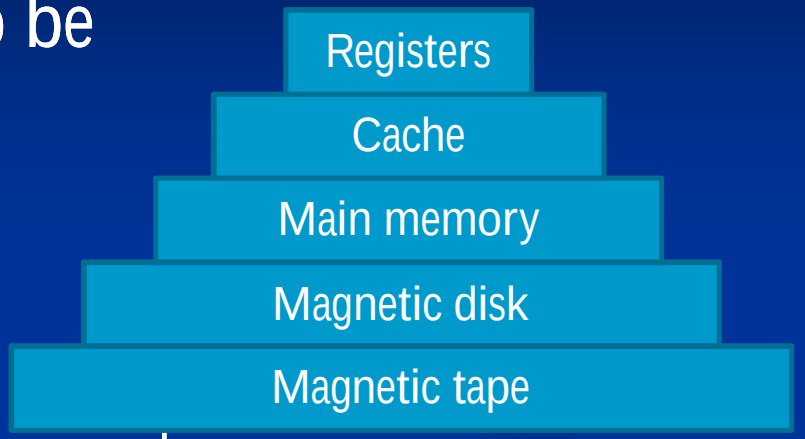
Operating Systems: Memory management

Fall 2008

Tiina Niklander

Memory Management

- Programmer wants memory to be
  - Indefinitely large
  - Indefinitely fast
  - Non volatile
- Memory hierarchy
  - small amount of fast, expensive memory – cache
  - some medium-speed, medium price main memory
  - gigabytes of slow, cheap disk storage
- Memory manager handles the memory hierarchy
- Requirements for memory management
  - Logical and physical organization
  - Protection and sharing

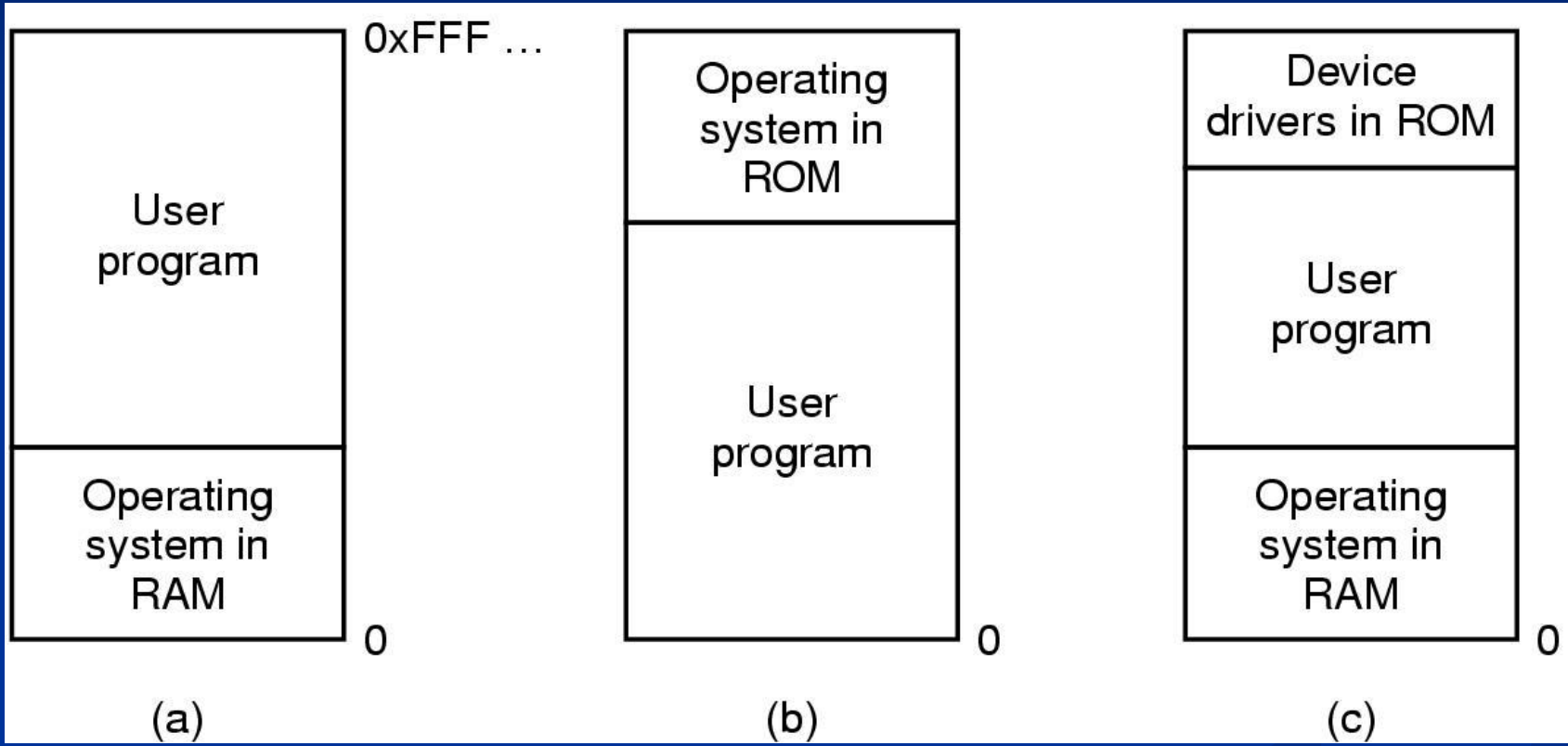


Memory management

- Programs use logical addresses of their own address space (0..MAX)
- OS kernel usually in a fixed location using physical memory addresses directly
- Rest of the physical memory for user processes and other OS parts
- OS task: memory allocation and process relocation
- Hardware task: address translation (to protect memory)
  - MMU – memory management unit

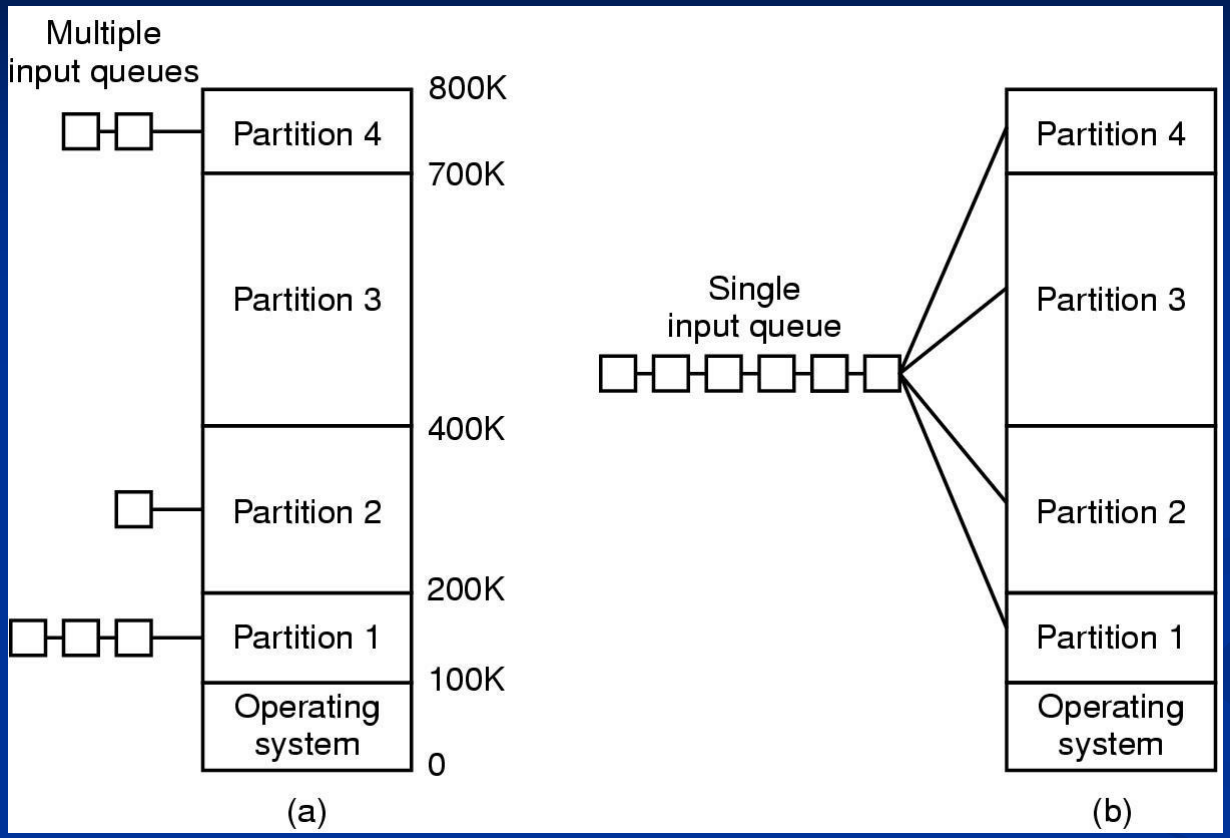
Basic Memory Management: One program

Monoprogramming without Swapping or Paging



No memory abstraction, no address space,
just an operating system with one user process

Multiprogramming with Fixed Partitions



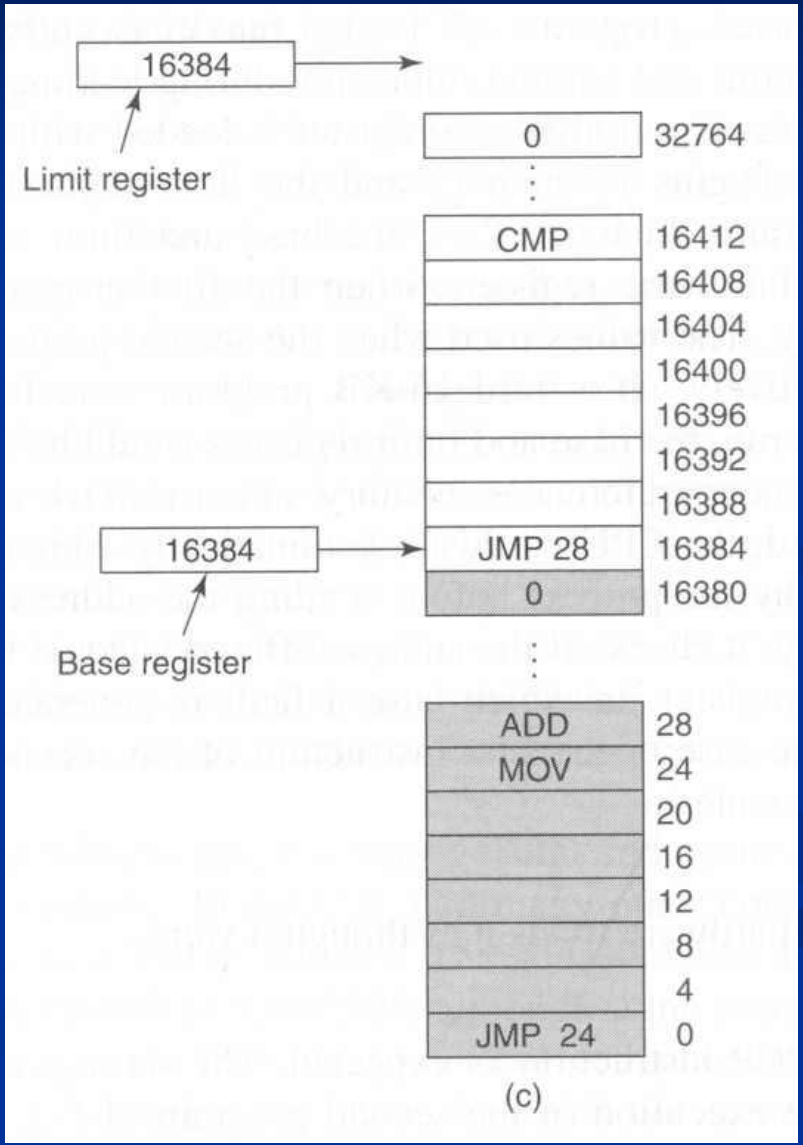
- OS places one process in one partition
  - Internal fragmentation (whole partition allocated to a smaller process)

Fixed partitions

- Process queue for partition on disk
  - In shared queue for all partitions or
  - In multiple queues for different sizes,
- No free partition, OS can swap
  - Move one process to disk
  - PCB always in memory
- Program too large for any partition; programmer has to design solution
  - Overlaying: keep just part of the program in memory
  - Write the program to control the part swapping between memory and disk

Relocation and Protection

- Cannot be sure where program will be loaded in memory
  - address locations of variables, code routines cannot be absolute
  - must keep a program out of other processes' partitions
- Use base and limit values
  - address locations added to base value to map to physical addr
  - address locations larger than limit value is an error
- Address translation by MMU

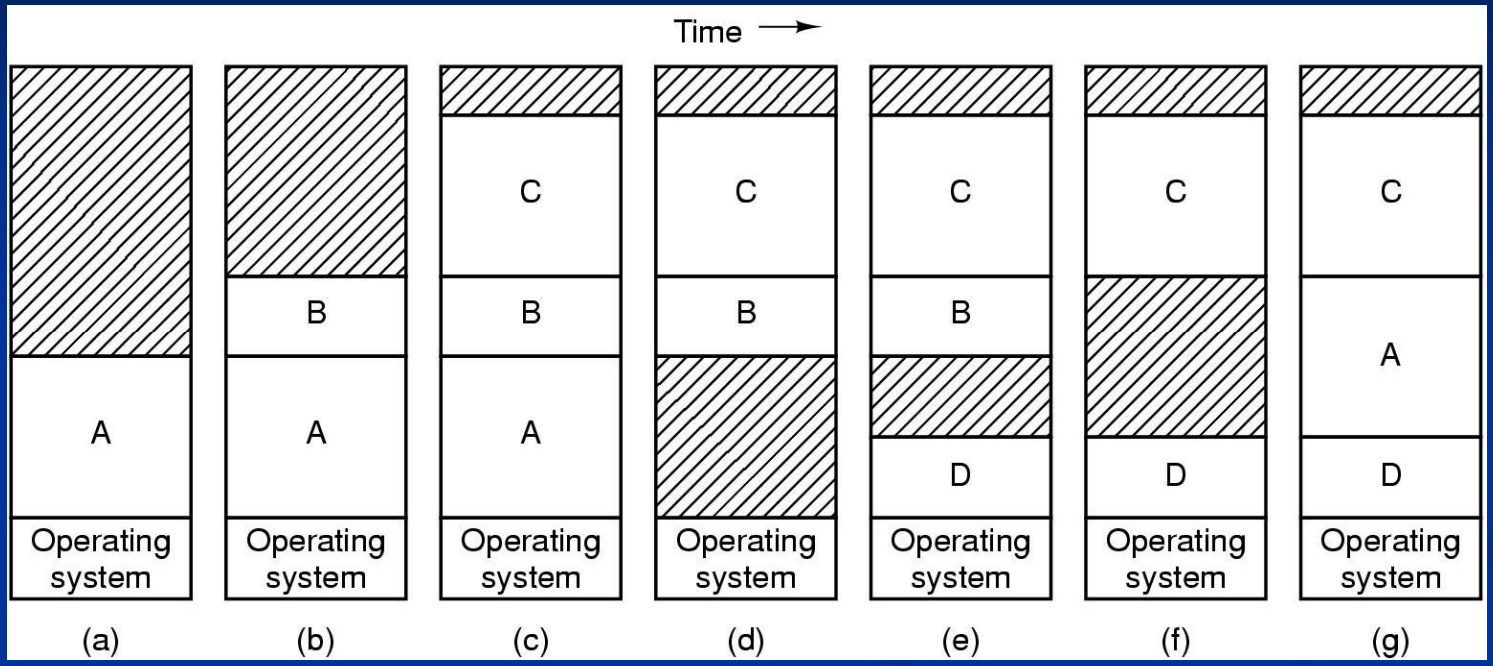


Sharing

- Access to shared code / data
  - No violation of the protections!
- Shared code
  - must be reentrant, not to change during execution
  - Just one copy of the shared code (e.g. library)
- Shared data
  - Processes co-operate and share data structures
  - E.g. shared buffer of producer and consumer

- Solution: system calls between processes, threads within a process (in virtual memory alt. solution)

Swapping (1)

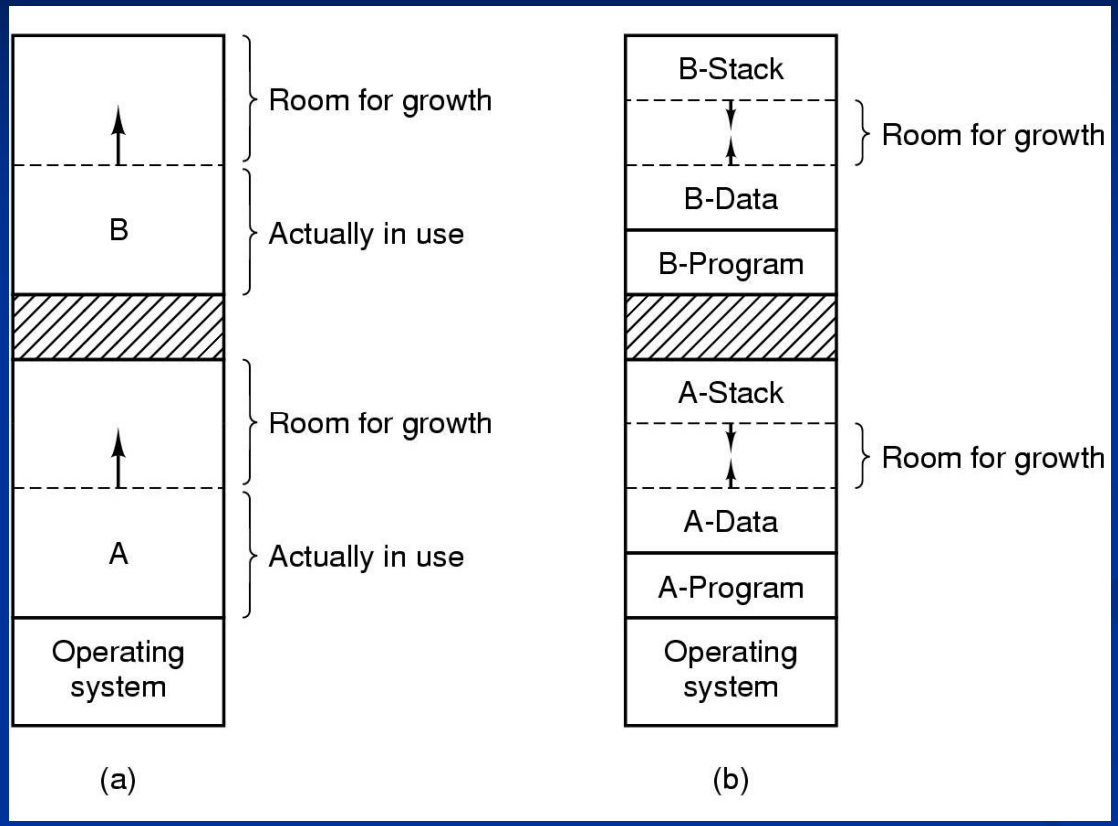


Memory allocation changes as

- processes come into memory
- leave memory

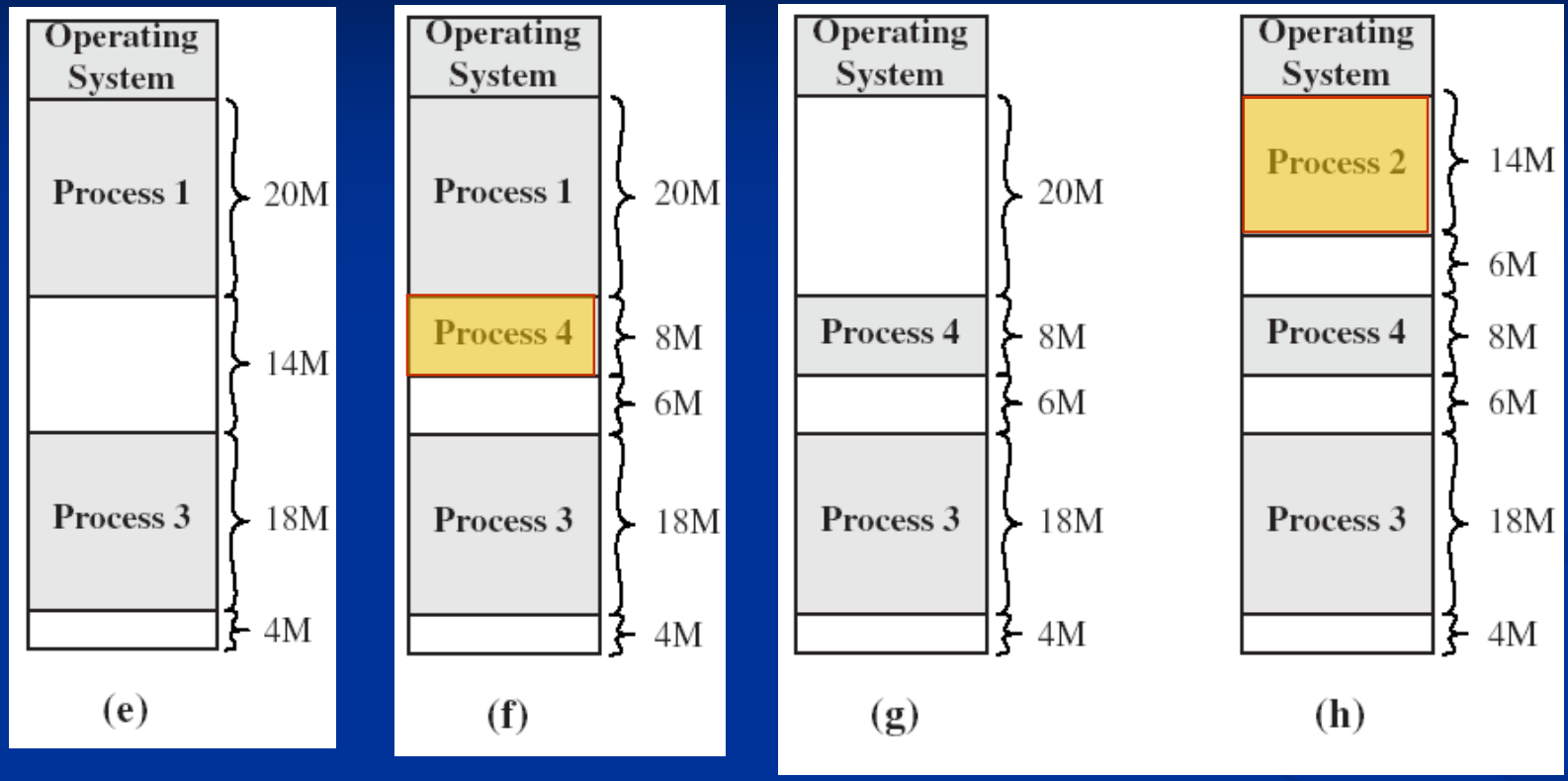
Shaded regions are unused memory

Swapping (2)



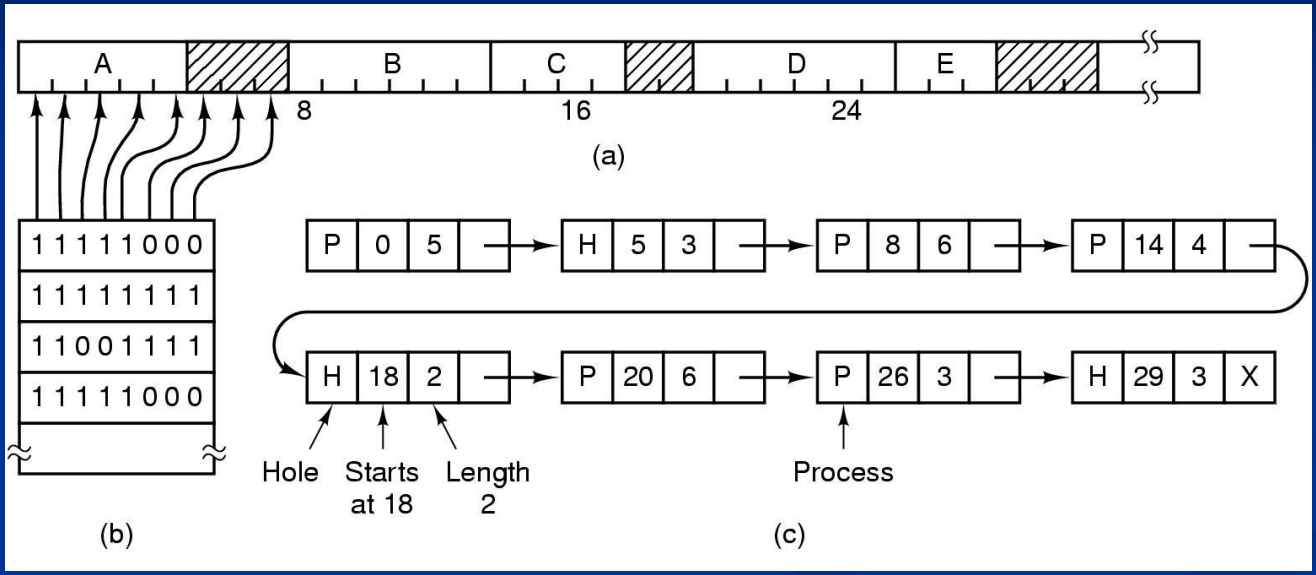
- (a) Allocating space for growing data segment
- (b) Allocating space for growing stack & data segment

Dynamic partitions



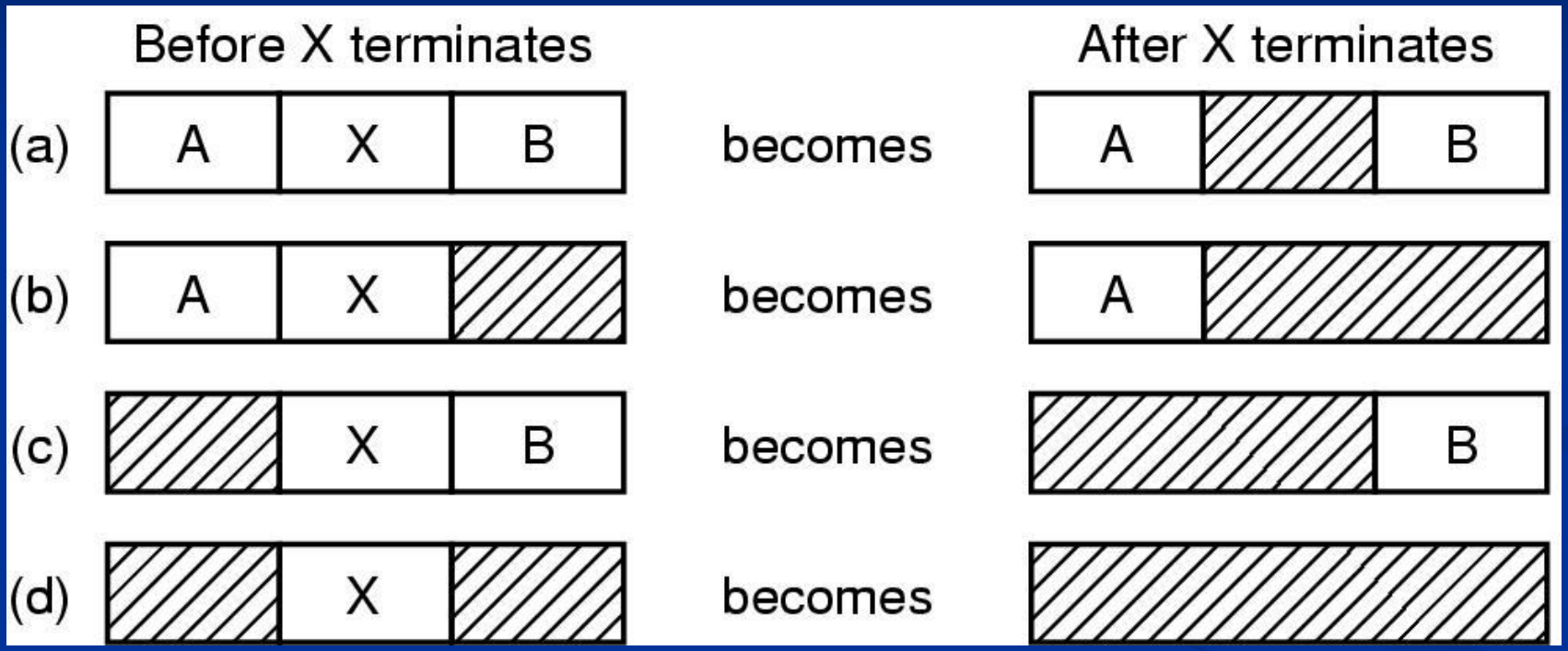
- No fixed predetermined partition sizes
- External fragments: 6M + 6M + 4M = 14M
- OS must occasionally reorganize the memory **(compaction)**

Memory Management: bookkeeping allocations and free areas



- Part of memory with 5 processes, 3 holes
  - tick marks show allocation units
  - shaded regions are free
- (b) Corresponding bit map
- (c) Same information as a list

Combining freed areas

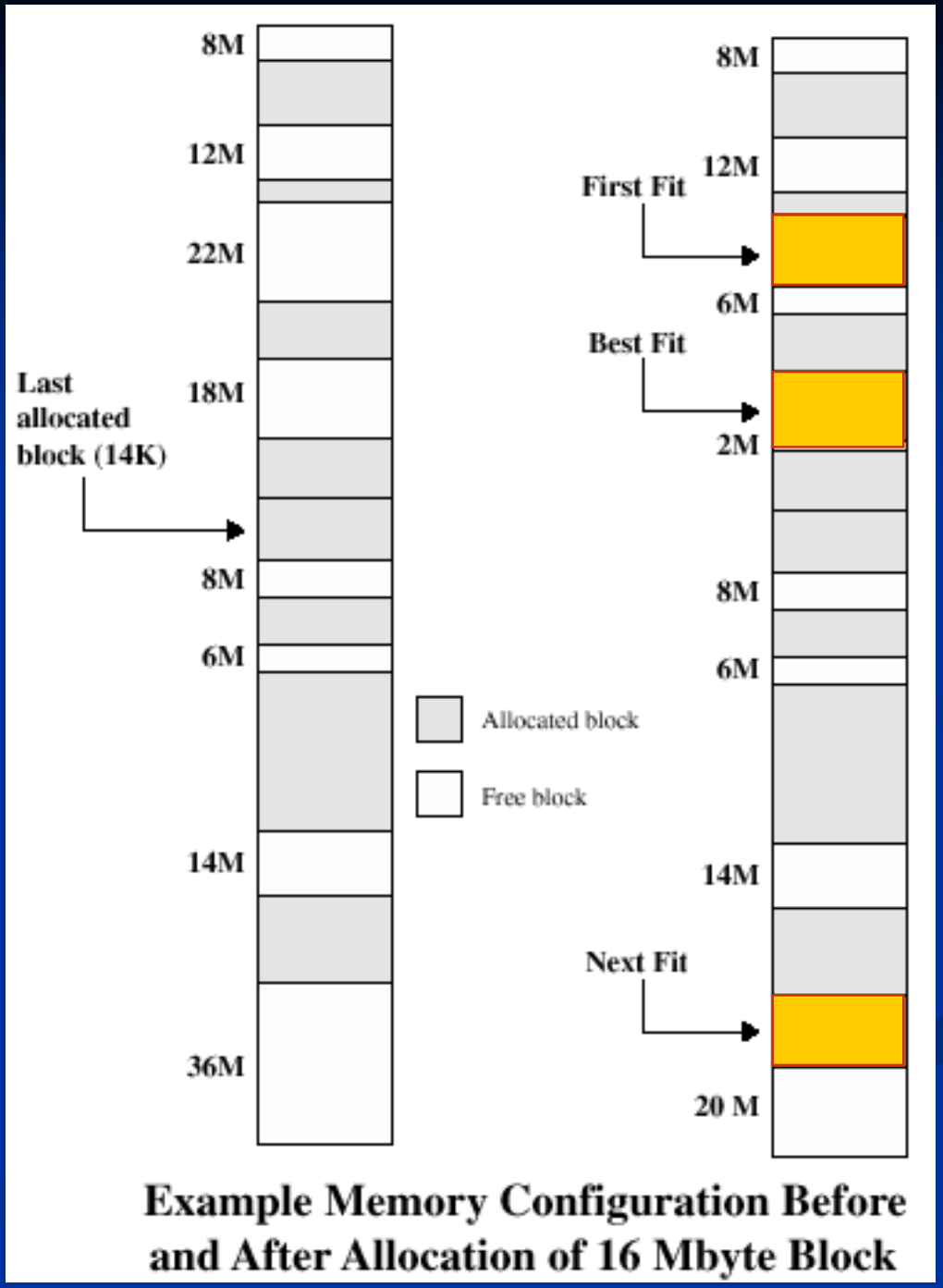


Four neighbor combinations for the terminating process X

Allocation

Where to place the new process?

- Goal: avoid external fragmentation and compaction
- Some alternatives:
- Best-fit
- First-fit
- Next-fit
- Worst-fit
- Quick-fit

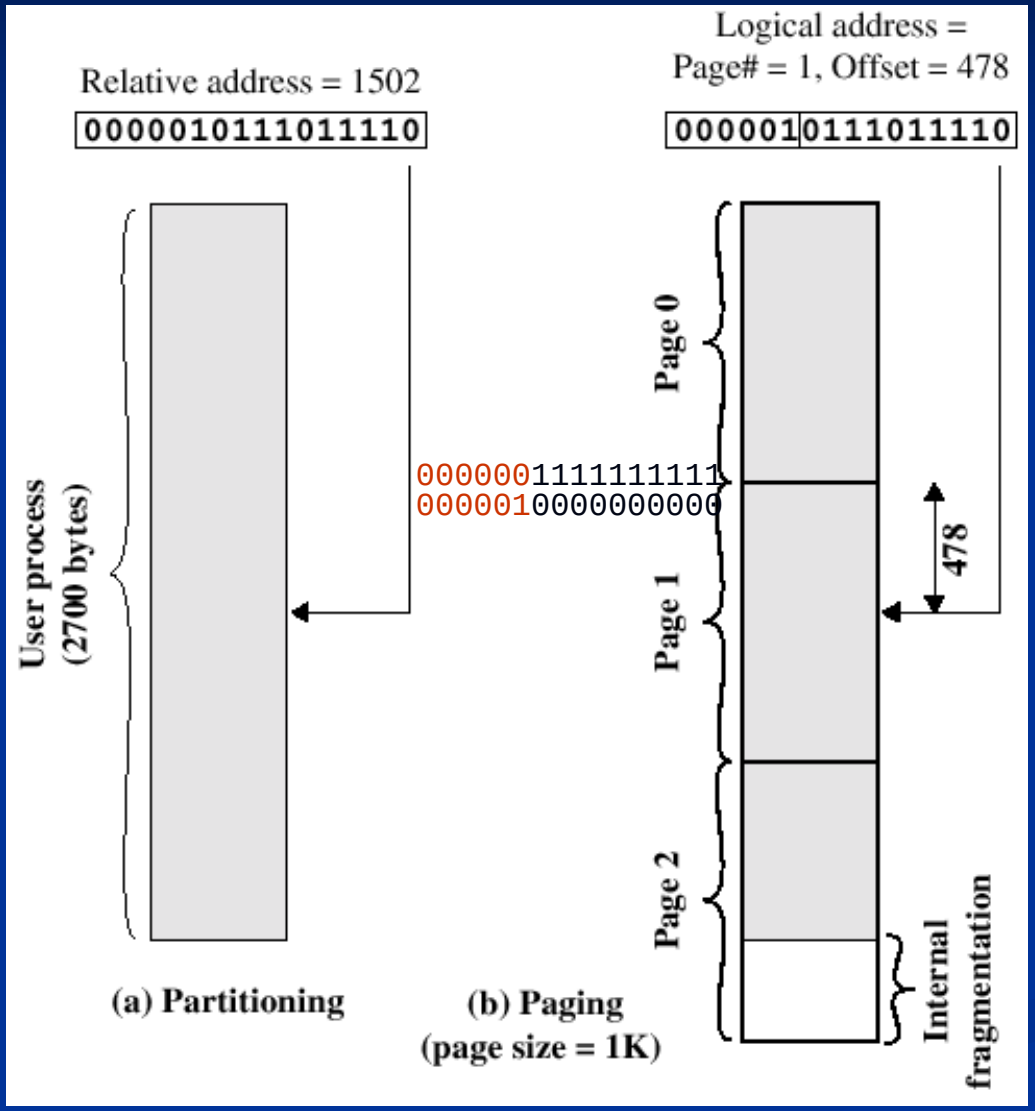


Example Memory Configuration Before and After Allocation of 16 Mbyte Block

Sta Fig 7.5

Virtual memory (using paging)

Paging

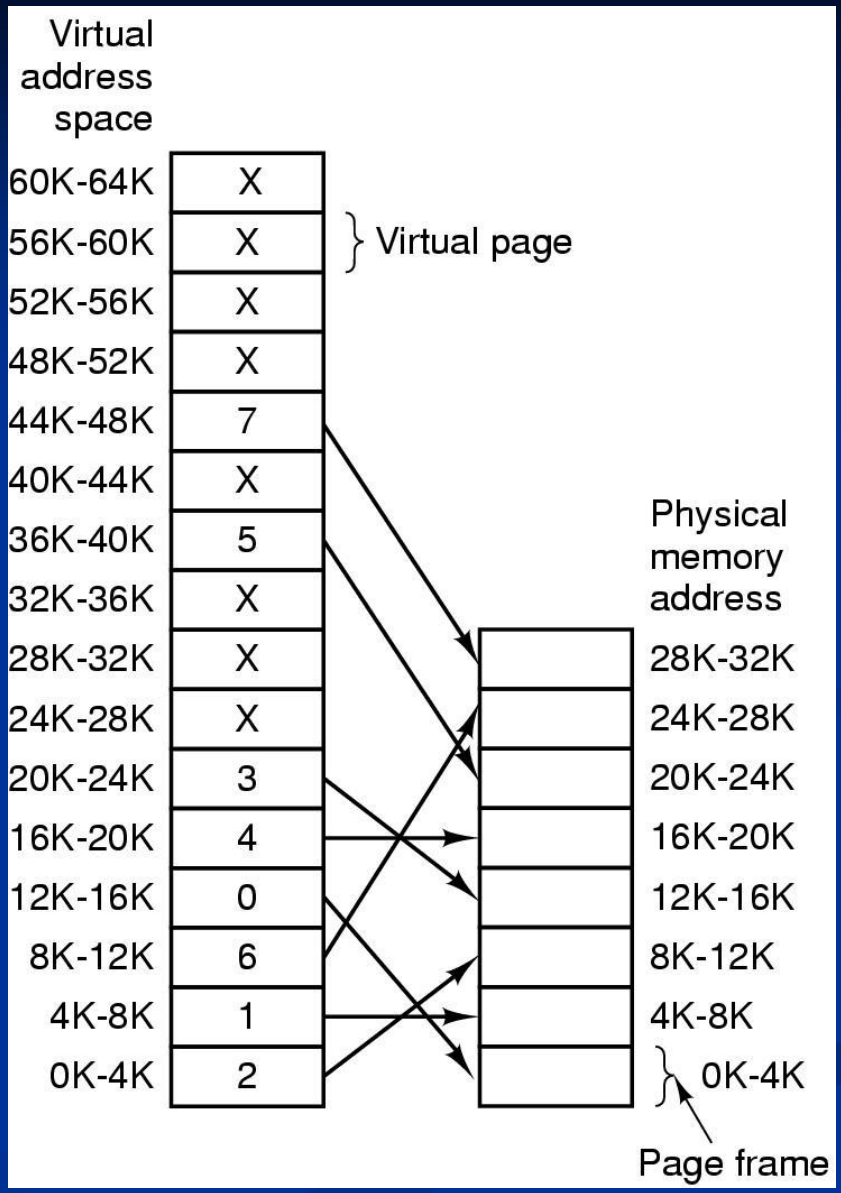


- OS: the program split to pages
  - Page location stored in page table
- Process location
  - Logical address always the same
  - MMU translates logical address to physical address using page table
  - Each page relocated separately

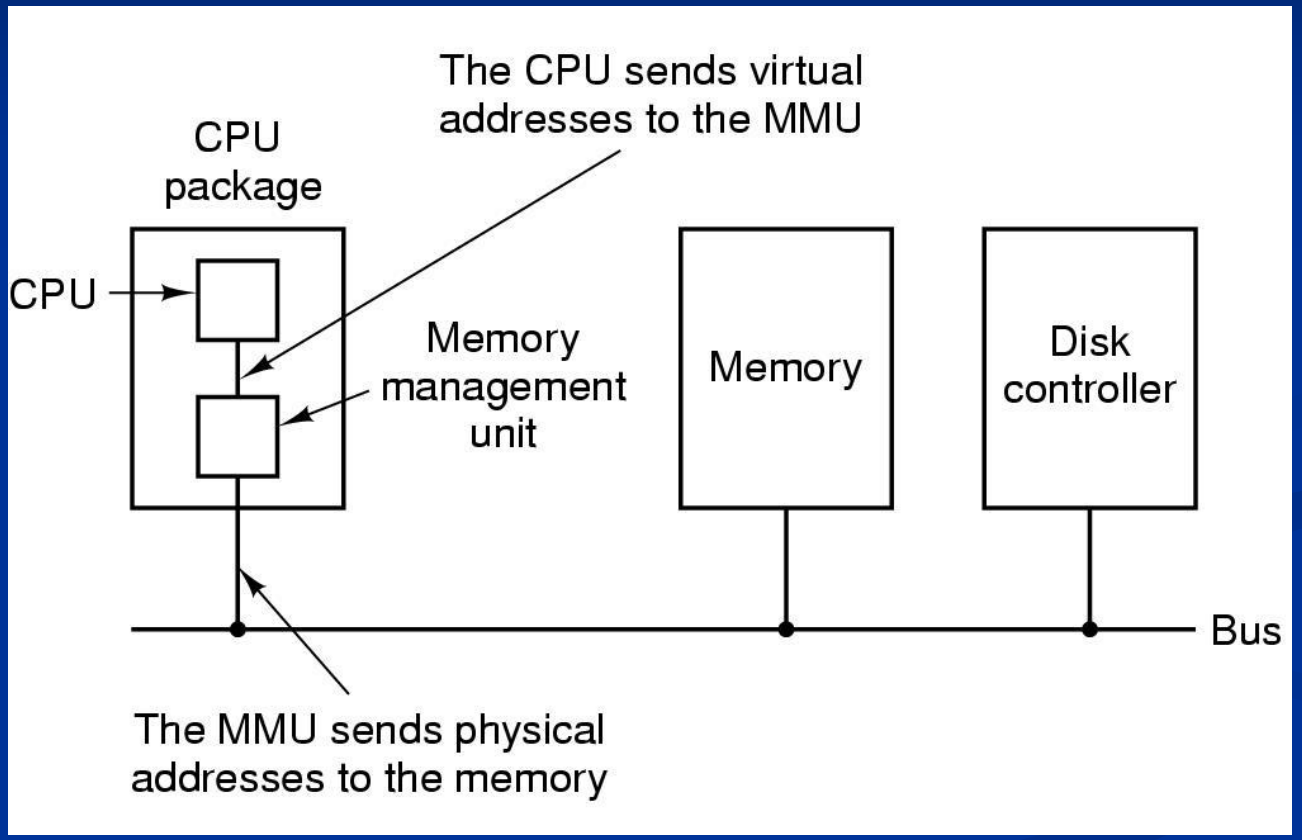
Sta Fig 7.11

Paging

- Each process has own page table
  - Contains the locations (frame numbers) of allocated frames
  - Page table location stored in PCB, copied to PTR for execution
- OS maintains a table (or list) of page frames, to know which are unallocated

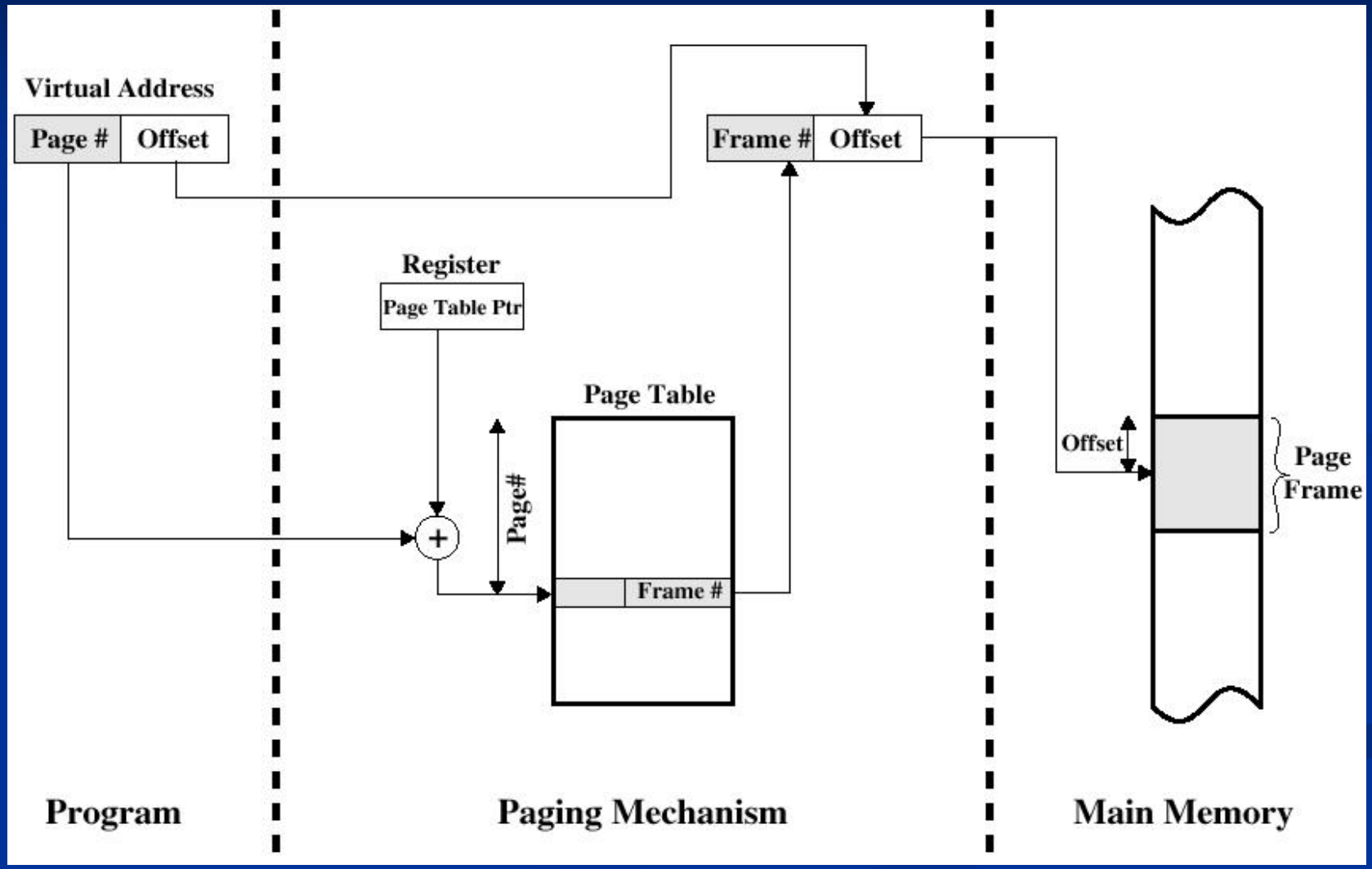


Paging: Address Translation



MMU has one special register, Page Table Register (PTR), for address translation

Paging: Address Translation

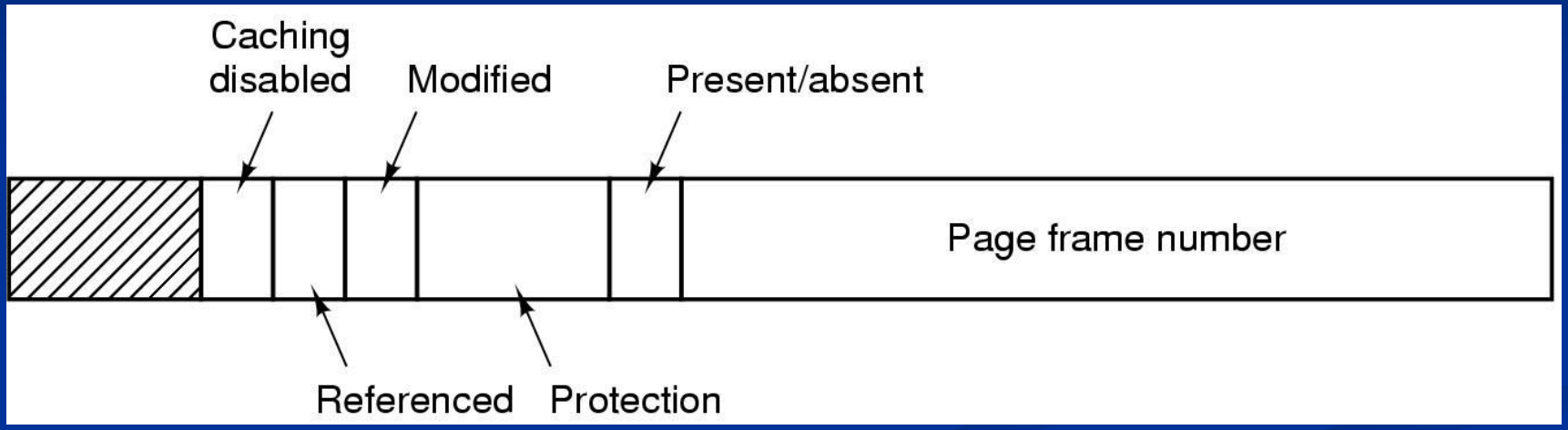


Sta 8.3

Page table

Page table

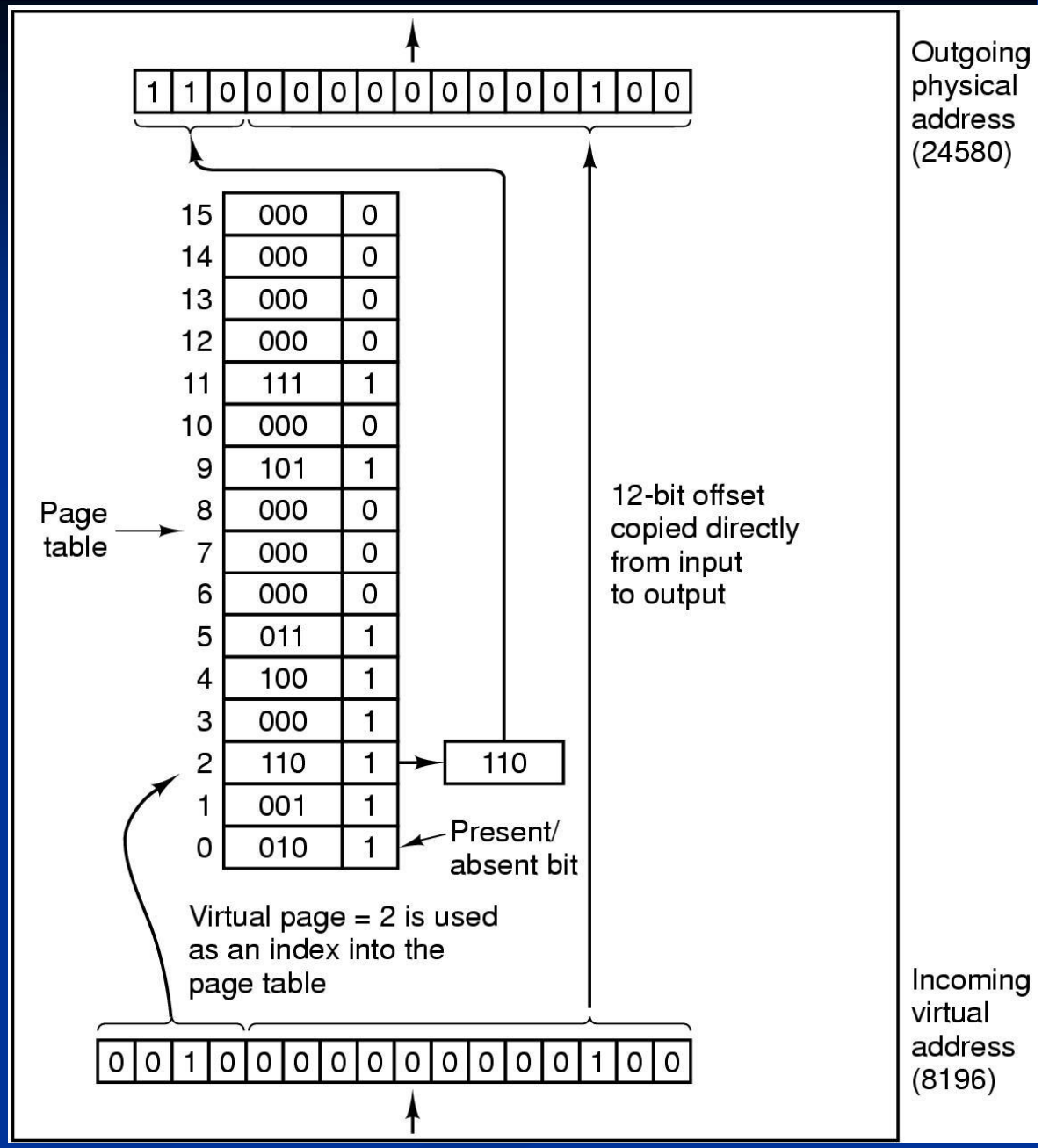
Typical Page Table Entry, Fig. 3.11



- Each process has its own page table
- Each entry has a present bit, since not all pages need to be in the memory all the time -> page faults
- Remember the locality principle
- Logical address space can be much larger than the physical

Page Tables

Internal operation of MMU with 16 4 KB pages



Translation Lookaside Buffer – TLB

TLB – Translation Lookaside Buffer

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Goal is to speed up paging

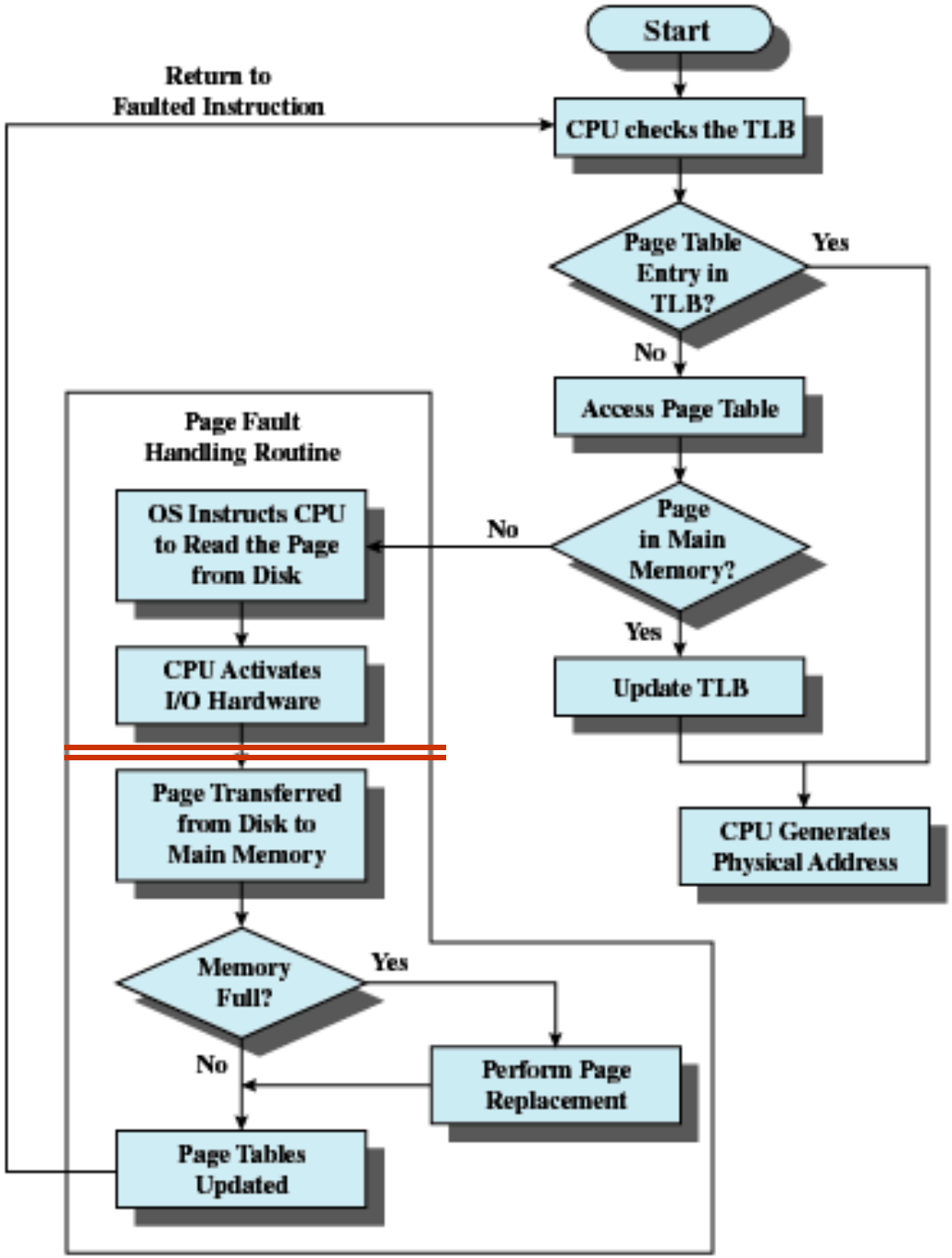
TLB is a cache in MMU for page table entries

TLB – translation lookaside buffer

- Part of memory management unit (MMU)
  - Cache for used page table entries to avoid extra memory access during address translation
- Associative search
  - Compare with all elements at the same time (fast)
- Each TLB element contains: page number, page table entry, validity bit
- Each process uses the same page numbers 0, 1, 2, …, stored on different page frames
  - TLB must be cleared during process switch
  - At least clear the validity bits (this is fast)

Operation of Paging and TLB

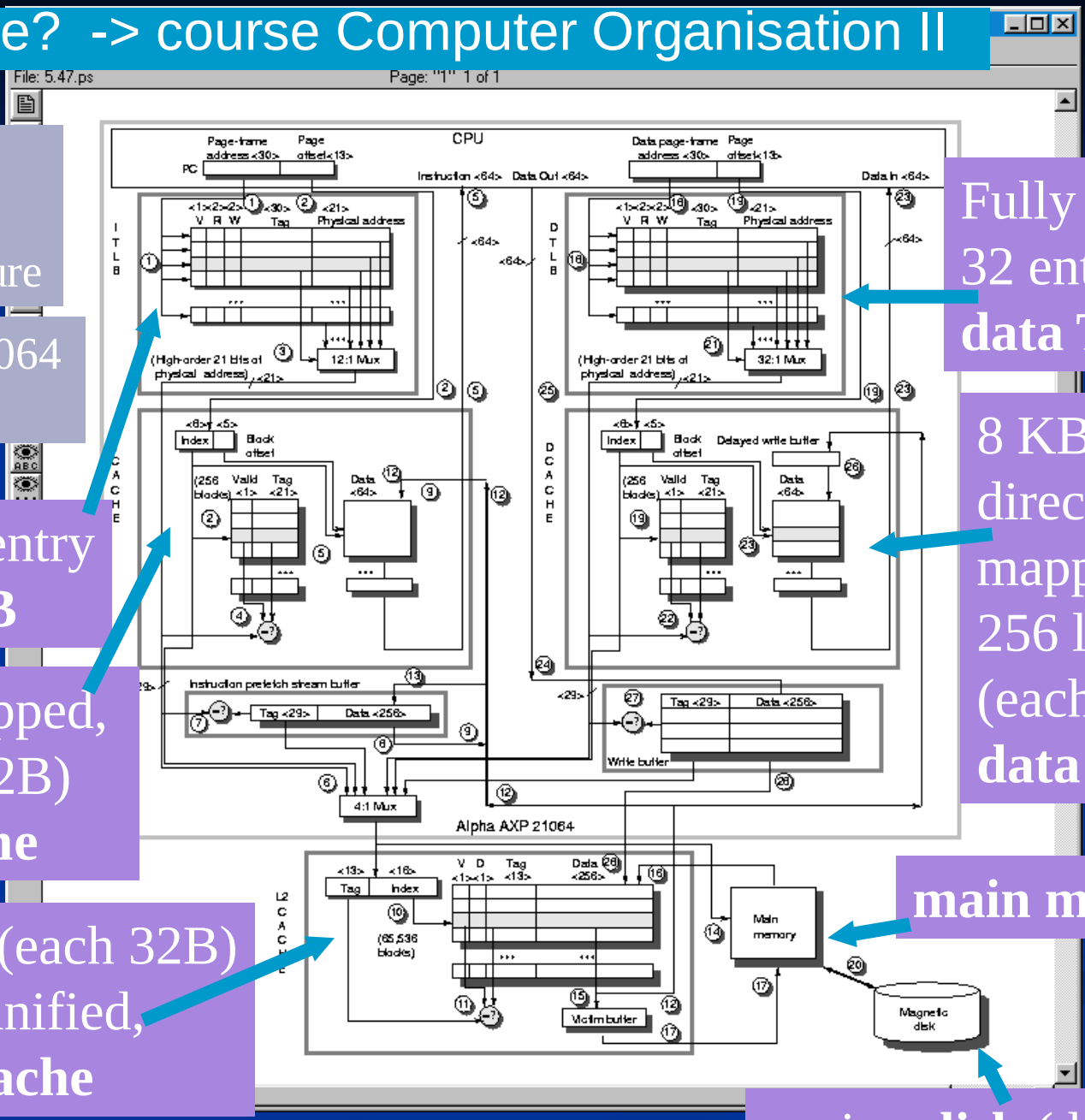
Sta Fig 8.8.



More? -> course Computer Organisation II

Fig. 5.47 from Hennessy-Patterson, Computer Architecture

DEC Alpha AXP 21064 memory hierarchy



Fully assoc, 12 entry **instruction TLB**

8 KB, direct mapped, 256 line (each 32B) **instruction cache**

2 MB, 64K line (each 32B) direct mapped, unified, write-back **L2 cache**

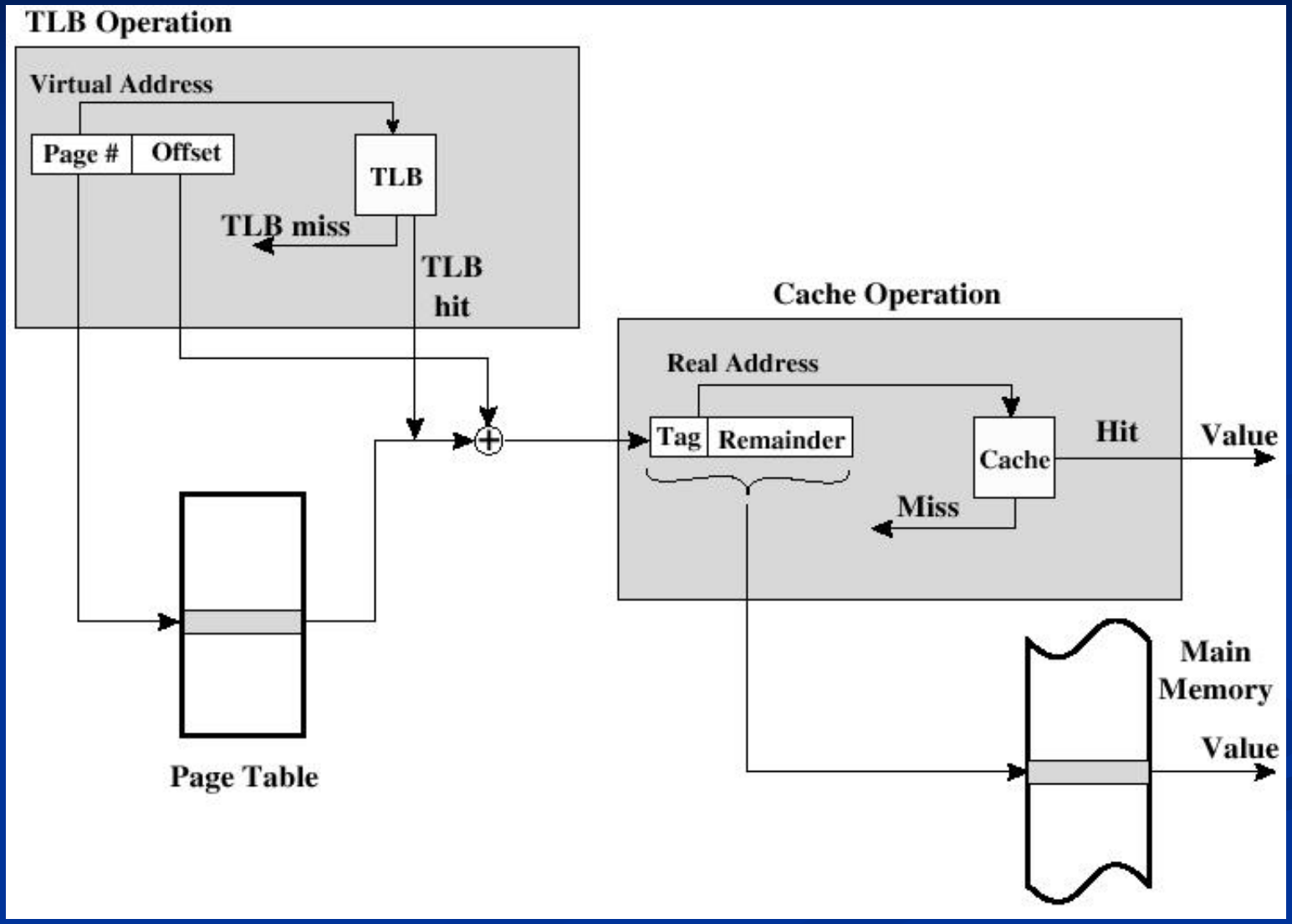
Fully assoc, 32 entry **data TLB**

8 KB, direct mapped, 256 line (each 32B) **data cache**

main memory

paging **disk** (dma)

TLB and cache



Sta Fig. 8.10

Multilevel and Inverted Page Tables

Multilevel page table

- Large virtual address space
  - Logical address could be 32- or 64-bits
- Each process has a large page table
  - Using 32-bit address and frame size 4KB (12 bit offset), means 2^{20} = 1M of page tables entries for a single process
  - Each entry requires several bytes (lets say 4 bytes), so the final size of page table could be for example 4 MB
- Thus the page table is divided to several pages also and part of it can be on the disk
  - Only the part of the page table that covers the pages used currently in the execution of the process is in memory

Two-level hierarchical page table

- Top most level in one page and always in the memory

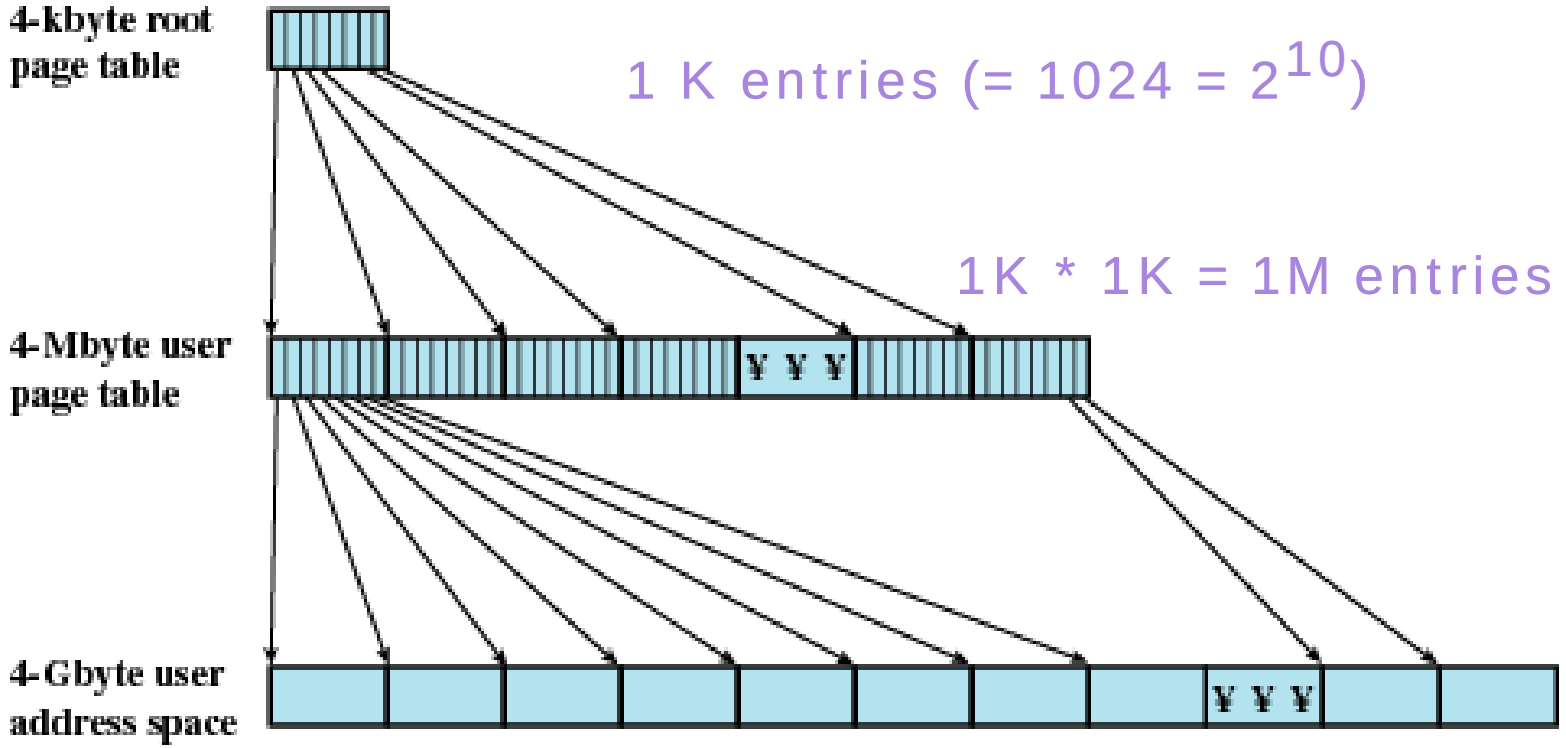
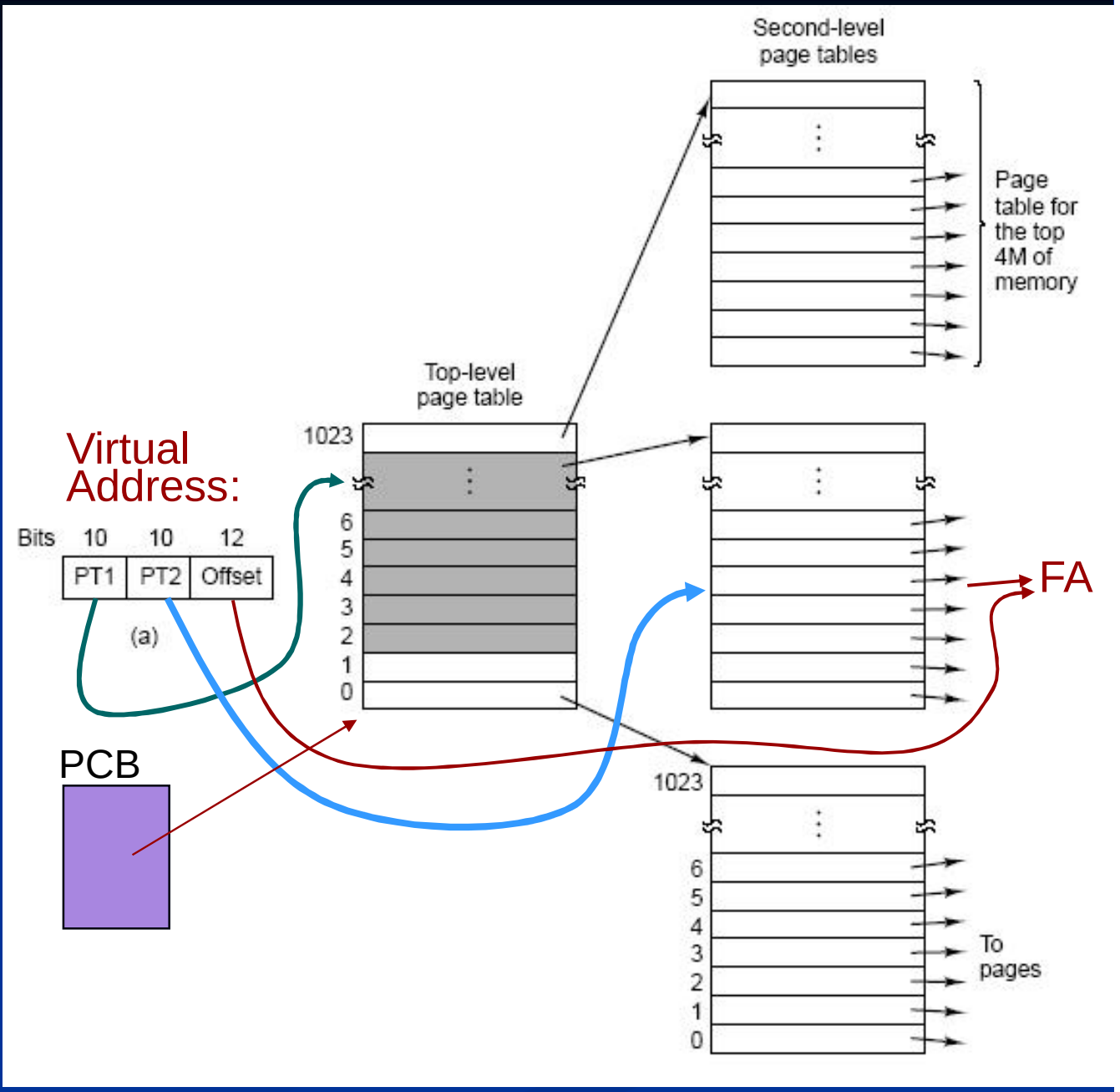


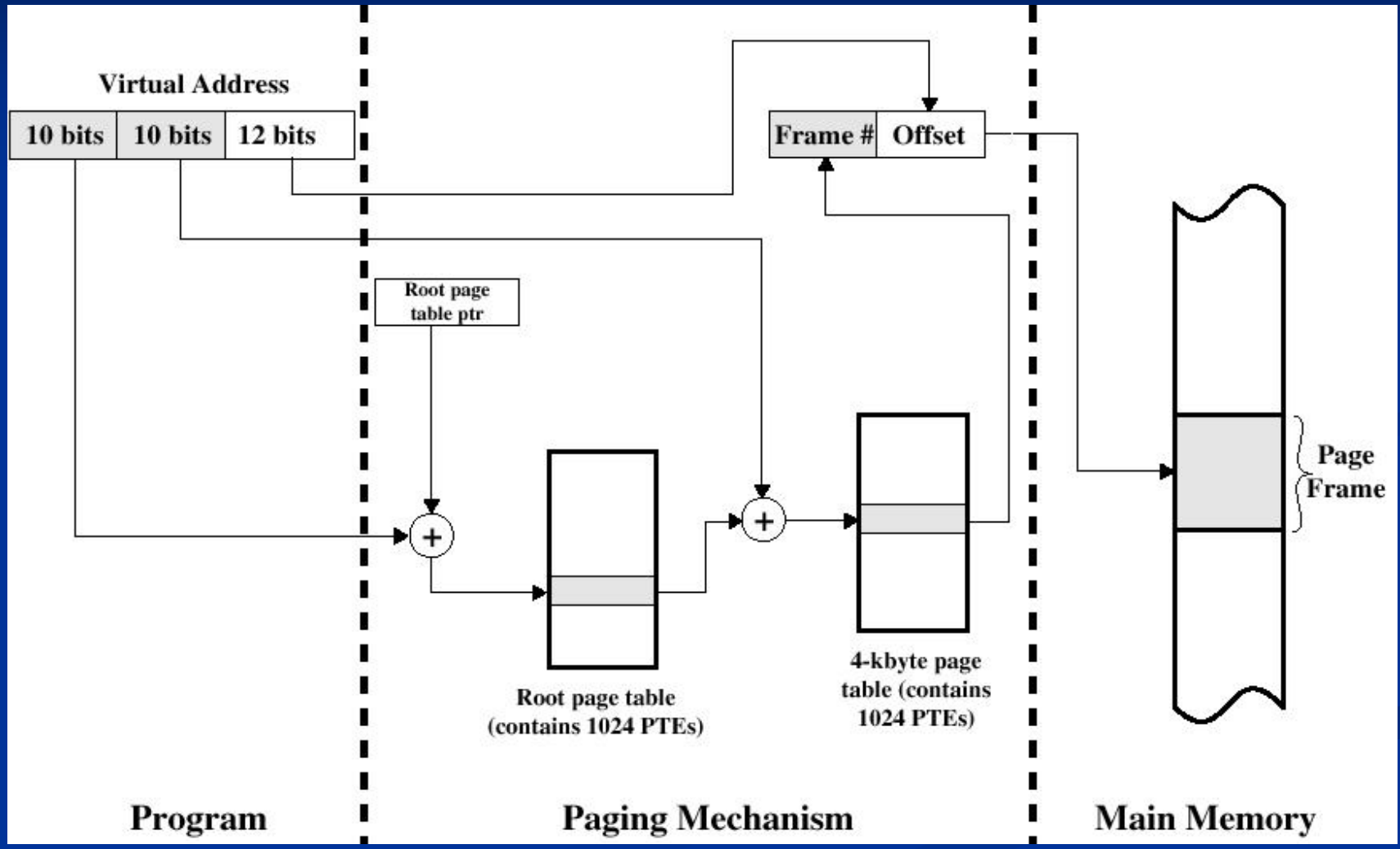
Figure 8.4 A Two-Level Hierarchical Page Table

Address translation with two levels



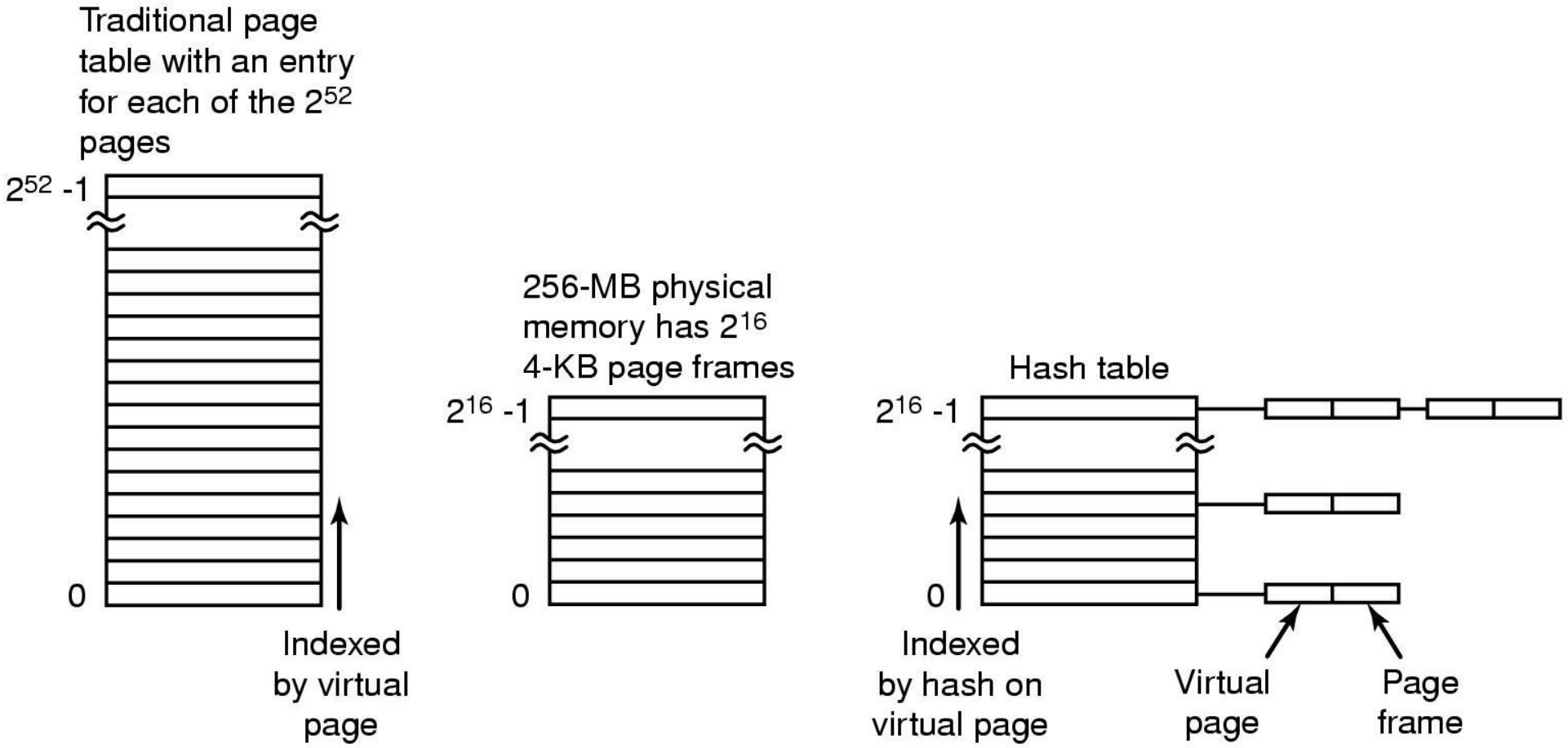
(Fig 4-12 [Tane01])

Address translation with two levels



Sta Fig 8.5

Inverted Page Tables



Comparison of a traditional page table with an inverted page table

Inverted page table

- Physical memory often smaller than the virtual address space of processes
- Invert booking: Store for each page frame what page (of which process) is stored there
  - Only one global table (inverted page table), one entry for each page frame.
- Search for the page based on the content of the table
  - Inefficient, if done sequentially,
  - Use hash to calculate the location, start search from there
- If page not found, page fault
- Useful, only if TLB is large

Inverted page table

Sta Fig 8.6

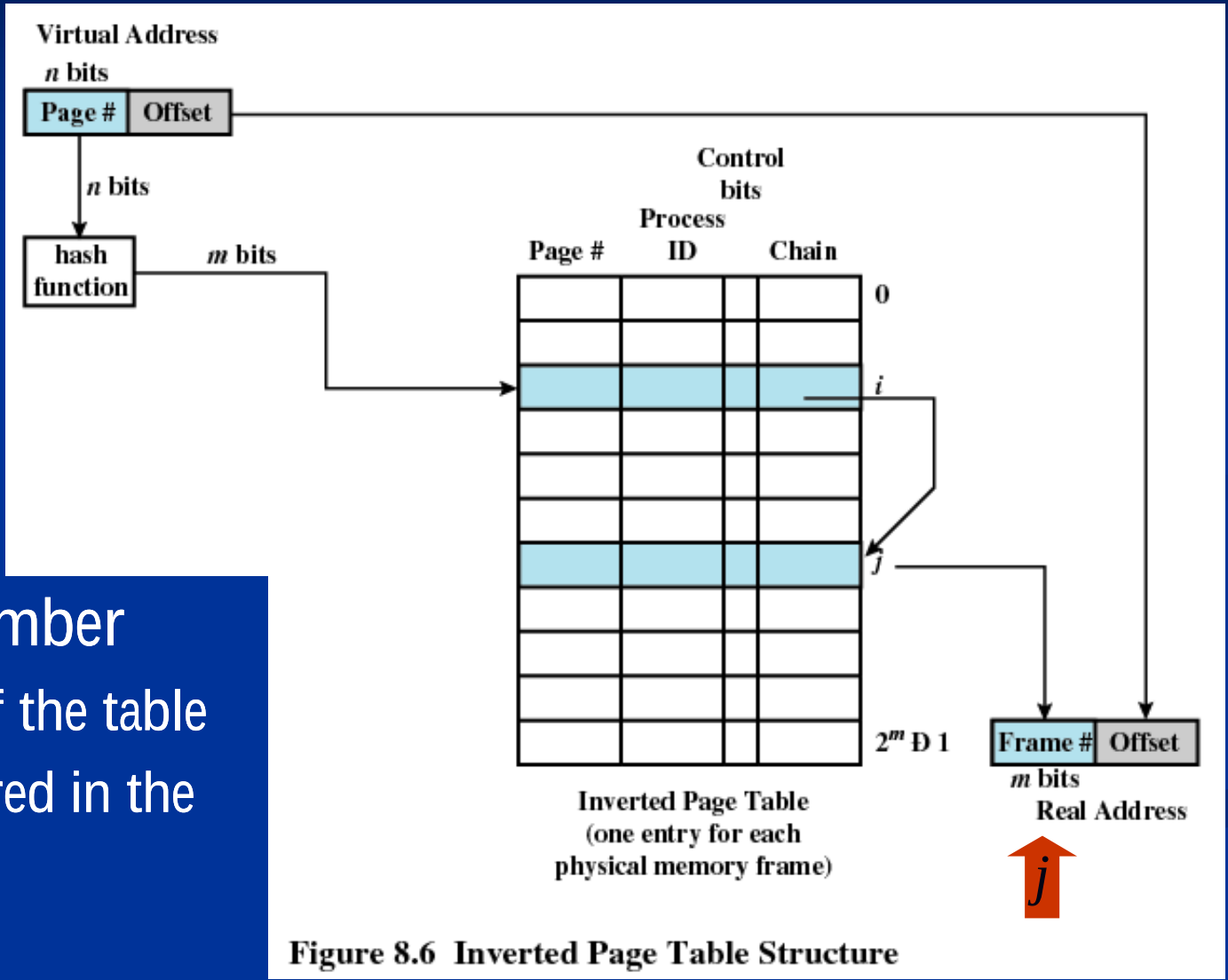


Figure 8.6 Inverted Page Table Structure

- Frame number
  - Index of the table
  - Not stored in the entry