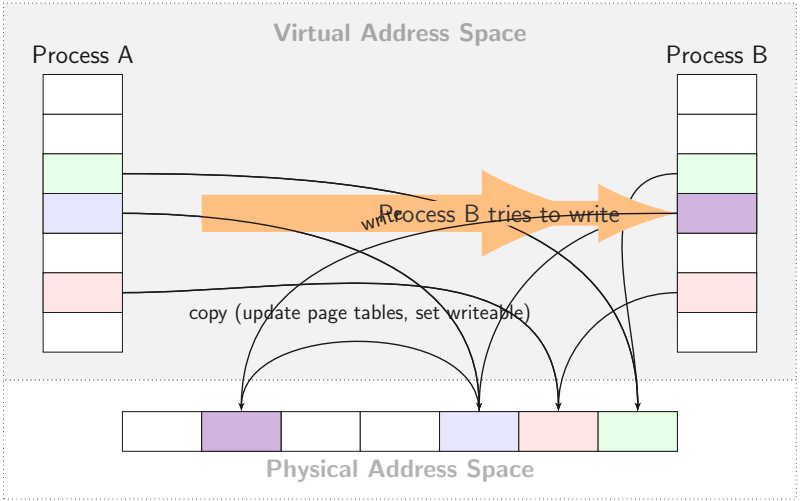
Paging Questions

- With paging, what is saved/restored on a process context switch?
  - Pointer to page table, size of page table
  - Page table itself is in main memory
- What if page size is very small?
- What if page size is very large?
  - Internal fragmentation: if we don't need all of the space inside a fixed size chunk

Paging and Copy on Write

- Can we share pages between processes (similar as segments before)?
  - Set entries in both page tables to the same physical page number
  - Need core map of physical page numbers to track which processes are pointing to which physical page numbers (e.g. *reference count*)

Copy-on-Write on Unix/Linux



Demand Paging

- Can I start running a program before its code is in physical memory?
  - Set all page table entries to invalid
  - When a page is referenced for first time, kernel trap
  - Kernel brings page in from disk
  - Resume execution
  - Remaining pages can be transferred in the background while program is running

Scheduling a Process (with Demand Paging)

- Only load what's required
- Initially start with no pages in memory
- Process will be scheduled eventually. What happens?
  - a page fault will occur when fetching the first instruction
  - further page faults for stacks and data
  - after a while, things will stabilize
- The principle of locality ensures that

Prepaging

Prepaging as an optimization

- If it is known upon scheduling which pages will be required ...
  - page referenced by instruction pointer, stack pointer, etc.
- ... load required pages into RAM ahead of time

$\rightarrow$ may lower page fault frequency

Sparse Address Spaces

- Every process needs an address space.
- What if virtual address space is large?
  - 32-bits, 4KB pages $\rightarrow$ 1 million page table entries
  - 64-bits $\rightarrow$ 4 quadrillion page table entries

Multi-level Translation

- Tree of translation tables
  - Paged segmentation
  - Multi-level page tables
  - Multi-level paged segmentation

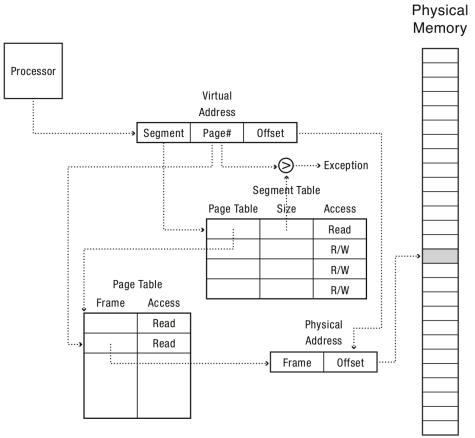
Multi-level Translation

- Fixed-size page as lowest level unit of allocation
  - Efficient memory allocation (compared to segments)
  - Efficient for sparse translation tree (compared to simple paging)
  - Efficient disk transfers (fixed size units, page size multiple of disk sector)
  - Easier to build translation lookaside buffers
  - Efficient reverse lookup (from physical $\rightarrow$ virtual)
  - Fine granularity for protection/sharing

Paged Segmentation

- Process memory is segmented
- Segment table entry:
  - Pointer to page table
  - Page table length (# of pages in segment)
  - Access permissions
- Page table entry:
  - Physical page number
  - Access permissions
- Share/protection at either page or segment-level

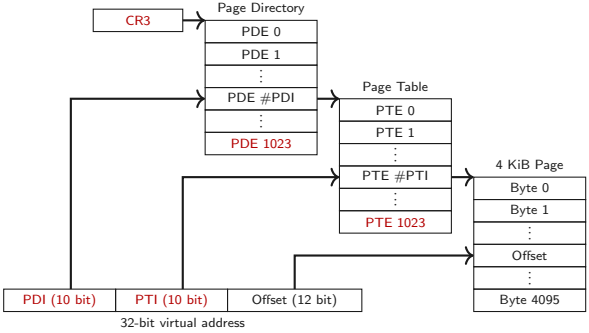
Paged Segmentation



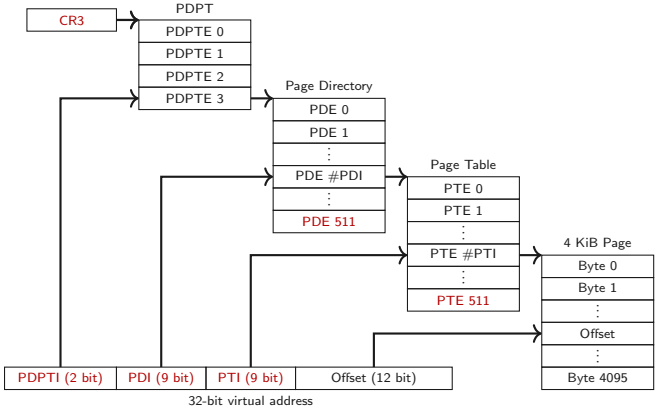
Question

- With paged segmentation, what must be saved/restored across a process context switch?

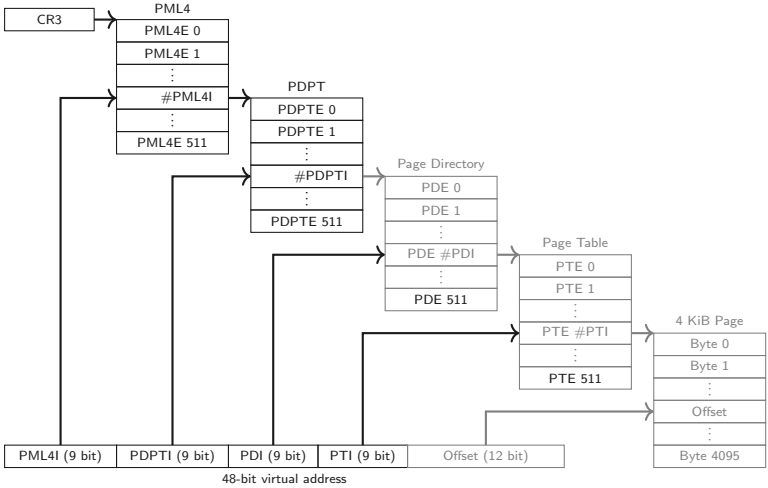
Paging: x86-32 with page size 4 KiB



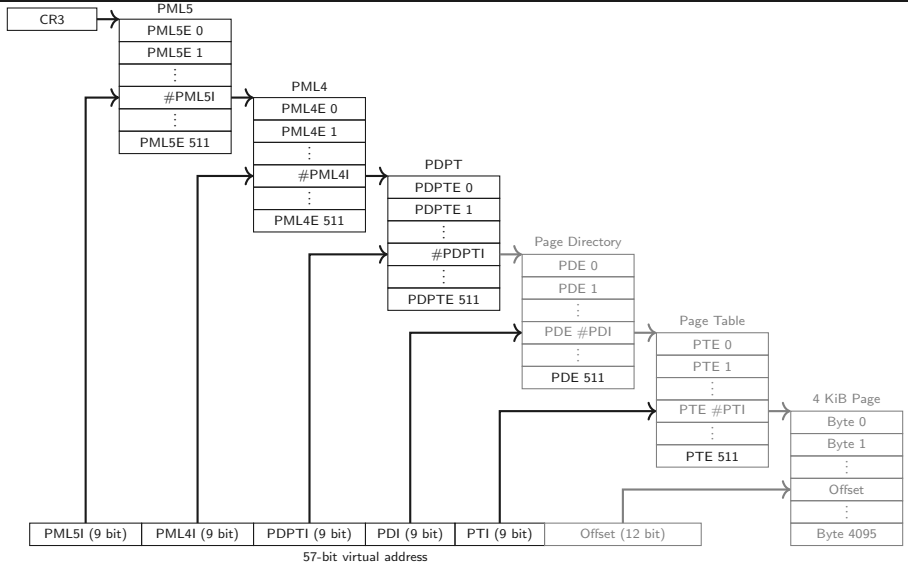
Paging: x86-32-PAE with page size 4 KiB



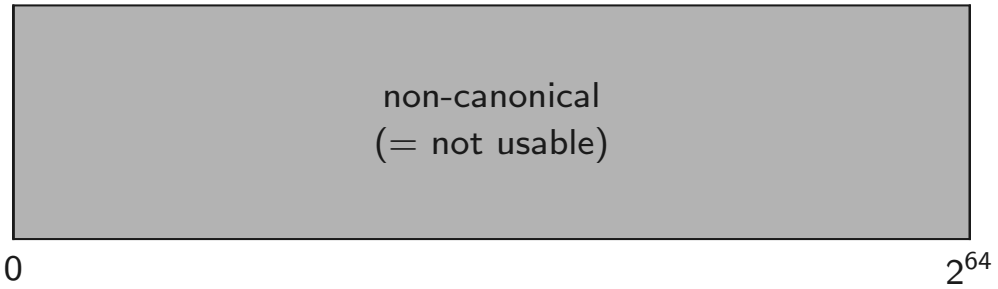
Paging: x86-64 with page size 4 KiB



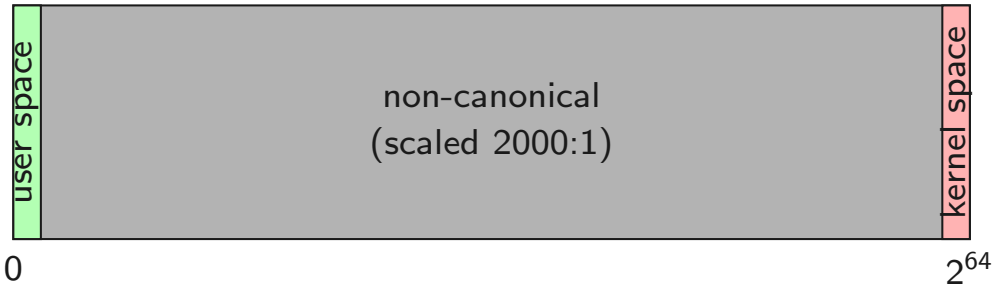
Paging: x86-64 with PML5 and page size 4 KiB



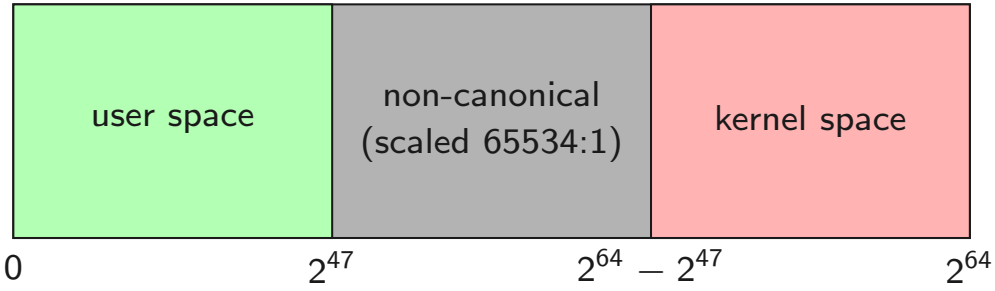
x86-64 Memory Layout (with PML4)



x86-64 Memory Layout (with PML4, scaled)



x86-64 Memory Layout (with PML4, scaled)



Address Translation on x86 processors

Intel Pentium

- Segmentation and paging
- 16 K segments, each 4 GB
  - Few segments
  - Large segments

LDT - GDT

- Local Descriptor Table LDT
  - for each process
  - local segments (Code, Data, Stack)
- Global Descriptor Table GDT
  - for system segments
  - also for kernel

Segment Registers

- 6 segment registers
  - CS: Selector for Code Segment
  - DS: Selector for Data Segment
  - ES: Selector for Data Segment
  - FS: Selector for Data Segment
  - GS: Selector for Data Segment
  - SS: Selector for Stack Segment

Segment Selector

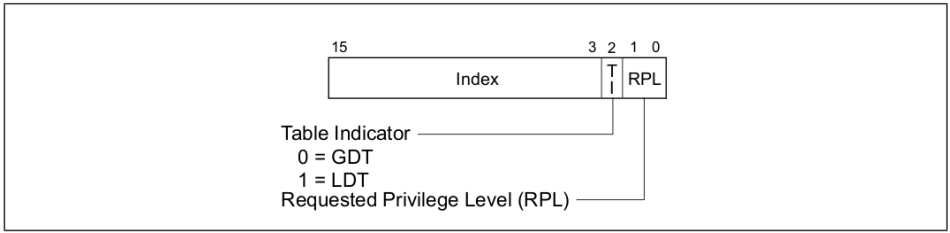


Figure 3-6. Segment Selector

- Null Segment at index 0 $\rightarrow$ cannot be used
- Modifying a segment register loads corresponding descriptor into an internal CPU register

Hidden Part of Segment Registers

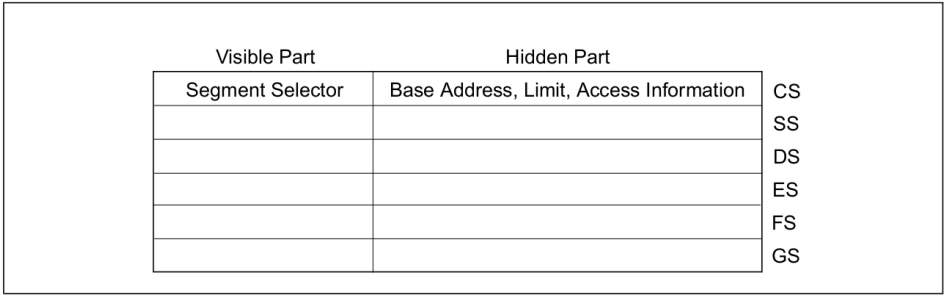
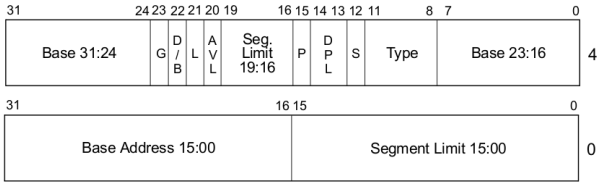


Figure 3-7. Segment Registers

Segment Descriptor



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

Figure 3-8. Segment Descriptor

Address translation

- we start with (selector, offset)
- CPU looks for correct descriptor in internal registers
- selector 0 or segment swapped out: interrupt
- offset exceeds segment size: interrupt
- add base field to offset
  - check limits of course
- result: linear address
- paging turned off: linear address is physical address

Address translation

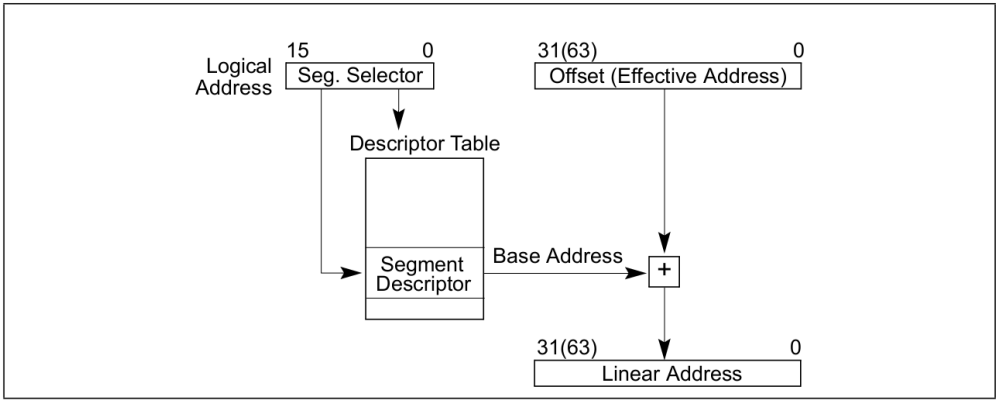


Figure 3-5. Logical Address to Linear Address Translation

Combining Segments and Paging

OSes today have only a very small number of segments:

- 1 for user code
- 1 for user data
- 1 for user thread local storage
- 1 for kernel code
- 1 for kernel data
- 1 for kernel core local storage

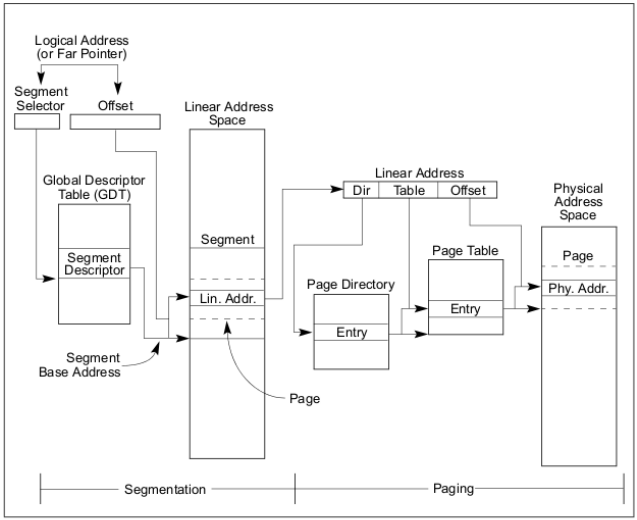


Figure 3-1. Segmentation and Paging

Segments Today

- x86-64 requires segment base to be 0 and limit to be unlimited
- not even used anymore to separate code and data
- most OSes today only use segments to determine the privilege level

Take Aways

Virtual memory

- is based on Segmentation and Paging
- enables effective protection mechanisms
- enables sparse address spaces