

Operating Systems

Virtual Memory Basics

Daniel Gruss

2024-03-07

1. Address Translation

First Idea: Base and Bound

Segmentation

Simple Paging

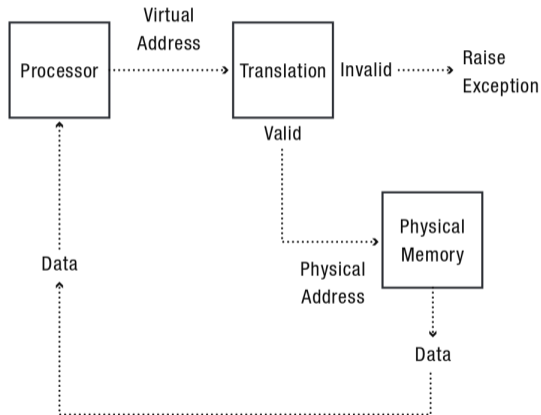
Multi-level Paging

2. Address Translation on x86 processors

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

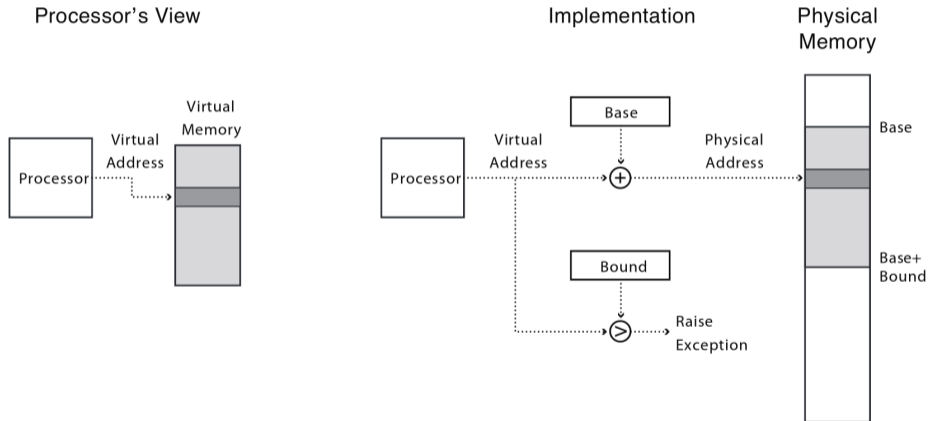
Address Translation - Idea / Overview



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

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

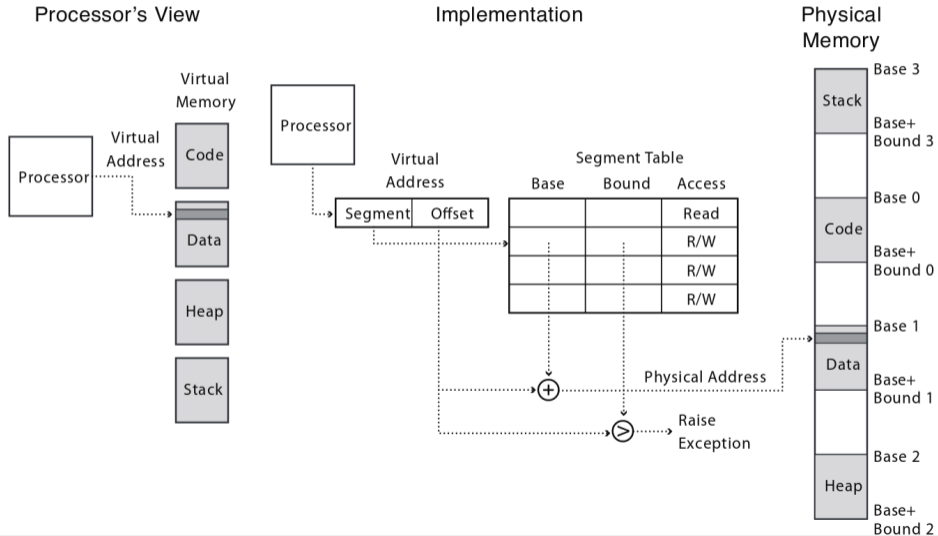
Base-Limit or Base and bounds



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

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

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

- 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

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

→ set segment read only

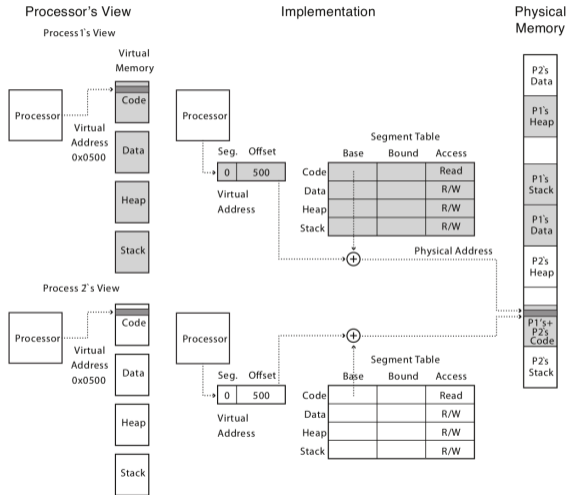
Fork:

- 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

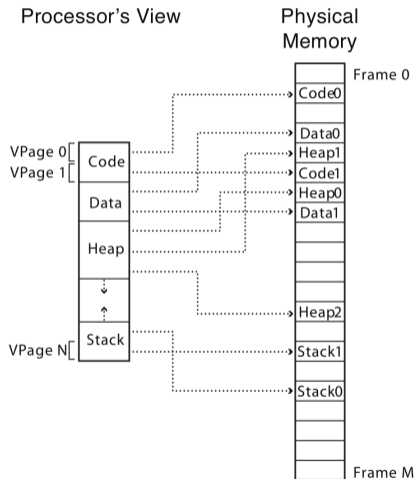
Copy on Write

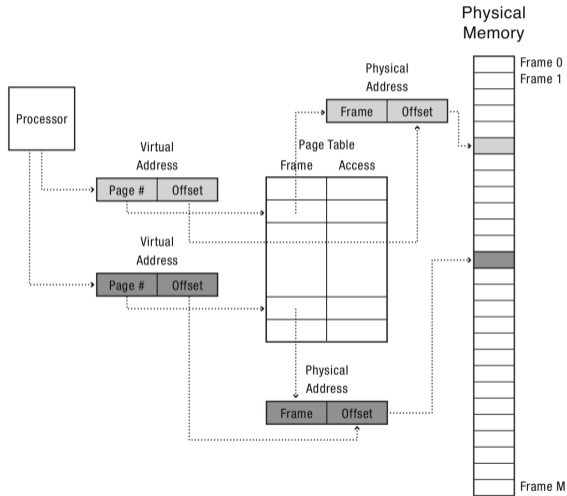


- How much physical memory needed for stack or heap?
 - Only what is currently in use
- When program uses memory beyond end of stack
 - Segmentation fault into OS kernel
 - Kernel allocates some memory
 - How much?
 - Zeros the memory
 - avoid accidentally leaking information!
 - Modify segment table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

- 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

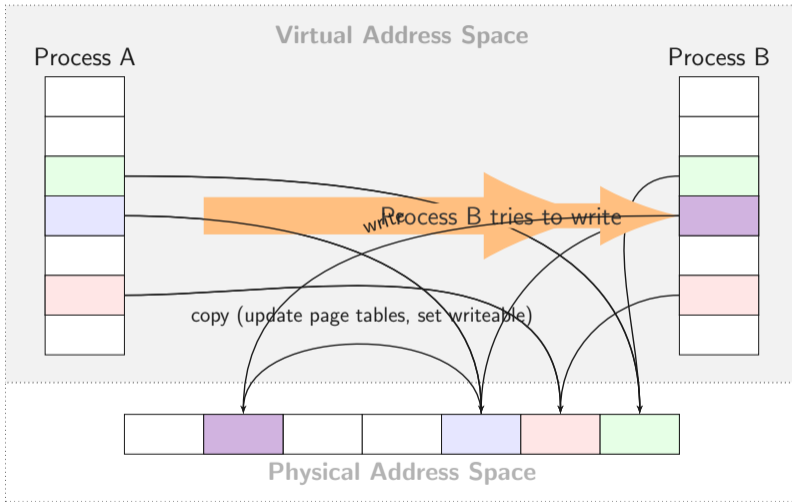




- With paging, what is saved/restored on a process context switch?
 - Pointer to page table, size of page table
 - Page table itself is in main memory
- What if page size is very small?
- What if page size is very large?
 - Internal fragmentation: if we don't need all of the space inside a fixed size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

- 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



- 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

- 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 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

→ may lower page fault frequency

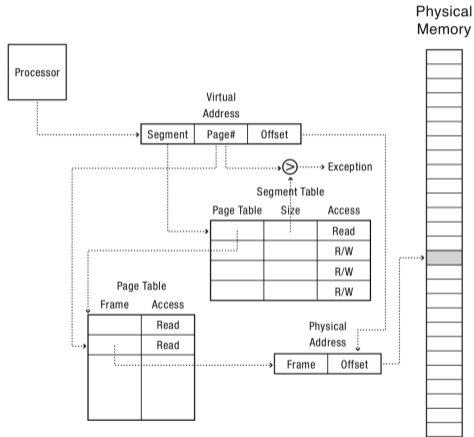
- Every process needs an address space.
- What if virtual address space is large?
 - 32-bits, 4KB pages → 1 million page table entries
 - 64-bits → 4 quadrillion page table entries

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

- 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

- 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

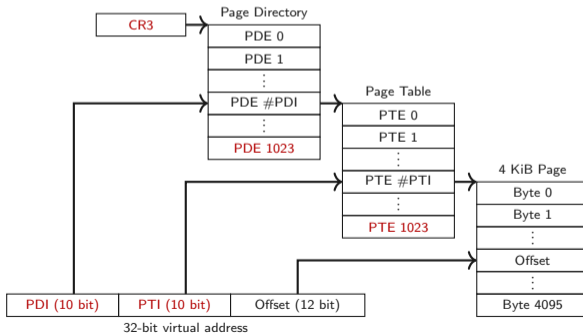
Paged Segmentation



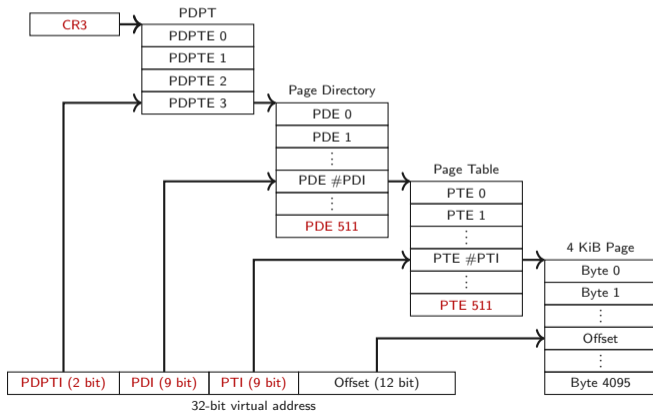


- With paged segmentation, what must be saved/restored across a process context switch?

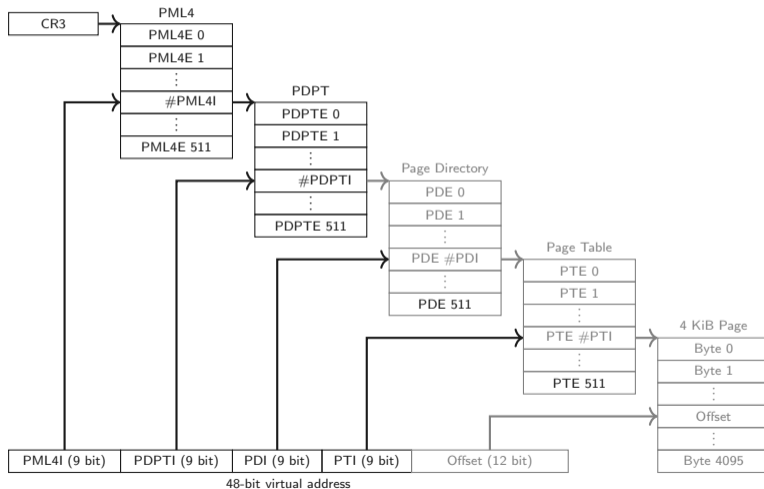
Paging: x86-32 with page size 4 KiB



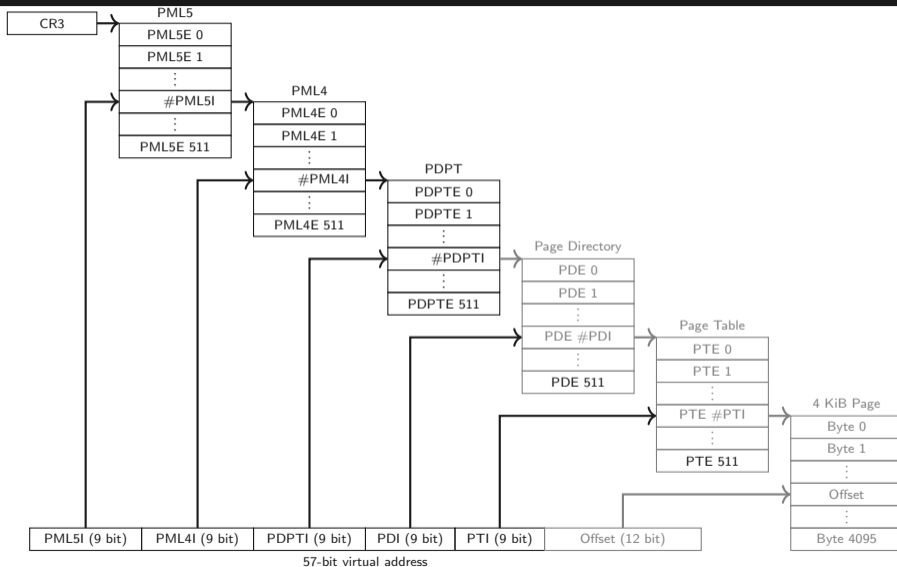
Paging: x86-32-PAE with page size 4 KiB

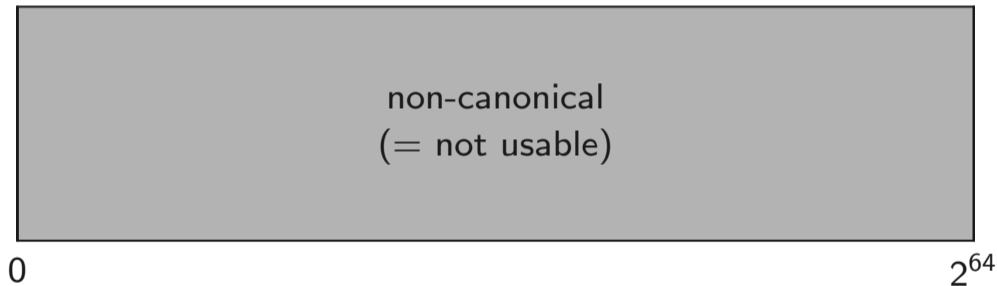


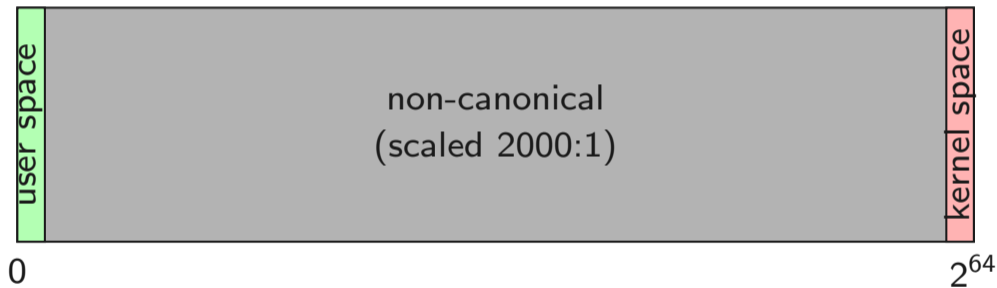
Paging: x86-64 with page size 4 KiB



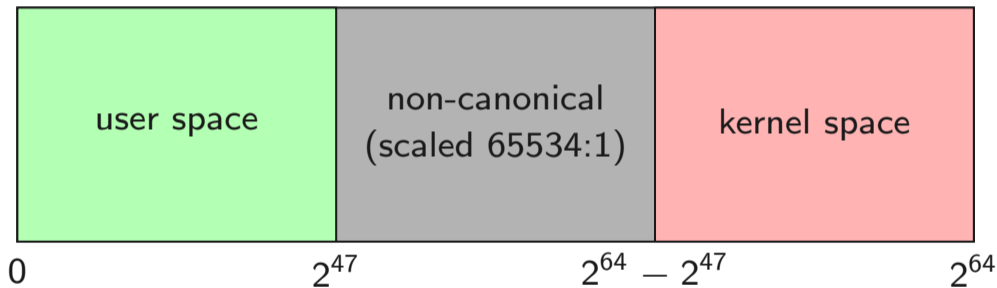
Paging: x86-64 with PML5 and page size 4 KiB







x86-64 Memory Layout (with PML4, scaled)



Address Translation on x86 processors



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

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

- 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

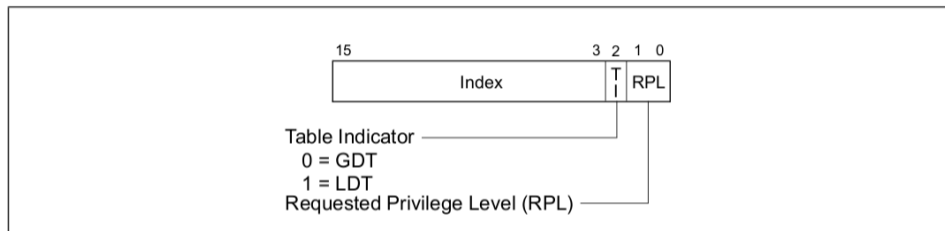


Figure 3-6. Segment Selector

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

Visible Part		Hidden Part	
Segment Selector	Base Address, Limit, Access Information		
			CS
			SS
			DS
			ES
			FS
			GS

Figure 3-7. Segment Registers

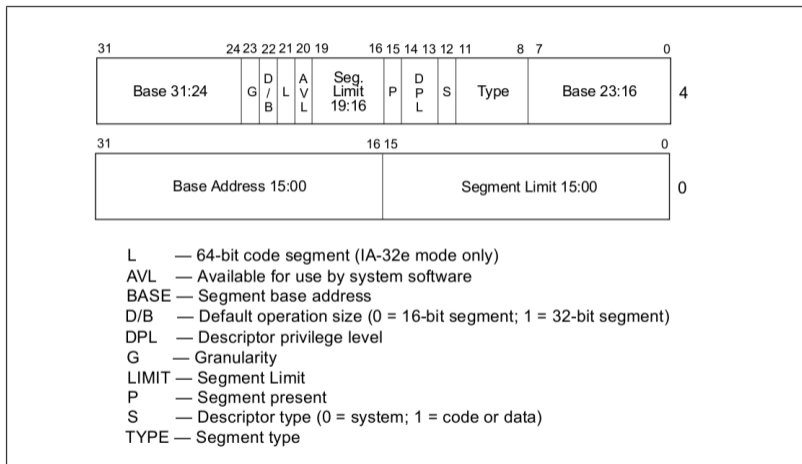


Figure 3-8. Segment Descriptor

- 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

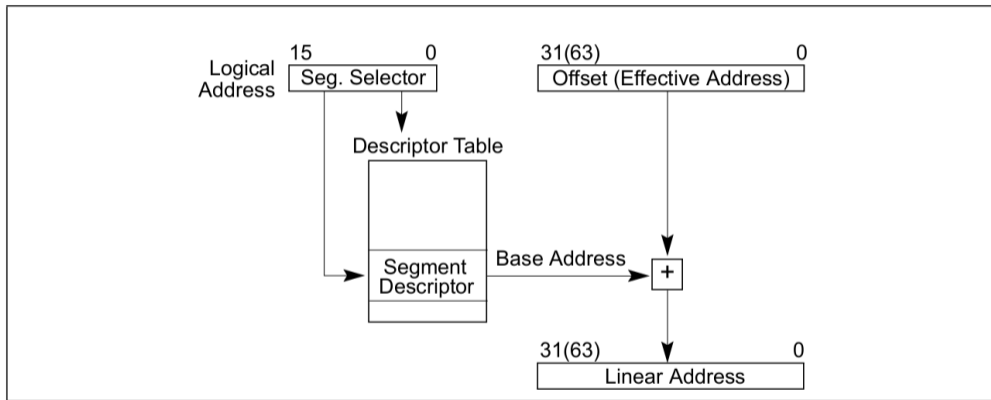
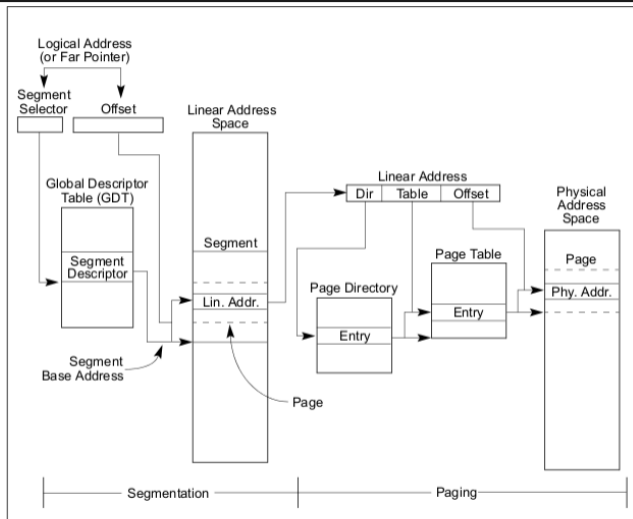


Figure 3-5. Logical Address to Linear Address Translation

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



- 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

Virtual memory

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

