CSE 306 Operating Systems
Virtual Memory

YoungMin Kwon

Memory Management

- Every programmer’s ideal memory
  - Private
  - Infinitely large
  - Infinitely fast
  - Nonvolatile

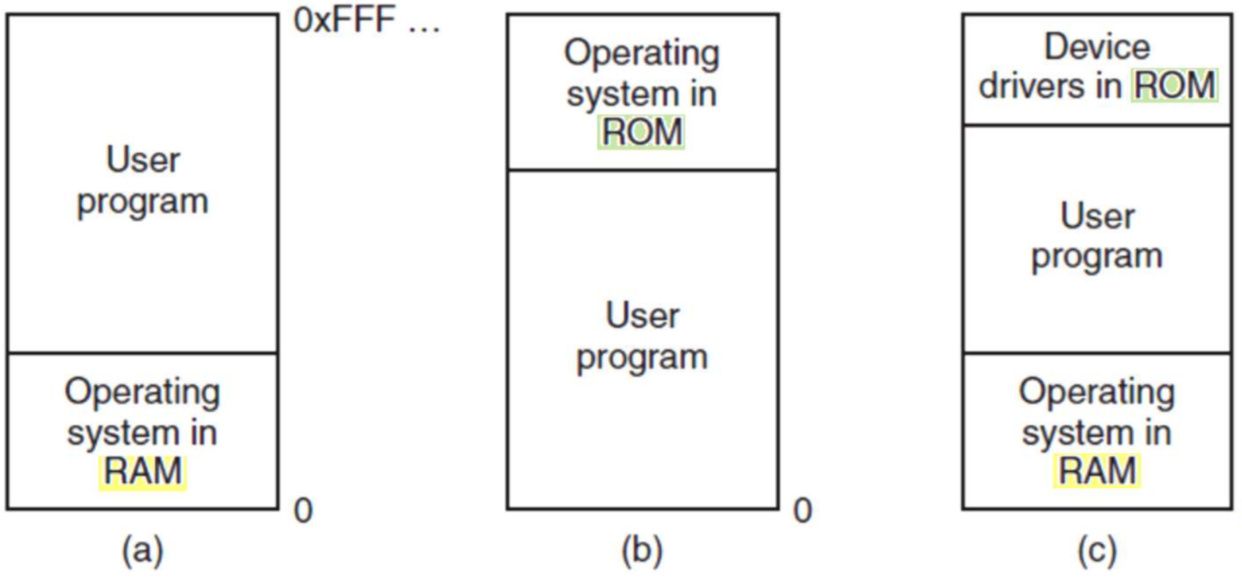
- However, those memory are expensive
  - Solution: memory hierarchy
  - OS: abstract the hierarchy into a useful model and manage the abstraction

No Memory Abstraction

- Every program sees the physical memory
  - `mov ax, 1000;    load ax with the contents of physical memory address 1000`

- Only one process might be running at a time

Three simple ways of organizing memory



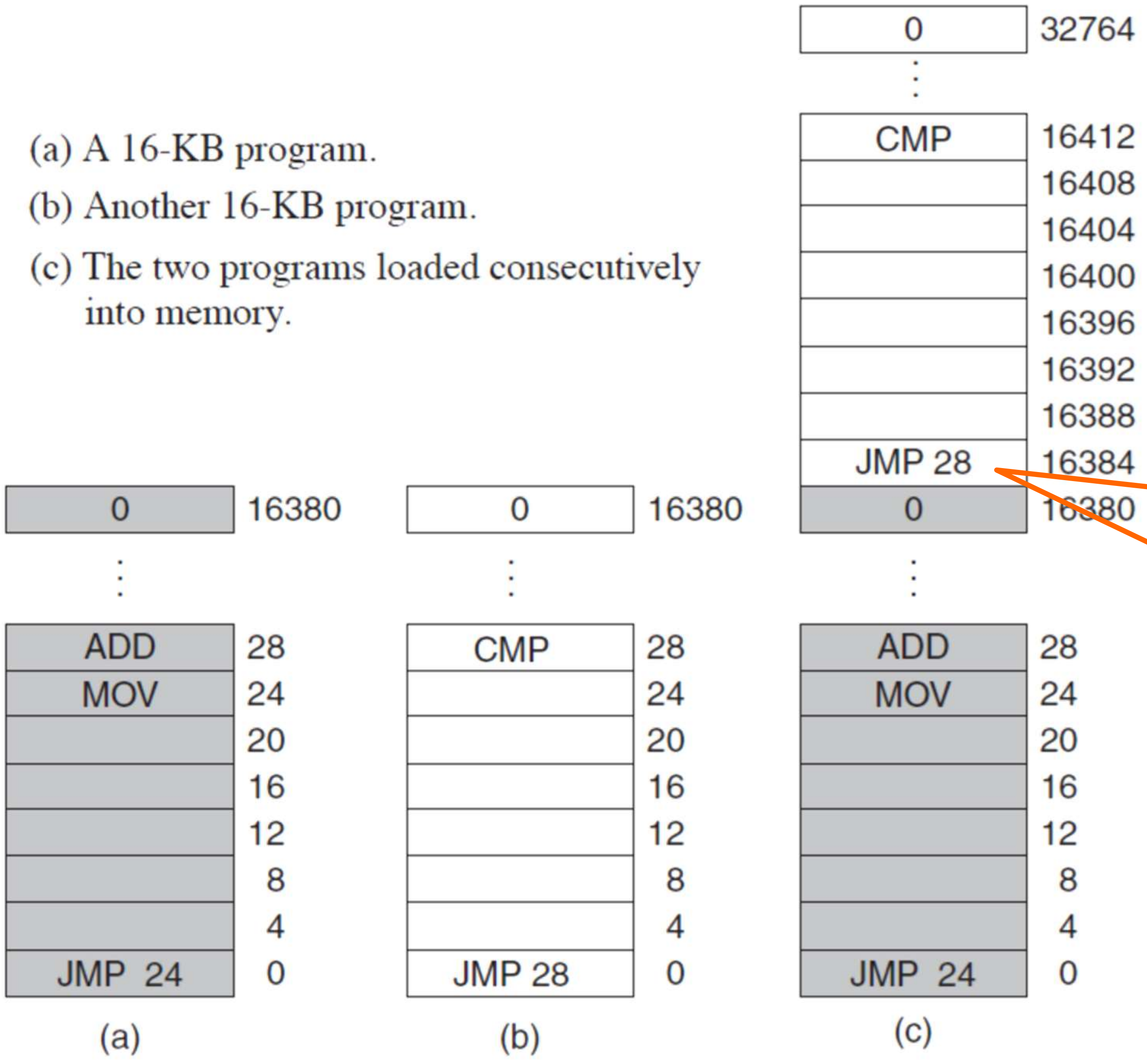
No Memory Abstraction:
Running multiple processes together

- Static relocation
  - Loading: add offset to all memory references

- Swapping
  - Save and load entire processes to/from disk
  - A way to run multiple processes

No Memory Abstraction:

Running multiple processes together



(a) A 16-KB program.

(b) Another 16-KB program.

(c) The two programs loaded consecutively into memory.

This code will not work

Static relocation: add 16384 to all addresses references

Memory Abstraction: Address Space

- Address space
  - The set of addresses that a process can use to address memory
    - E.g.: 000-0000 to 999-9999 for telephone numbers, 0.0.0.0 to 255.255.255.255 for IPV4 addresses, …
  - Issue: how to give each process its own address space

- Abstraction
  - Process: abstraction for CPU
  - Address space: abstraction for memory

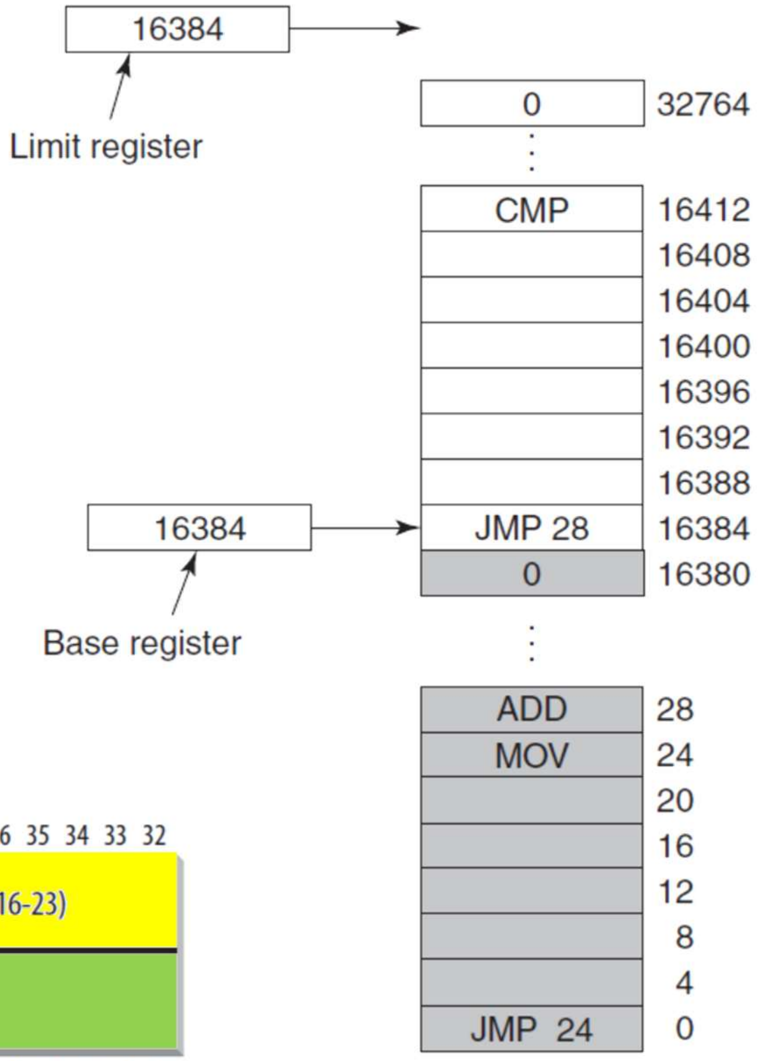
Memory Abstraction: Address Space

- Base and Limit registers
  - Base register: where a program is loaded
  - Limit register: length of the loaded program

- Whenever the CPU accesses the memory
  - The base register is added to the address
  - The address is checked with the limit register

Memory Abstraction: Address Space

- Base and Limit registers provide
  - Relocation
  - Protection



(c)

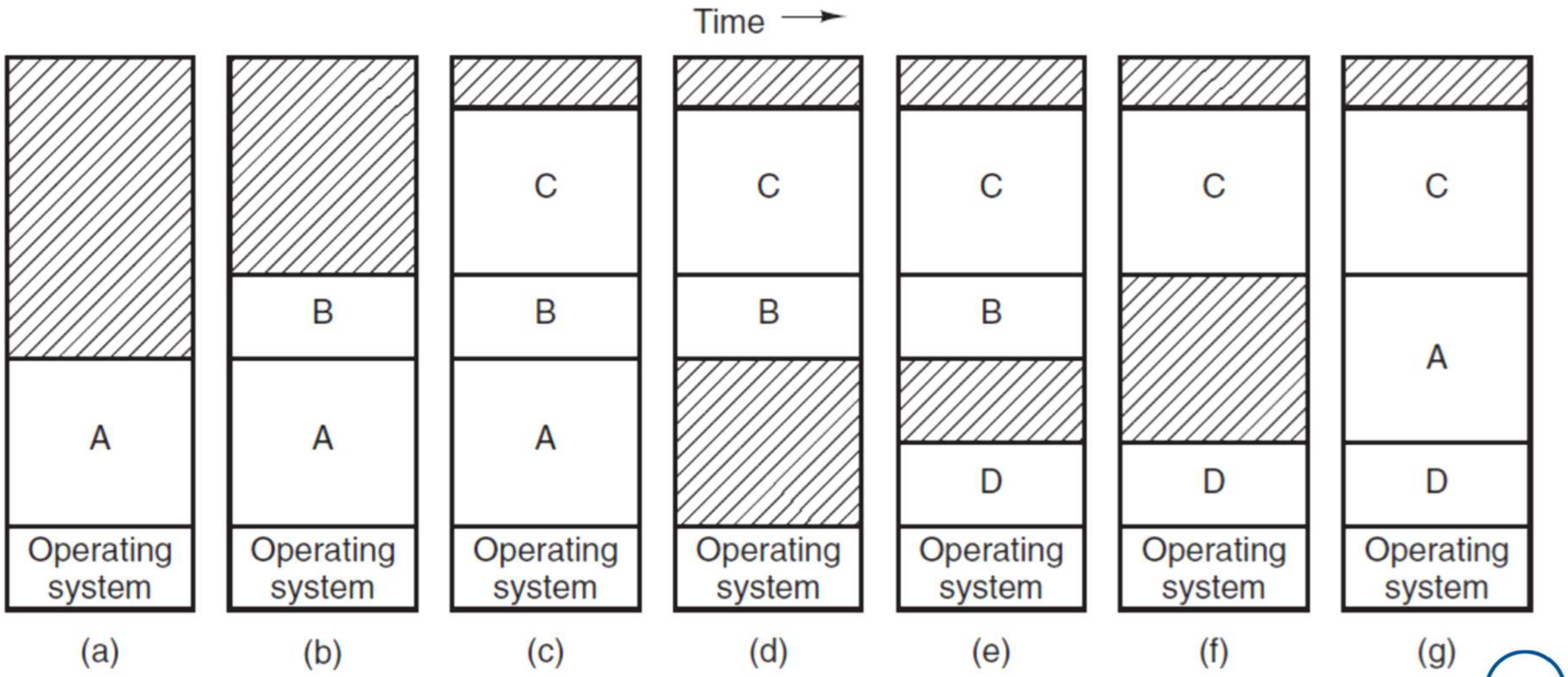
Code Segment Descriptor

63 62 61 60 59 58 57 56	55	54	53	52	51 50 49 48	47	46 45	44	43 42 41 40	39 38 37 36 35 34 33 32
BASE(24-31)	G	D	0	AVL	LIMIT (16-19)	1	DPL	S=1	TYPE	BASE (16-23)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
BASE(0-15)	LIMIT (0-15)

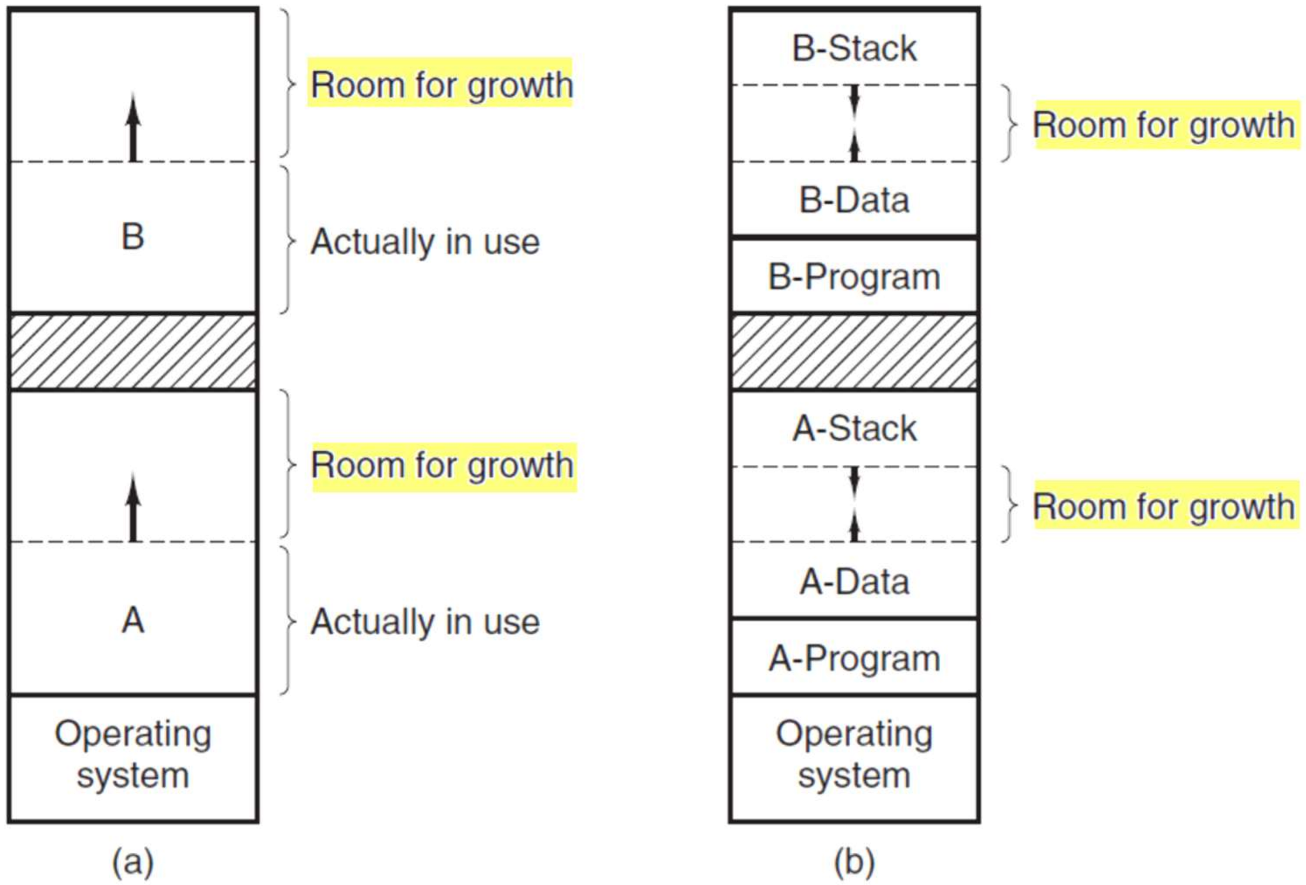
Memory Abstraction: Address Space

- Swapping
  - When there are more processes than the memory can hold
  - Loading a process in its entirety into memory, running it for a while, and store it into disk



Memory Abstraction: Address Space

- If a process's data area grows
  - E.g. Heap, Stack
  - Reserve extra memory when swap in/out



Hardware and Control Structures

- Two characteristics of paging and segmentation
  - All memory access within a process are logical addresses
  - A process may be broken up into pieces and they need not be contiguous in main memory

- It is not necessary that all of the pages or all of the segments of a process be in main memory

Hardware and Control Structures

- Partially loaded processes:
  to access instructions or data which are not in main memory
  - An interrupt occurs indicating a memory access fault
  - OS puts the interrupted process in a blocked state
  - OS issues a disk I/O
  - When the disk I/O is finished, an interrupt is issued
  - OS places the process back to the Ready state

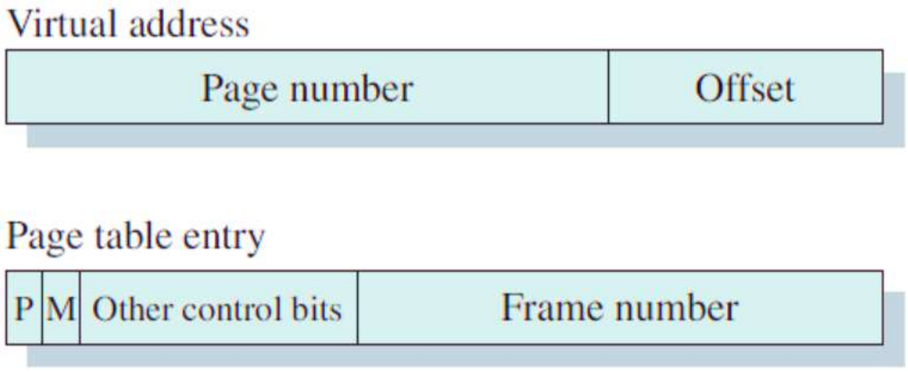
Hardware and Control Structures

- With partially loaded processes
  - More processes may be maintained in main memory
  - A process may be larger than all of main memory
  - Because of the locality, loading the entire process in main memory will be wasteful
  - Time will be saved during swap in and swap out

Virtual Memory by Paging

- Each process has a page table

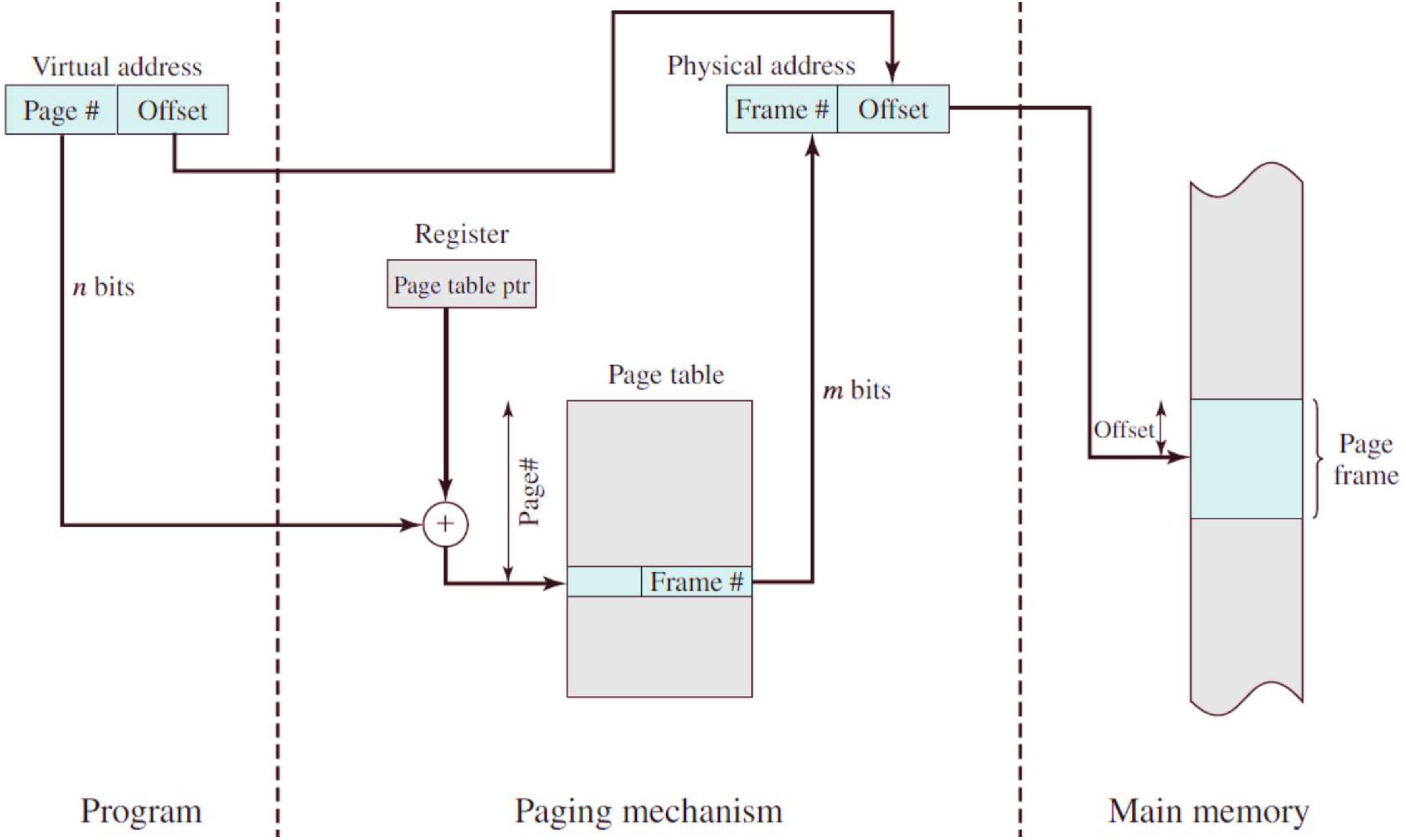
- Page table entry
  - Frame number of the page
  - Present bit (P): indicates whether the page is in main memory or not
  - Modify bit (M): indicates whether the page has been modified since it was loaded



Page Table Structure

- Address translation
  - Virtual address (page #, offset) → Physical address (frame #, offset)
  - A register (PTBR) points to the page table in memory
  - The page # is used as an index into the page table
  - The physical address comprises the frame # from the page table and the offset

Paging: Address Translation

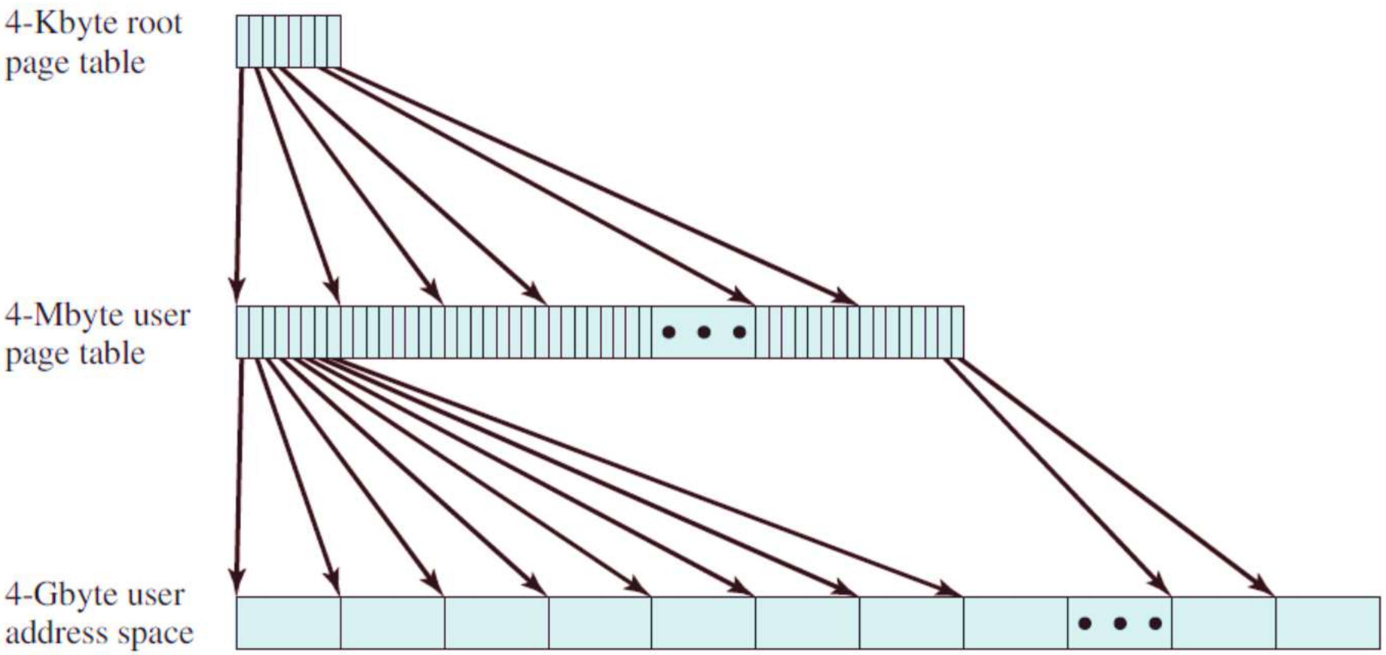


Multi-level Page Table

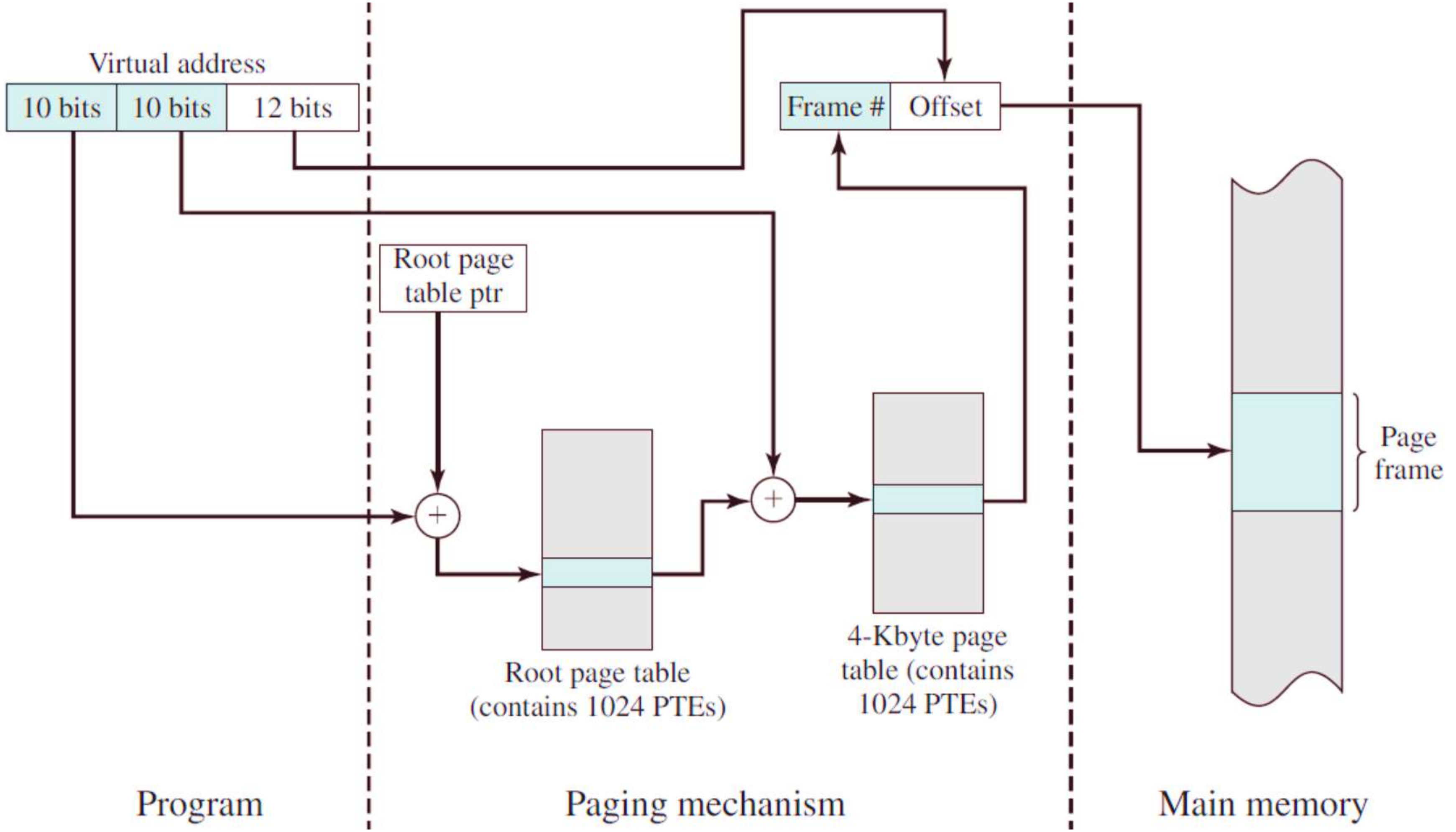
- Issue: page table size is proportional to that of the virtual memory
    - Solution: multi-level page table or inverted page table
- Example: page table size can be large
    - 2^{31} (2 GB) VM, 2^{9} (512 B) page size => 2^{22} page table entries
- Store page tables in virtual memory rather than in physical memory
    - Page the page table

Two-Level Page Table

- Example
  - 2^{32} (4 GB) VM, 2^{12} (4 KB) page size → 2^{20} pages
  - Assuming 4 byte for each PTE, page table size is 2^{22} (4 MB) that will be stored in 2^{10} pages
  - The page table can be mapped by a root page table of 2^{12} (4KB) in main memory



Two-Level Page Table: Address Translation

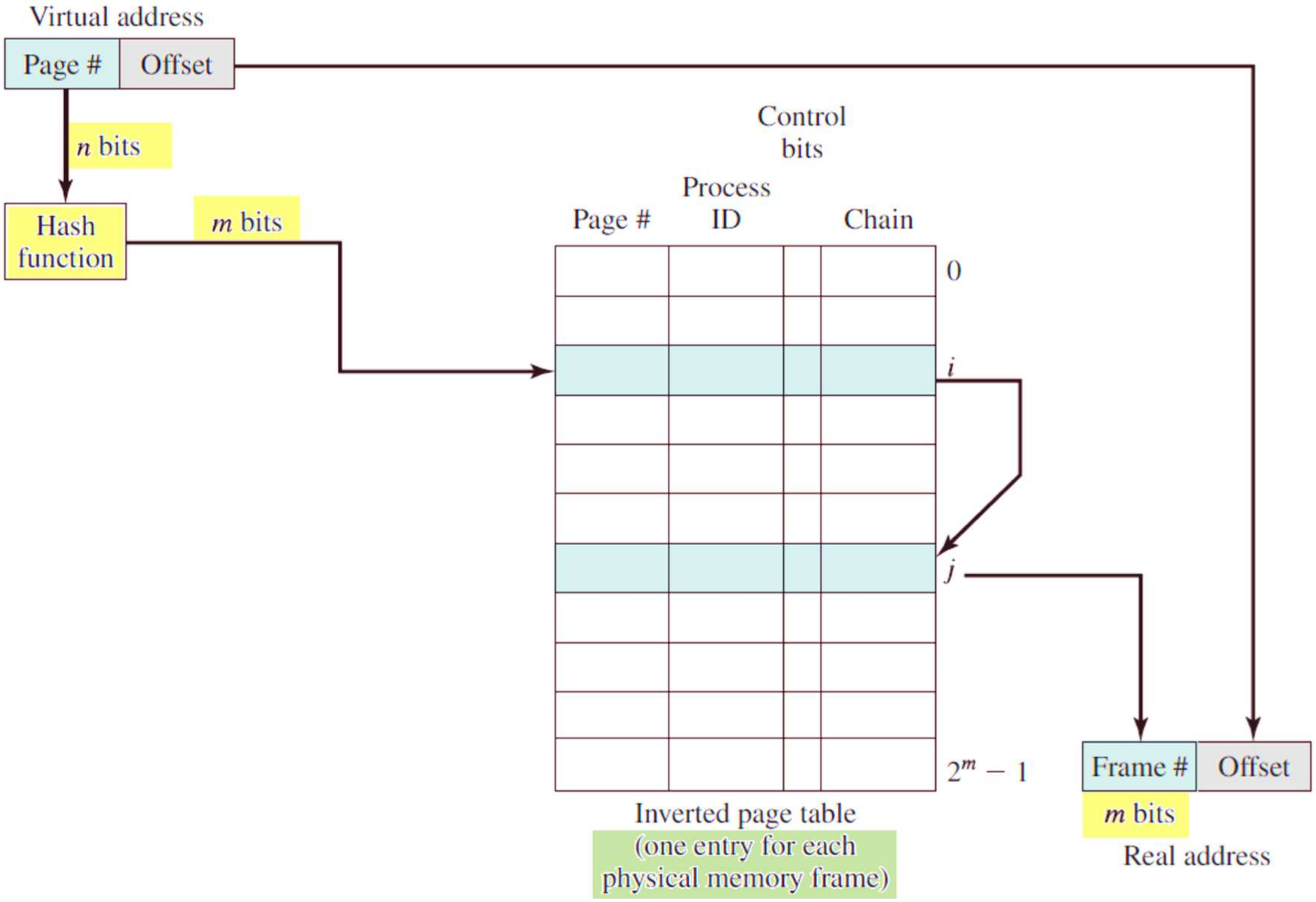


Inverted Page Table

- Inverted page table (not per process, but global)
  - Indexed by frame number (not by page number)
  - Table is searched (using hash) for the entries having the page number and the process id

- Page table entries
  - Page number: page # of virtual memory
  - Process identifier: page # alone is not unique
  - Chain pointer: to resolve the hash collision issue
  - Control bits: valid, referenced, modified, protection…

Inverted Page Table

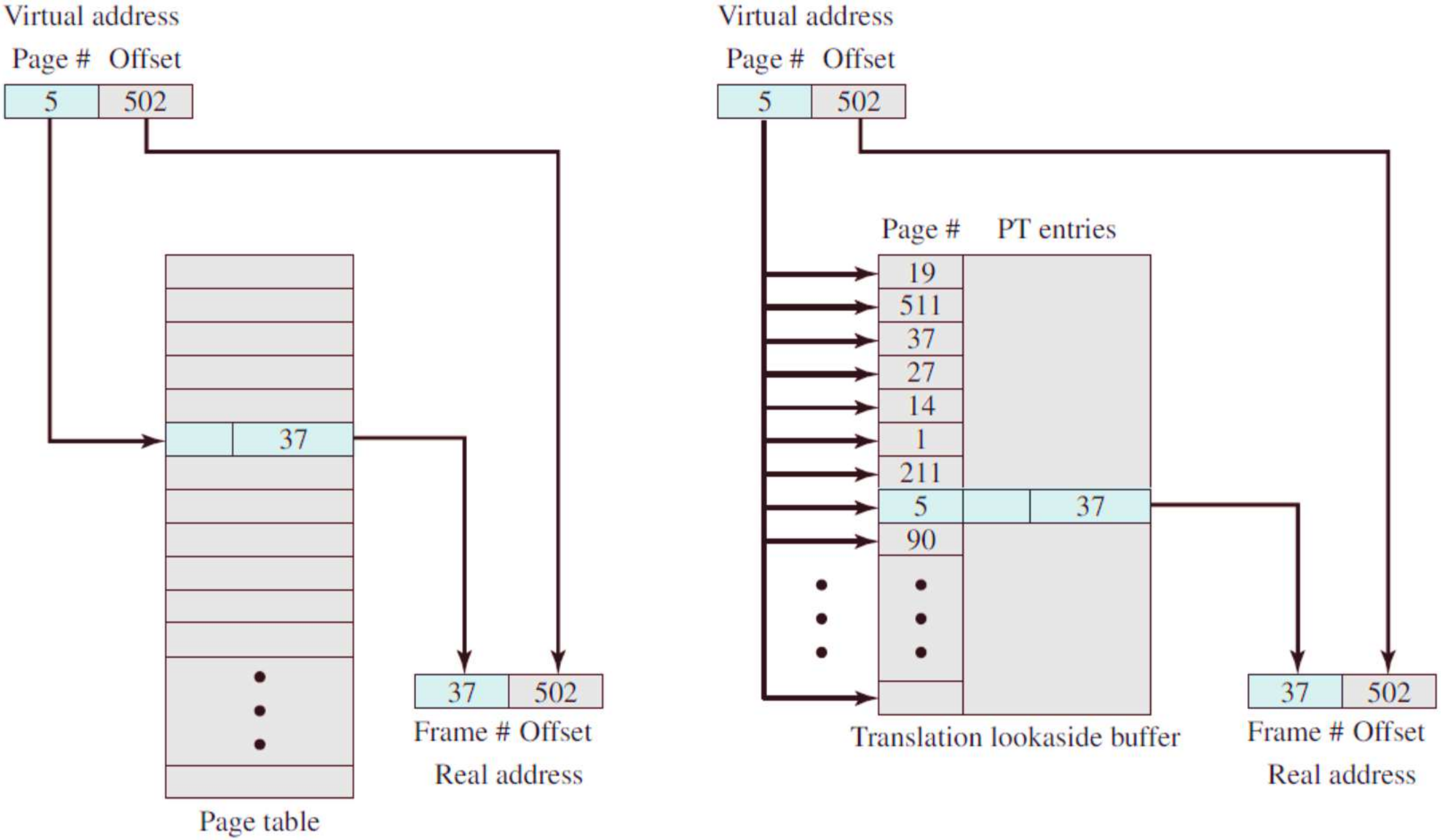


Translation Lookaside Buffer (TLB)

- Every virtual memory reference can cause two physical memory accesses
  - To fetch the page table entry
  - To fetch the desired data

- Translation Lookaside Buffer (TLB)
  - Cache for page table entries
  - Associative mapping using a page number

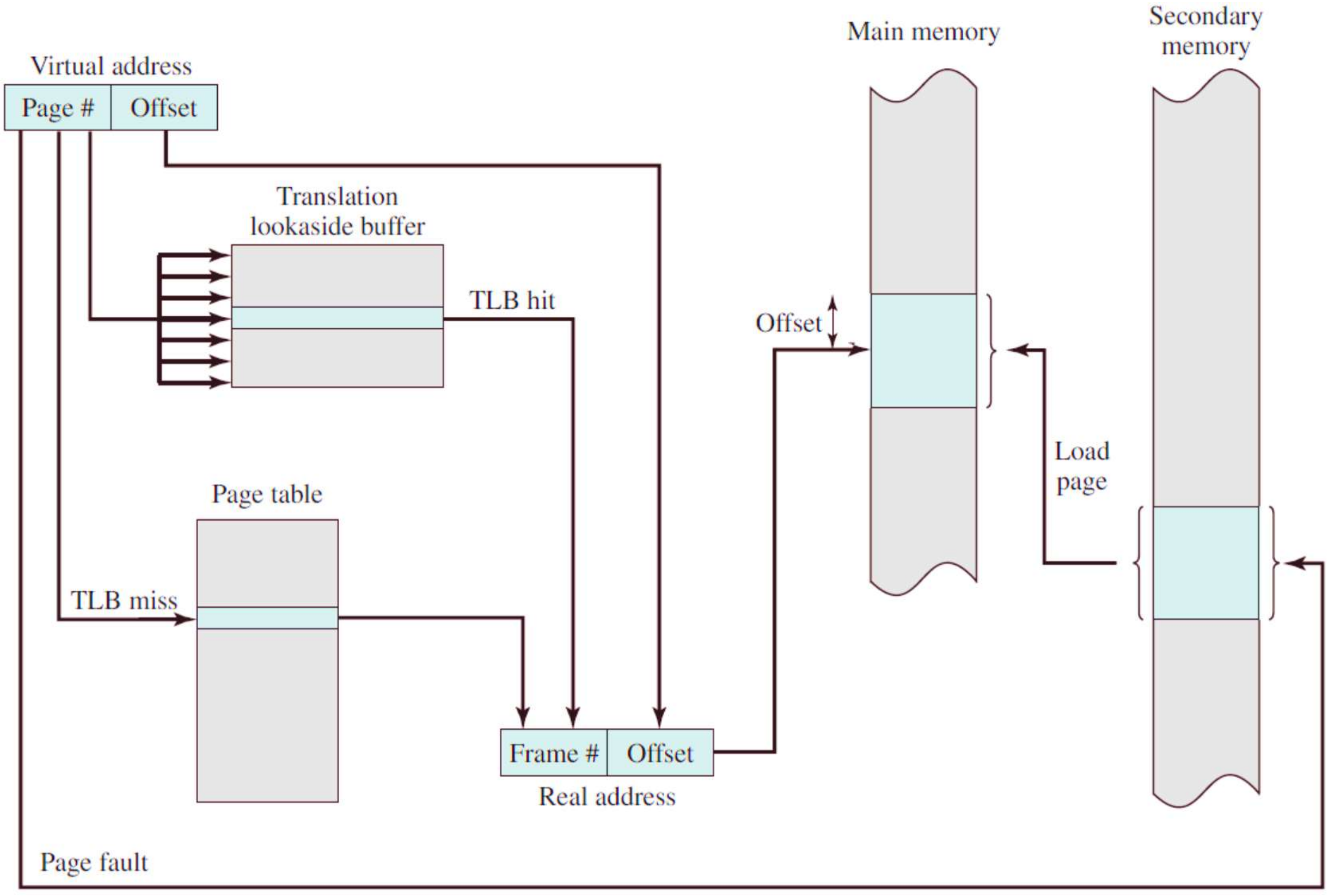
TLB: Associative Mapping

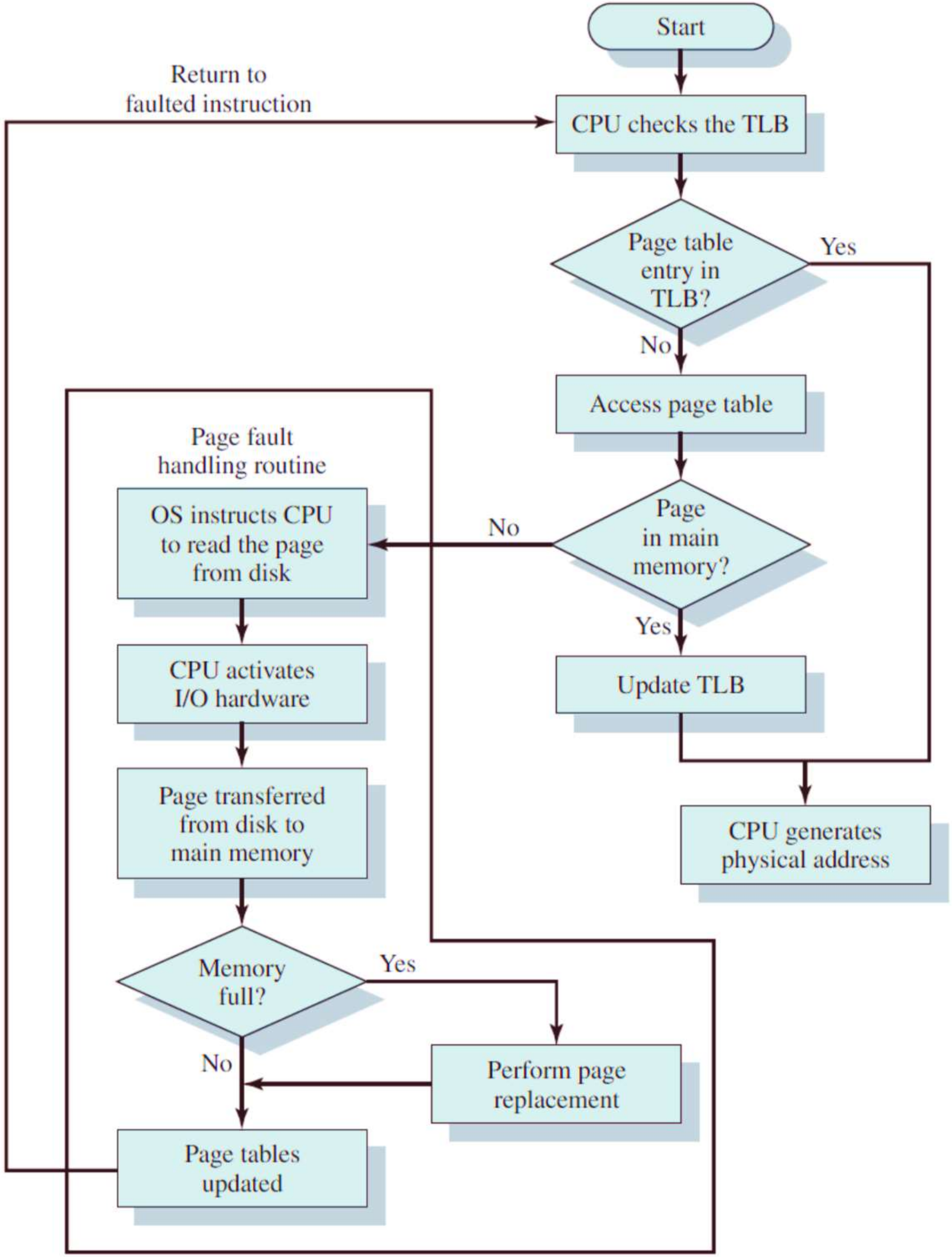


(a) Direct mapping

(b) Associative mapping

Translation Lookaside Buffer (TLB)





Start

CPU checks the TLB

Page table entry in TLB?

Yes

No

Access page table

Page in main memory?

No

Yes

Update TLB

CPU generates physical address

Page fault handling routine

OS instructs CPU to read the page from disk

CPU activates I/O hardware

Page transferred from disk to main memory

Memory full?

Yes

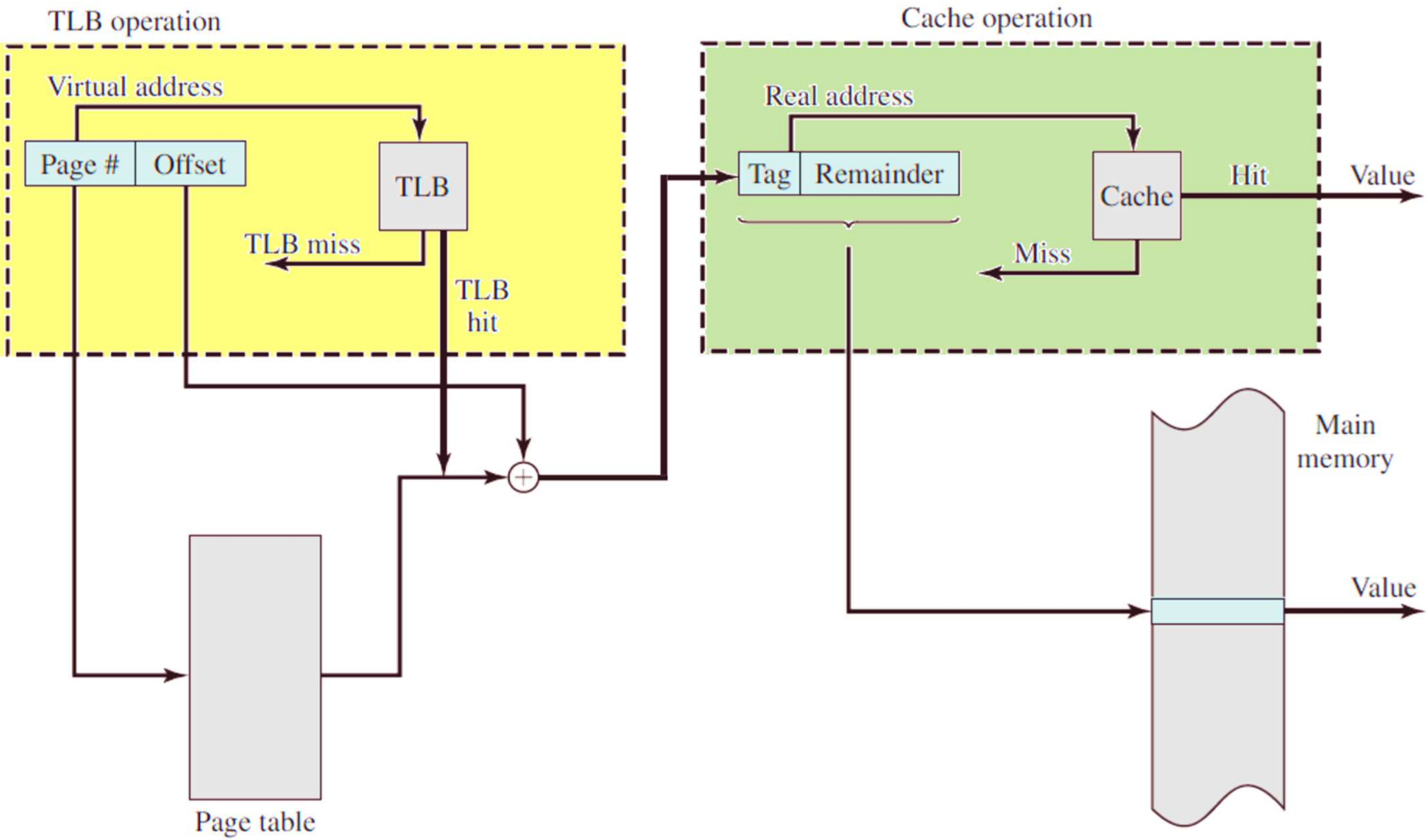
No

Perform page replacement

Page tables updated

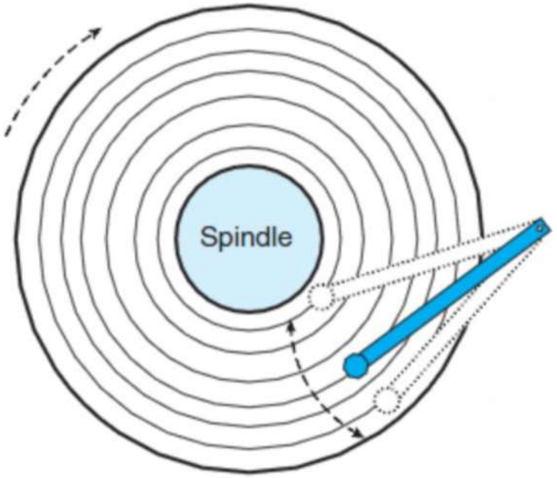
Return to faulted instruction

TLB and Cache

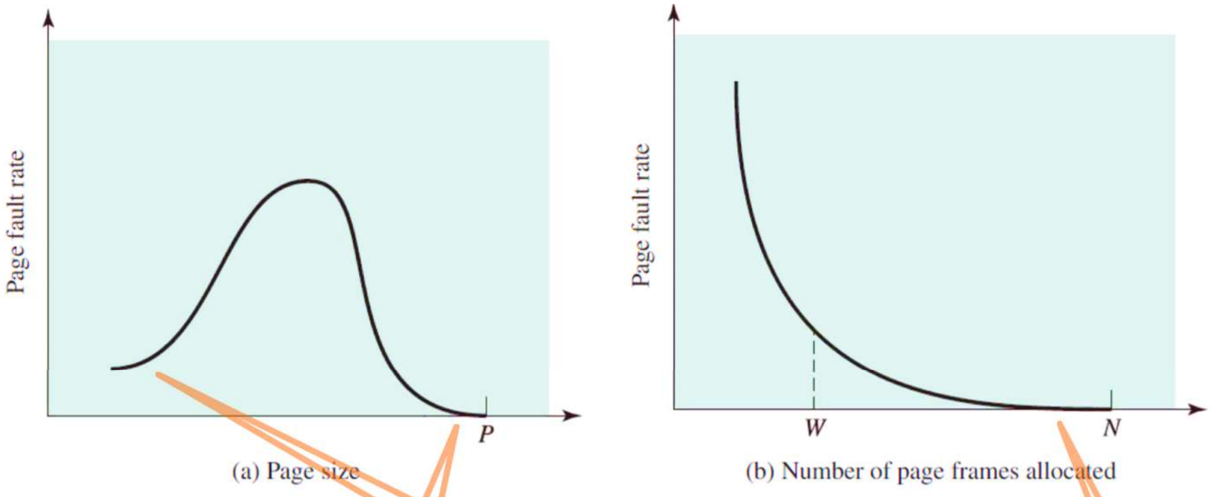


Page Size

- Factors to consider
  - Internal fragmentation:
    - The larger the page, the more the amount of internal fragmentation
  - Secondary memory:
    - With seek time, rotational delay, disks favor a larger page size
  - Page fault rate



Page Size



(a) Page size

(b) Number of page frames allocated

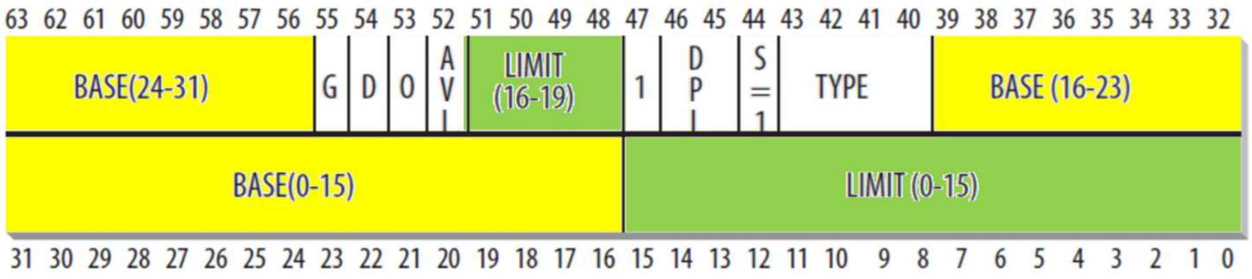
P = size of entire process W = working set size N = total number of pages in process

- Page fault rate
  - Figure (a): smaller pages have better locality
  - Figure (a): a page large enough to hold the entire process causes no page fault
  - Figure (b): given a fixed page size, page fault rate decreases with the number of pages allocated to a process

Virtual Memory by Segmentation

- Segments
  - Visible to programmers
  - Dynamic size
  - Virtual address (segment #, offset)

Code Segment Descriptor



63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32

BASE(24-31) | G | D | 0 | AVL | LIMIT (16-19) | 1 | DPL | S = 1 | TYPE | BASE (16-23)

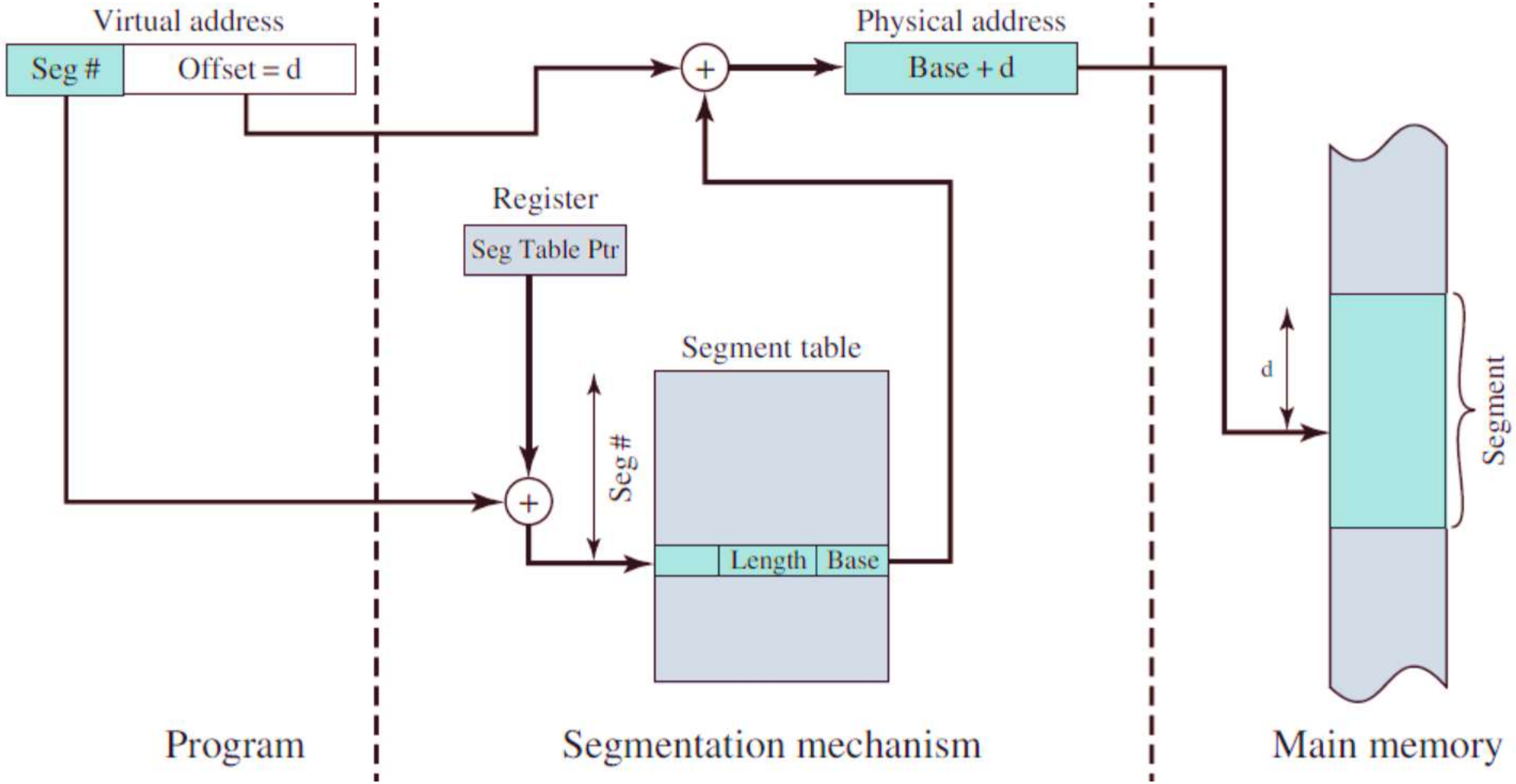
BASE(0-15) | LIMIT (0-15)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

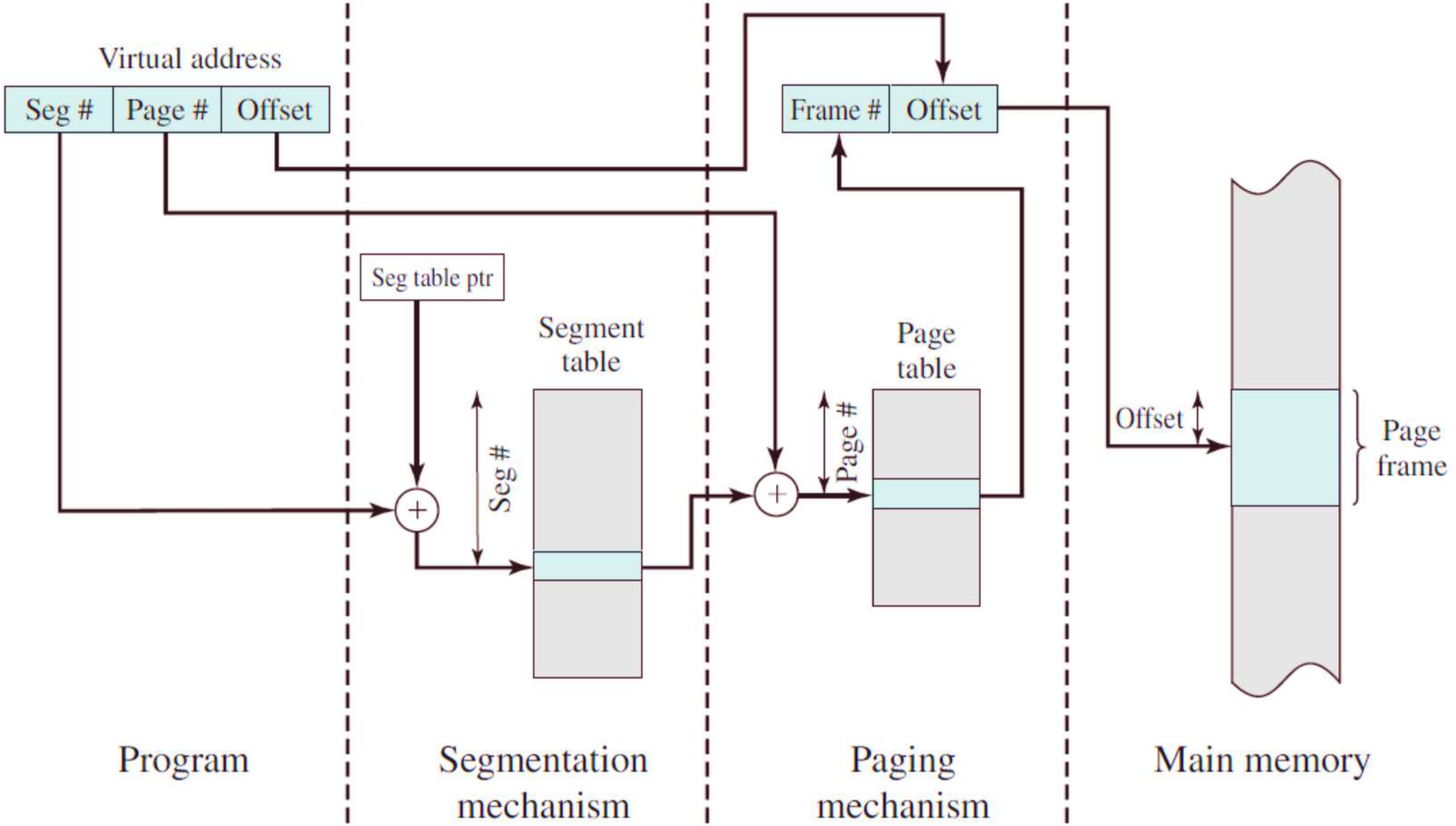
Segmentation

- Advantages
  - Simplifies the handling of growing data structure
  - Allows programs to be altered and recompiled without requiring entire set of programs to be relinked and reloaded
  - Provides a sharing mechanism among processes
  - Provides a protection mechanism

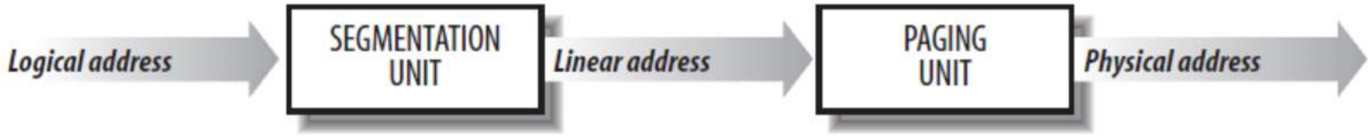
Segmentation: Address Translation

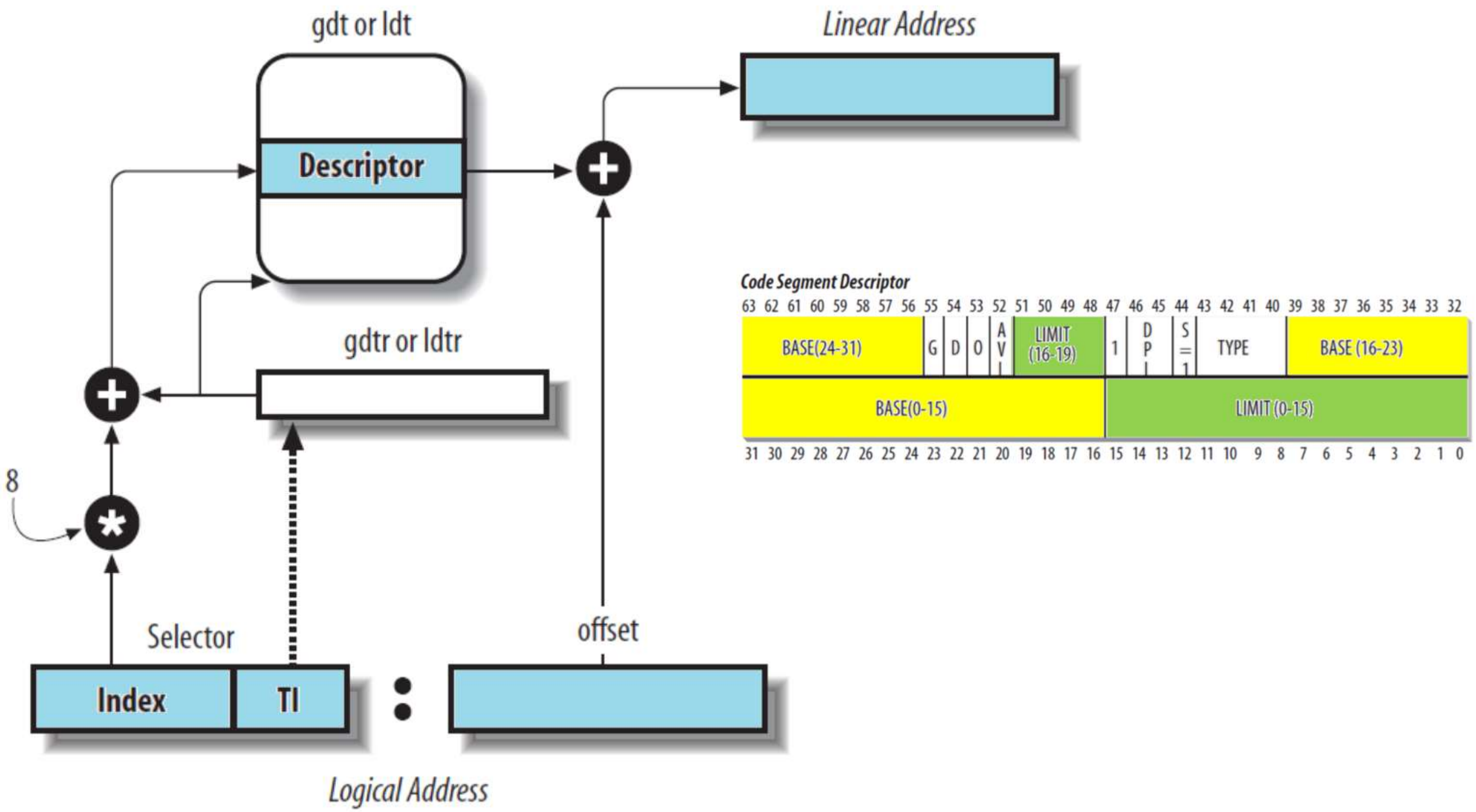


Combined Paging and Segmentation

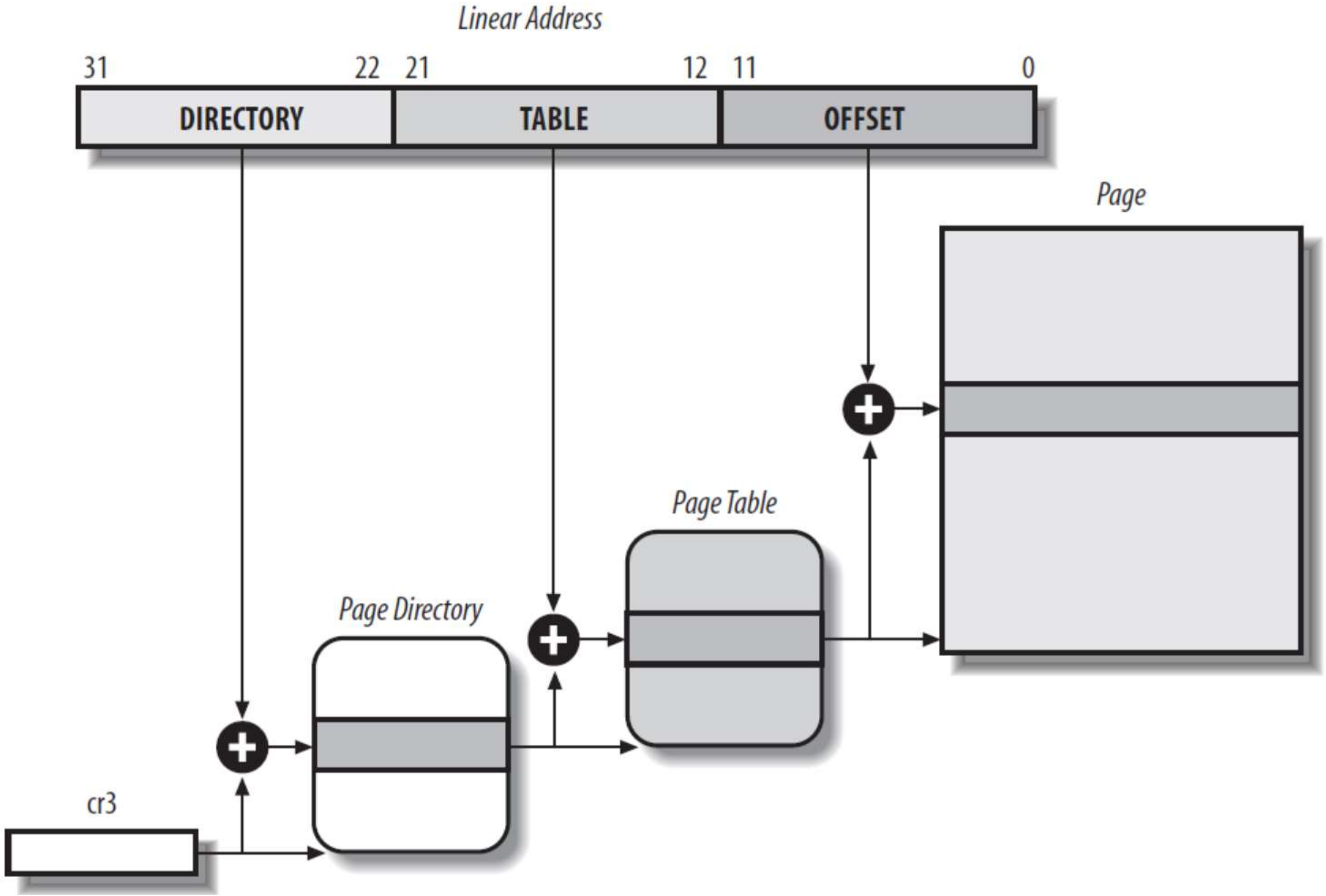


Address Translation in x86





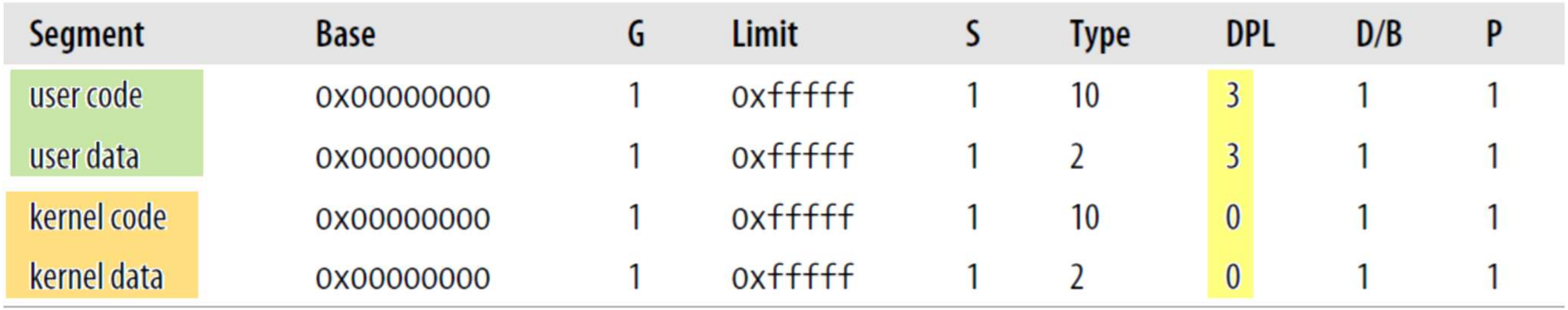
Address Translation in x86



Segmentation in Linux

- All processes running in User mode
  - Use the same pair of segments for instructions and data
  - User code segment, user data segment

- All processes running in Kernel mode
  - Use the same pair of segments for instructions and data
  - Kernel code segment, kernel data segment



Segment	Base	G	Limit	S	Type	DPL	D/B	P
user code	0x00000000	1	0xfffff	1	10	3	1	1
user data	0x00000000	1	0xfffff	1	2	3	1	1
kernel code	0x00000000	1	0xfffff	1	10	0	1	1
kernel data	0x00000000	1	0xfffff	1	2	0	1	1