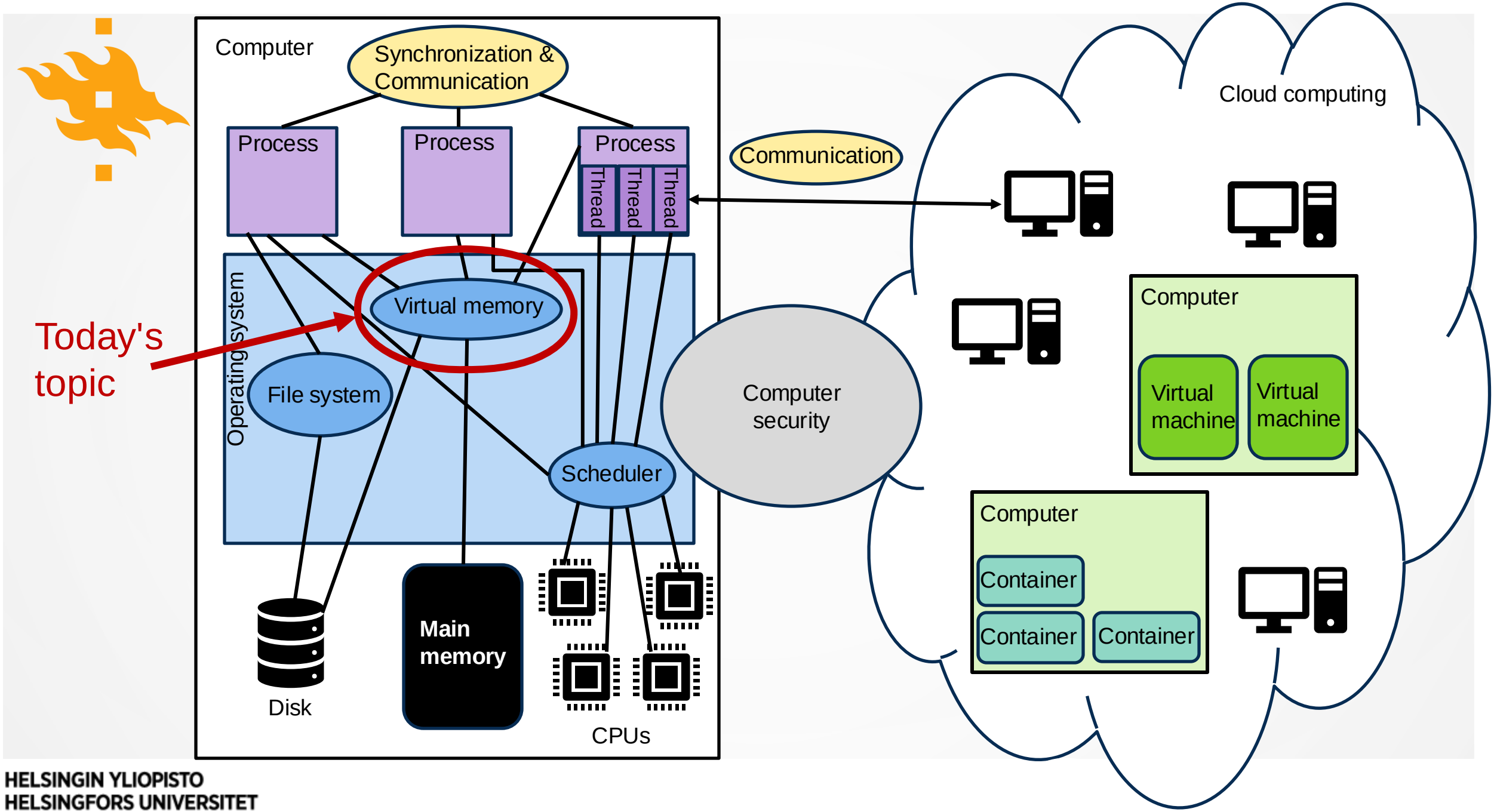
COMPUTING PLATFORMS

Virtual Memory:
Paging, segmentation, and address translation

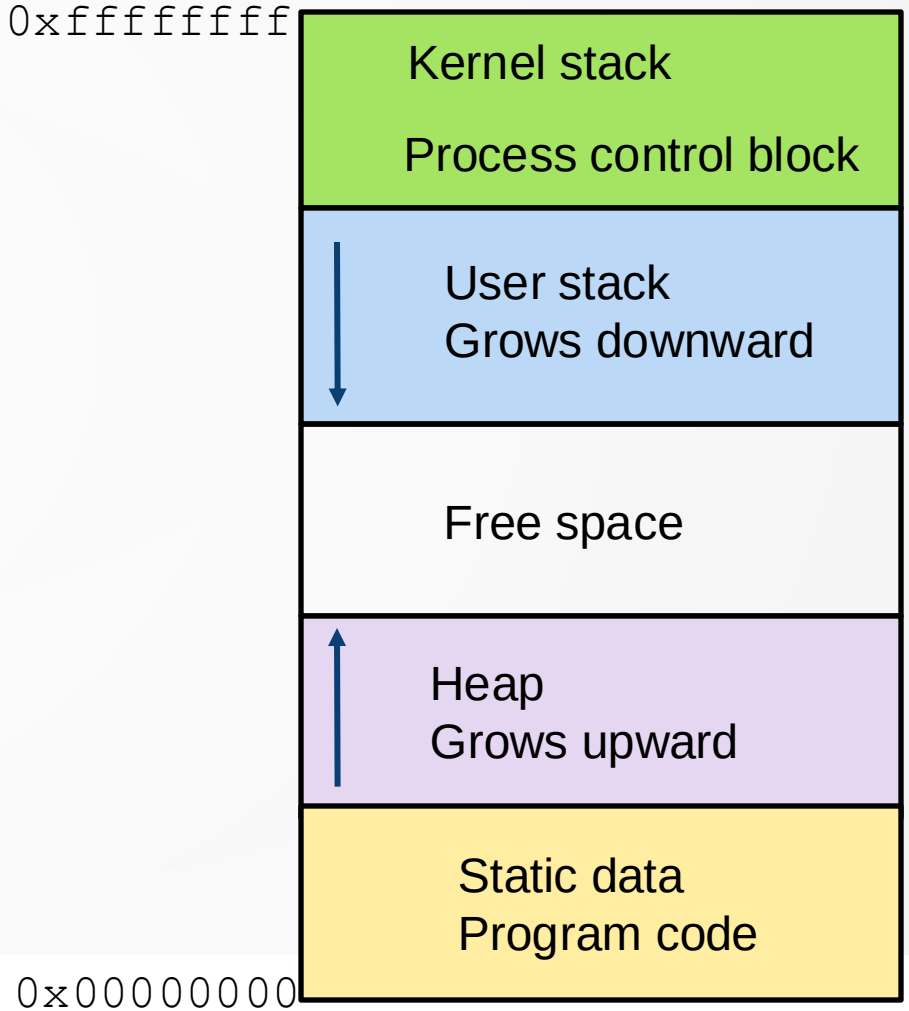

Computer
Synchronization & Communication
Process
Process
Process
Thread
Thread
Thread
Operating system
Virtual memory
File system
Scheduler
Main memory
Disk
CPUs
Communication
Computer security
Cloud computing
Computer
Virtual machine
Virtual machine
Computer
Container
Container
Container
Today's topic

LEARNING OUTCOMES

- After today's lecture you
  - Know the requirements for memory management
  - Understand the concepts of segmentation and paging and can compare their advantages and disadvantages
  - Can describe hardware and OS structures that support segmentation and paging

REMINDER: HOW THE USER PROCESS SEES THE MEMORY

- Each process has its own linear address space and can reference addresses within its address space e.g. in range `0x00000000`–`0xffffffff`
- Most programs are organized into **modules**, some unmodifiable (read only, execute only) and some contain data that can be modified
  - Modules can be written and compiled independently, with references between modules resolved by the system at run time
  - Different degree of protection (read only, execute only) can be given to different modules
  - Some modules (library code) are shared between different processes



HOW THE OPERATING SYSTEM SEES MEMORY

- In a multiprogramming system, the main memory is divided to accommodate all processes
- The main memory may be smaller than the address space of a user process
- At least two levels of memory (in real systems more): **main memory** and **secondary memory**
  - Main memory provides fast access at relatively high cost
  - Main memory is volatile (not permanent storage)
  - Secondary memory is slower and cheaper but (usually) not volatile
  - Secondary memory of large capacity used for long-term storage of data and programs, while smaller main memory holds programs and data currently in use

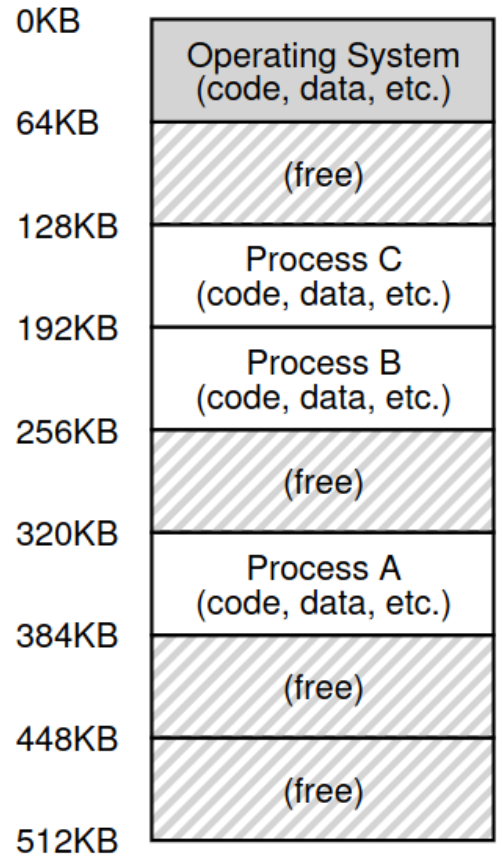


Figure 13.2: **Three Processes: Sharing Memory**

Figure from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

HOW TO ACCOMMODATE BOTH USER PROCESS'S AND OS'S POINT OF VIEW?

- User processes use **virtual addresses** to reference the process image
- **Address translation** is used to translate **virtual addresses** referencing the address space of the process to **physical addresses** referencing physical memory locations
- Both hardware and OS support needed to implement memory management!

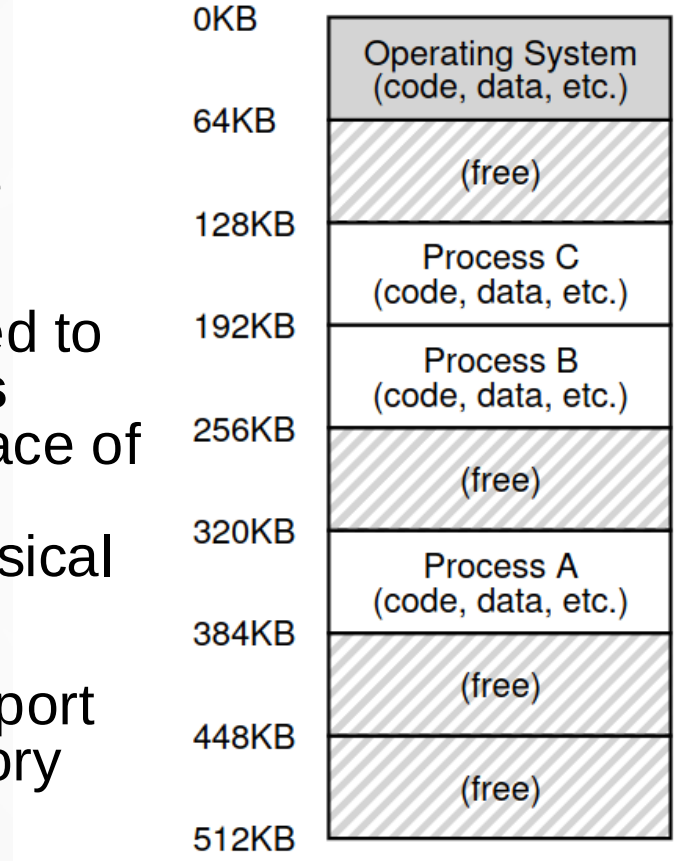


Figure 13.2: **Three Processes: Sharing Memory**

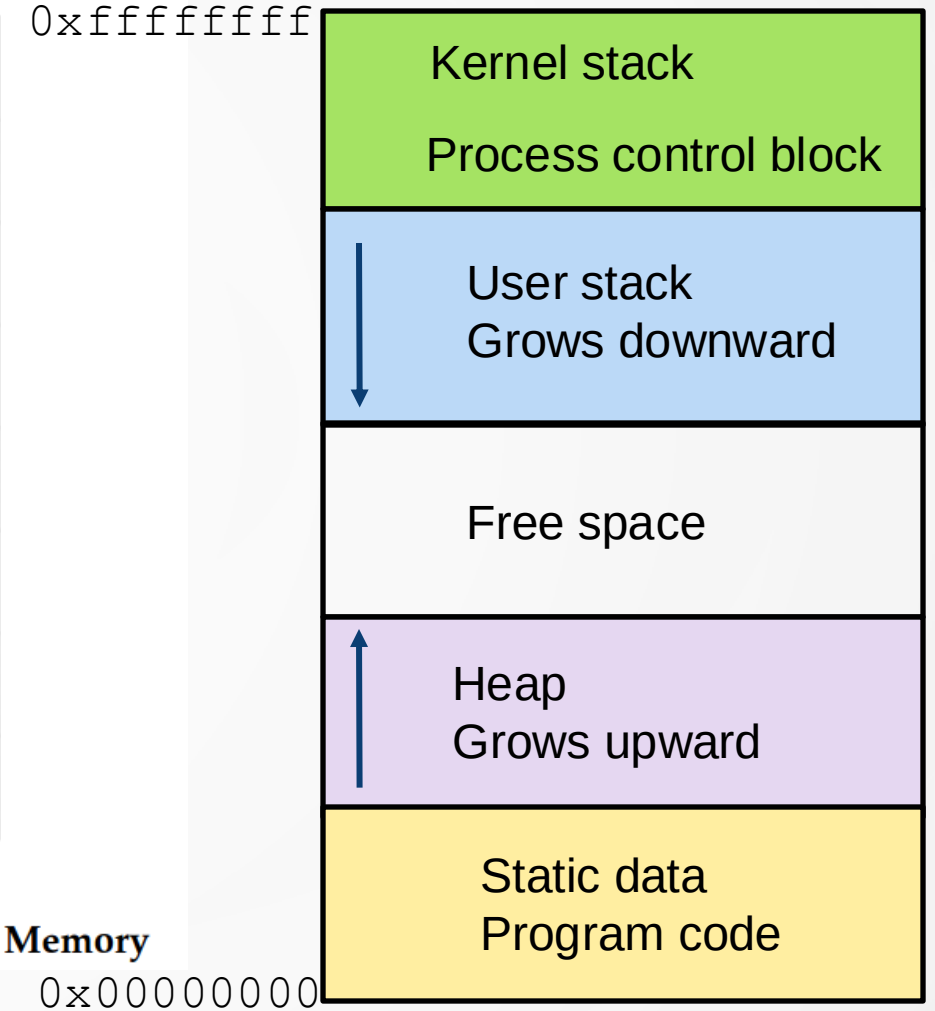


Figure from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

MEMORY MANAGEMENT REQUIREMENTS

- **Relocation**: the process image can be loaded to different locations of the physical memory
  - Invisible to the running program: Programmer should not need to know where in physical memory code, variables etc. will be placed
  - Enables swapping processes in and out of memory
- **Efficiency**
  - Programs should not run slower due to memory management
  - Structures supporting memory management should not take too much space
- **Protection & isolation**
  - Process should not be able to access or modify the address space of another process
  - Protect process from itself: some parts of address space should be execute-only (code), read-only, ...

RELOCATION AND ADDRESS TRANSLATION

- Simple example of address translation used in some early systems
- **Virtual address** (also **logical address**)
  - Reference to a memory location independent of the current assignment of data to memory
- **Relative address** (type of virtual address)
  - Address expressed as location relative to some known point (e.g. **base register**)
  - **Bounds register** gives limit of valid addresses
- **Physical address**
  - Actual location of data in main memory

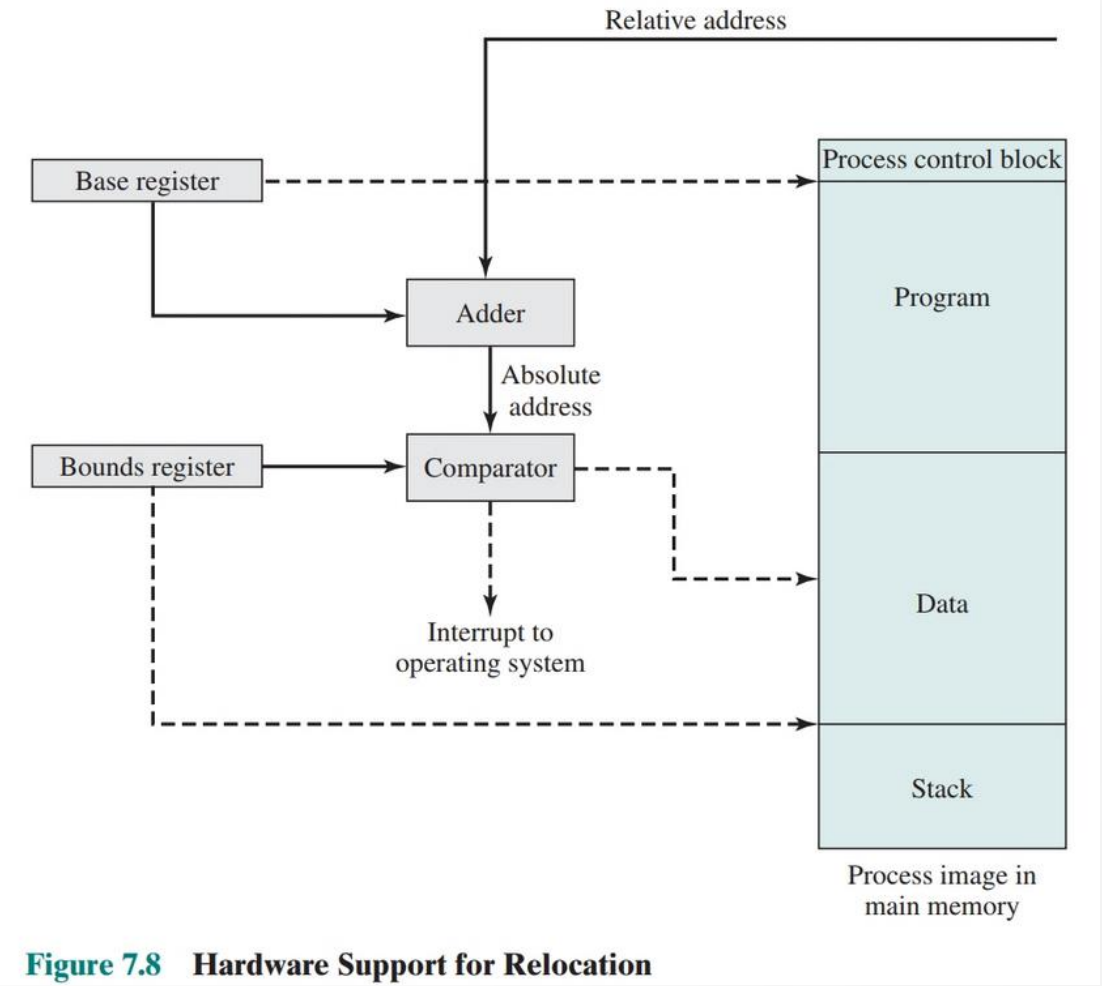


Figure 7.8 Hardware Support for Relocation

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

RELOCATION AND ADDRESS TRANSLATION: EXAMPLE

- Base register: 0001 0000 0000 0000 = 4096
- Bounds register: 0001 1000 0000 0000 = 6144
- Relative address: 0000 0011 0110 0111 = 439

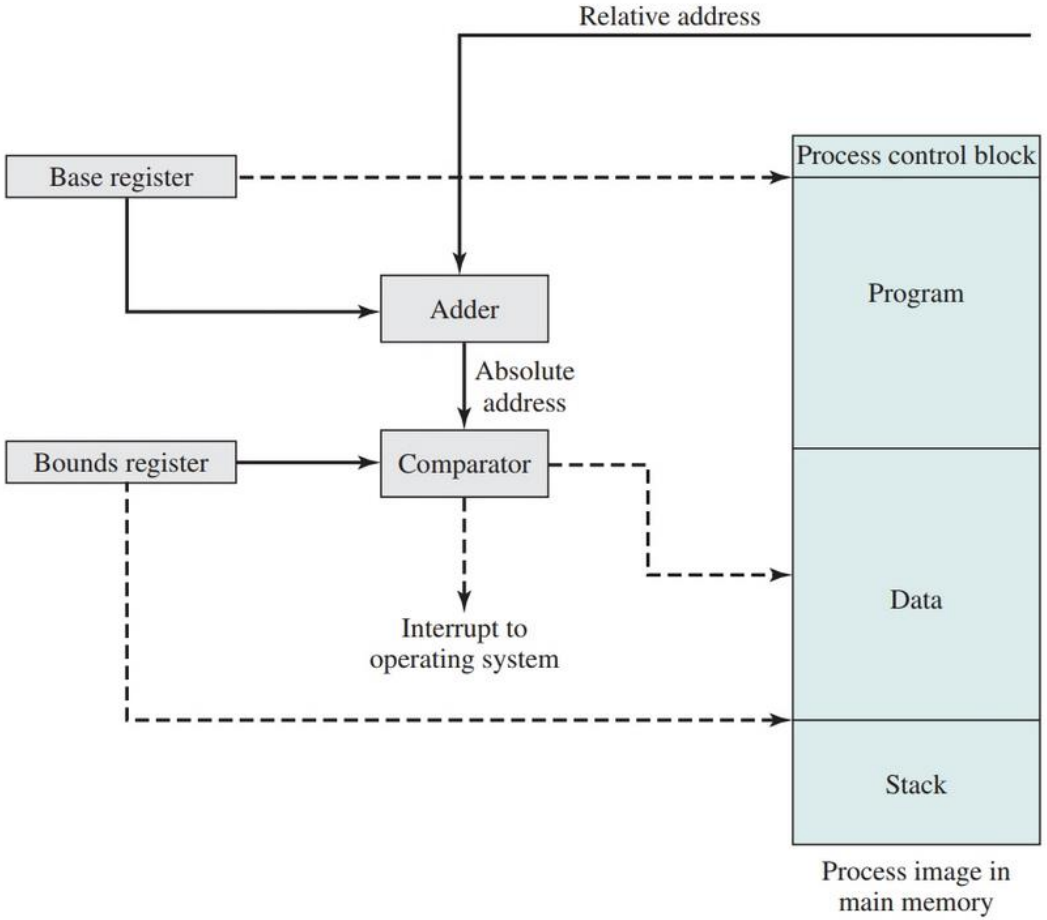


Figure 7.8 Hardware Support for Relocation

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

RELOCATION AND ADDRESS TRANSLATION: EXAMPLE

- Base register: 0001 0000 0000 0000 = 4096
- Bounds register: 0001 1000 0000 0000 = 6144
- Relative address: 0000 0011 0110 0111 = 439
- Physical address: Base + Relative address
  = 0001 0000 0000 0000 + 0000 0011 0110 0111
  = 0001 0011 0110 0111
  = 4535
- Physical address is less than value in Bounds register -> Address is valid

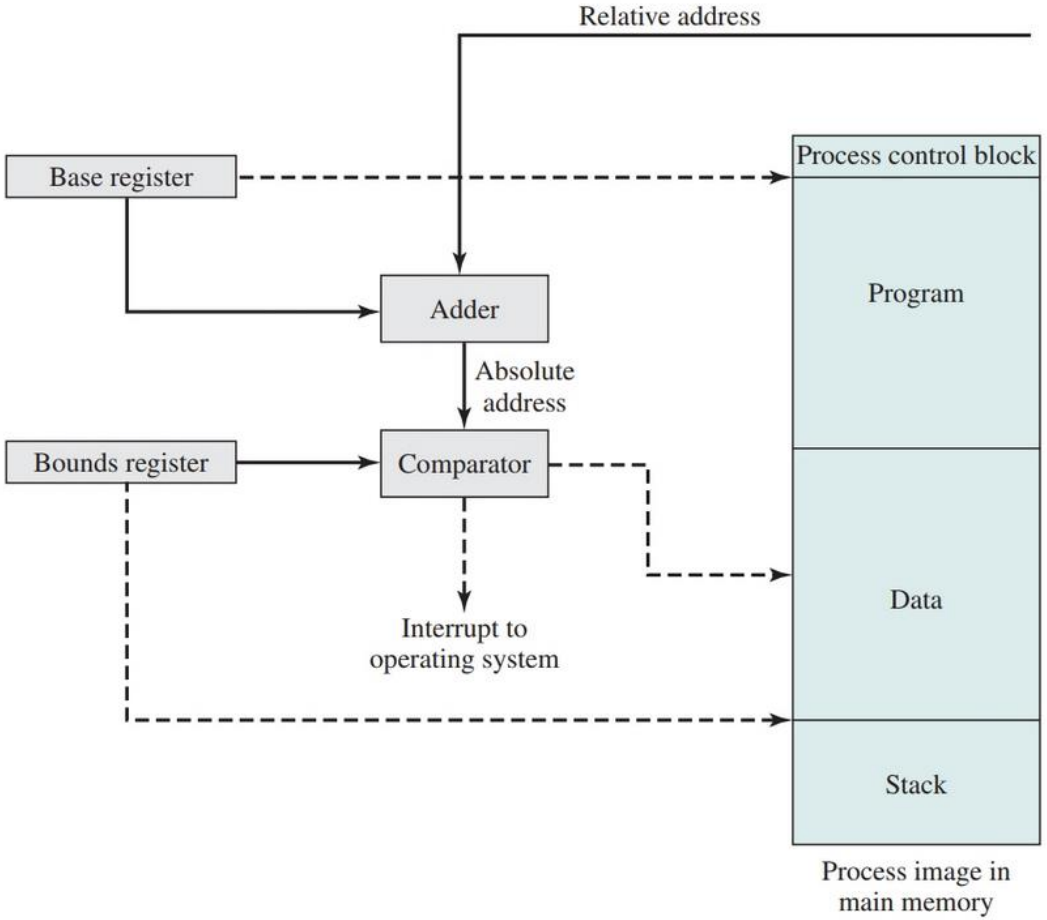


Figure 7.8 Hardware Support for Relocation

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

RELATIVE ADDRESSES

- Allows processes to be relocated (swapped in and out of memory)
- Process image is handled as one continuous block of memory
  - Need to find a continuous free block in main memory to accommodate the process
  - Cannot set different permissions (execute-only, read-only) for different parts of the process image
  - Does not provide a mechanism to share modules between processes
- Provides protection from other processes: Each process's addresses are contained within the values of Base and Bounds registers

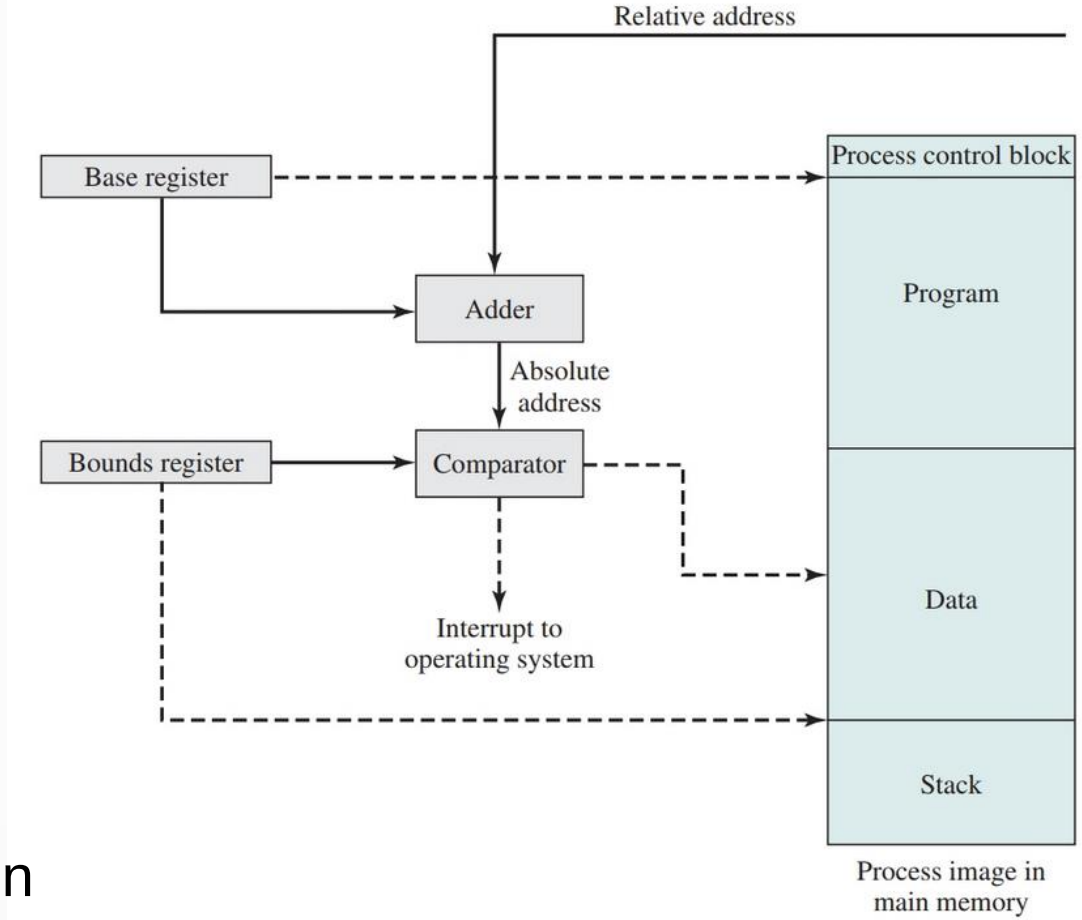


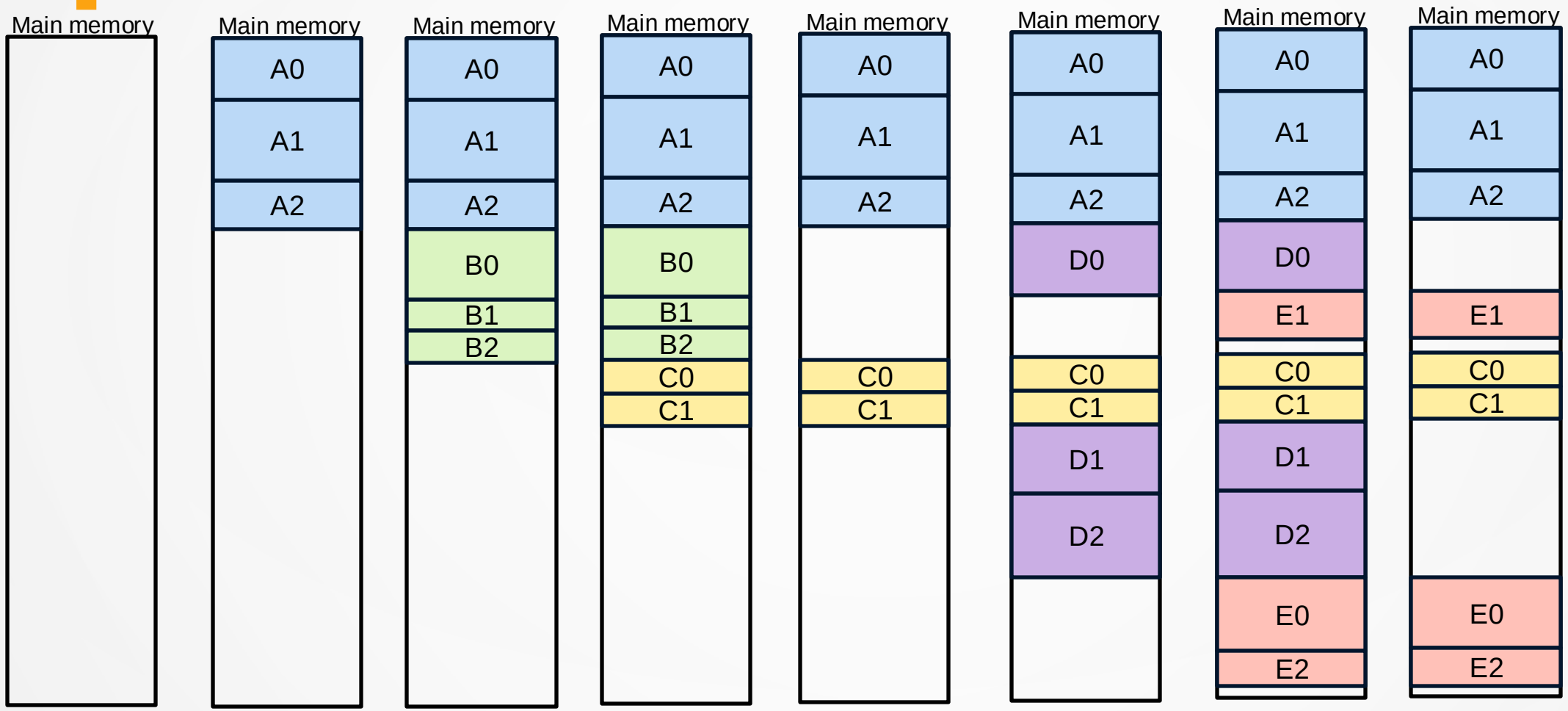
Figure 7.8 Hardware Support for Relocation

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

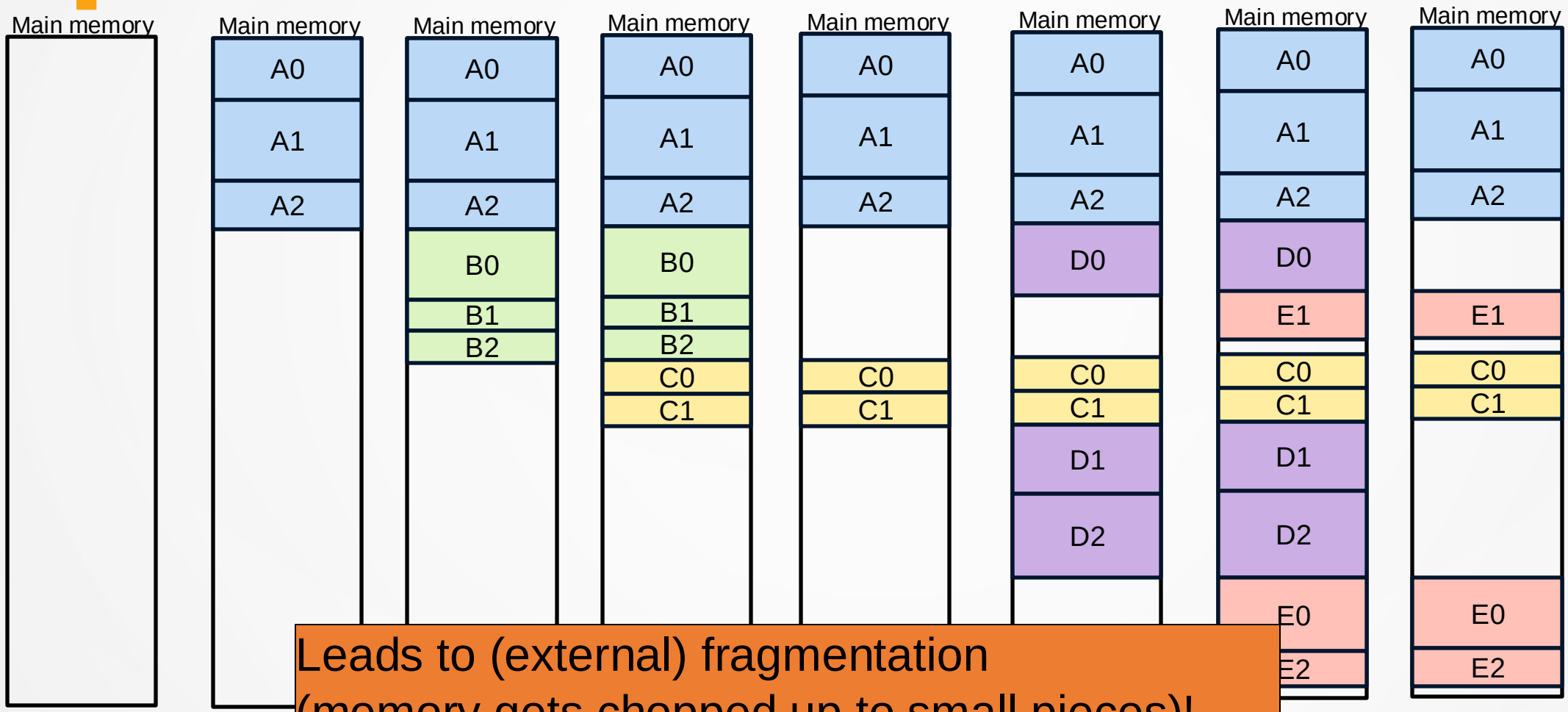
SEGMENTATION

- Program can be divided into **segments**
  - **Segment** is a variable length block of data (there is a maximum length)
- Virtual addresses consist of two parts
  - Segment number
  - Offset
- Visible to the programmer
- Typically code and data are assigned to different segments
  - Different permissions (execute-only, read-only) can be set for different segments
- Program or data can be broken down to multiple segments
  - Programmer must be aware of maximum segment size

SEGMENTATION EXAMPLE



SEGMENTATION EXAMPLE



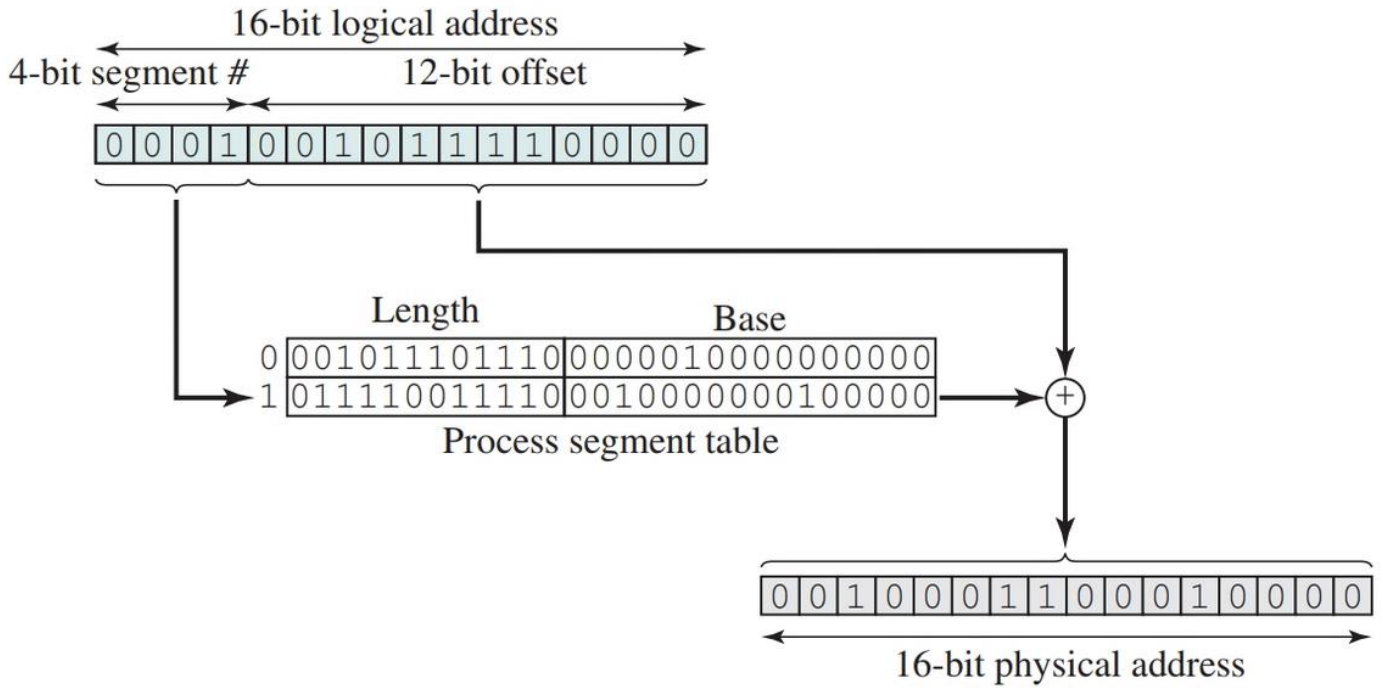
Leads to (external) fragmentation
(memory gets chopped up to small pieces)!

SEGMENTATION: ADDRESS TRANSLATION

- Due to unequal segment sizes, **there is no simple relationship between virtual and physical addresses**
- Consider a virtual address of *n*+*m* bits, where the leftmost *n* bits give the segment number and the rightmost *m* bits give the offset.
- The steps of address translation
  1. **Extract the segment number** as the *n* leftmost bits of the virtual address
  2. Use the segment number as an index to the process segment table to find the starting physical address of the segment
  3. **Compare the offset** (rightmost *m* bits of the virtual address) to the length of the segment. If offset ≥ segment length, **address is invalid.**
  4. Physical address is **sum of the starting physical address of the segment and the offset**

SEGMENTATION: ADDRESS TRANSLATION EXAMPLE

- Virtual address:
  0001 0010 1111 0000



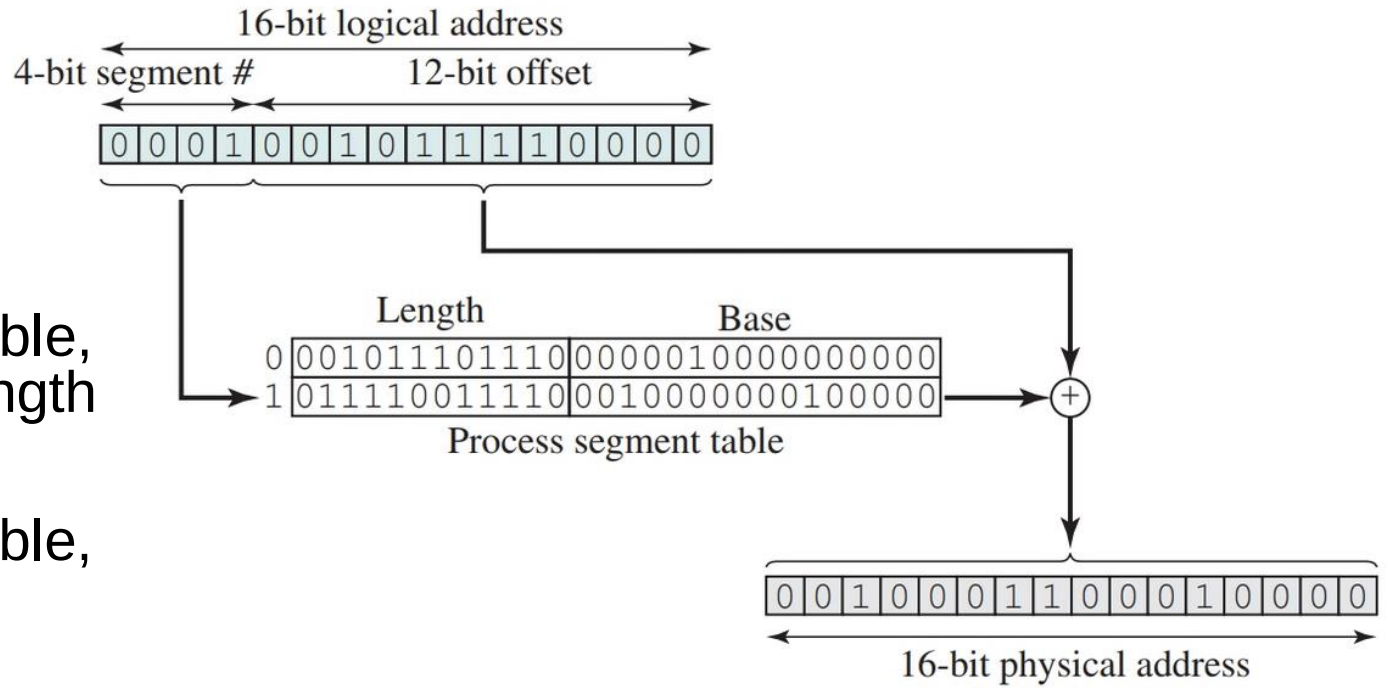
(b) Segmentation

Figure 7.12 **Examples of Logical-to-Physical Address Translation**

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

SEGMENTATION: ADDRESS TRANSLATION EXAMPLE

- Virtual address:
  0001 0010 1111 0000
  - Segment 0001=1
  - Offset 0010 1111 0000 = 752
- According to process segment table, offset is smaller than segment length -> address is valid
- According to process segment table, this segment starts at physical address 0010 0000 0010 0000
- Physical address is then
  0010 0000 0010 0000 +
  0010 1111 0000 =
  0010 0011 0001 0000



(b) Segmentation

Figure 7.12 Examples of Logical-to-Physical Address Translation

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

SEGMENTATION: HARDWARE SUPPORT

- Memory management unit (MMU) has a **segment table pointer register:** points to the beginning of the current process's segment table
- Segment table has a structure dictated by MMU
- MMU does the address translation based on the segment table at runtime

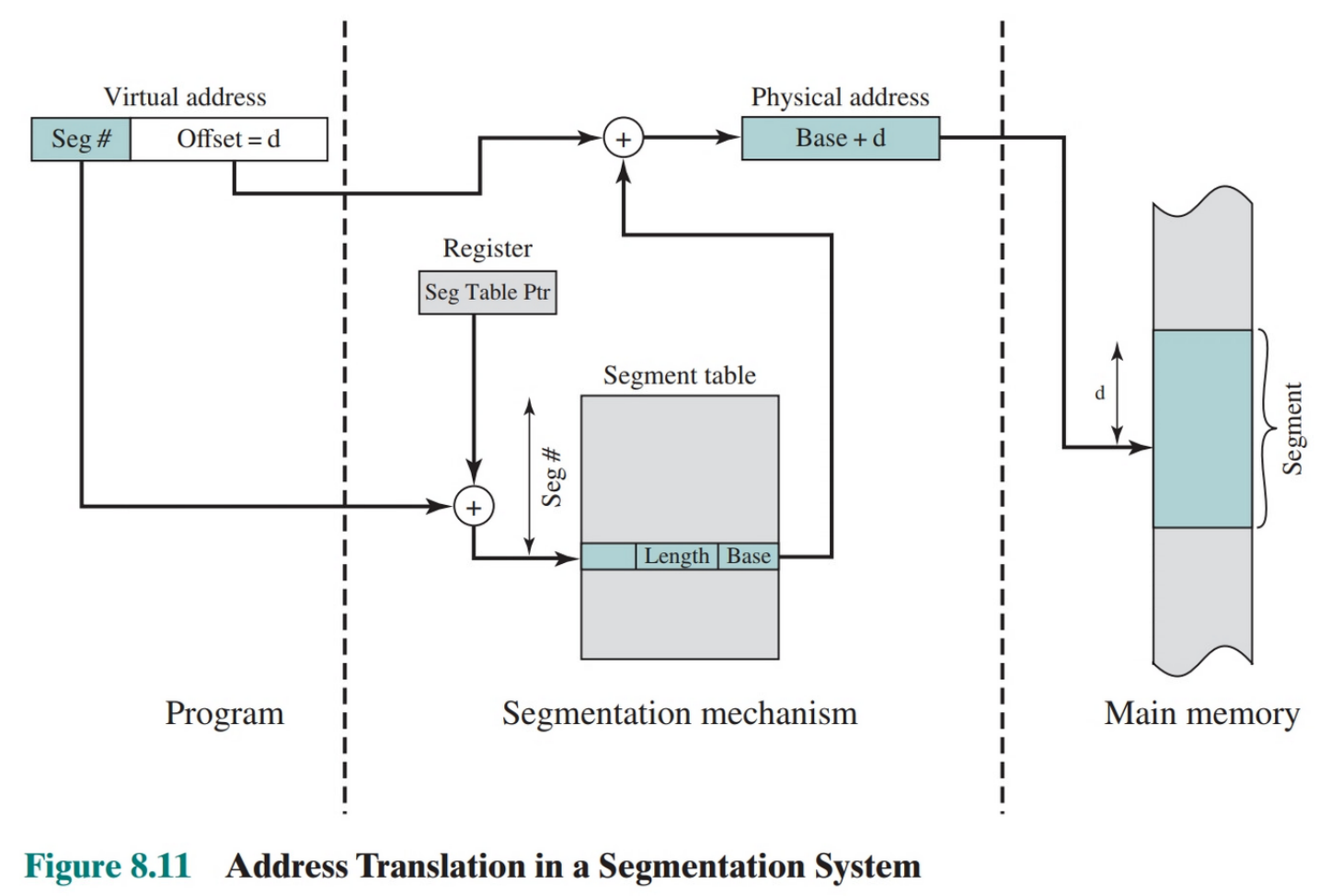


Figure 8.11 **Address Translation in a Segmentation System**

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

SEGMENTATION

- Allows processes to be relocated (swapped in and out of memory)
- Process image is divided into segments
  - Different permissions (execute-only, read-only) can be set for different segments
  - Allows sharing at segment level
  - Simplifies handling of growing data structures
  - Allows programs to be altered and recompiled independently
- Each segment is handled as one continuous block of memory
  - Need to find a continuous free block in main memory to accommodate the segment
  - Leads to **fragmentation:** free space gets chopped up to small pieces
- Provides protection from other processes: Each segment's addresses are contained within the values of base address and (base + length) as dictated in the segment table

PAGING

- Partition main memory into relatively small, **equal fixed-size** chunks
  - **Frames** = available chunks of memory
- Partition processes also into small, **fixed-size chunks** (same size!)
  - **Pages** = chunks of process (address space)
- **Page table**
  - OS maintains for each process
  - Contains frame location of each page in the process
  - Processor produces physical addresses (must know how to access the current process)
- Invisible to the programmer

PAGING: EXAMPLE

Frame number	Main memory
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	

Frame number	Main memory
0	A0
1	A1
2	A2
3	A3
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	

Frame number	Main memory
0	A0
1	A1
2	A2
3	A3
4	B0
5	B1
6	B2
7	
8	
9	
10	
11	
12	
13	

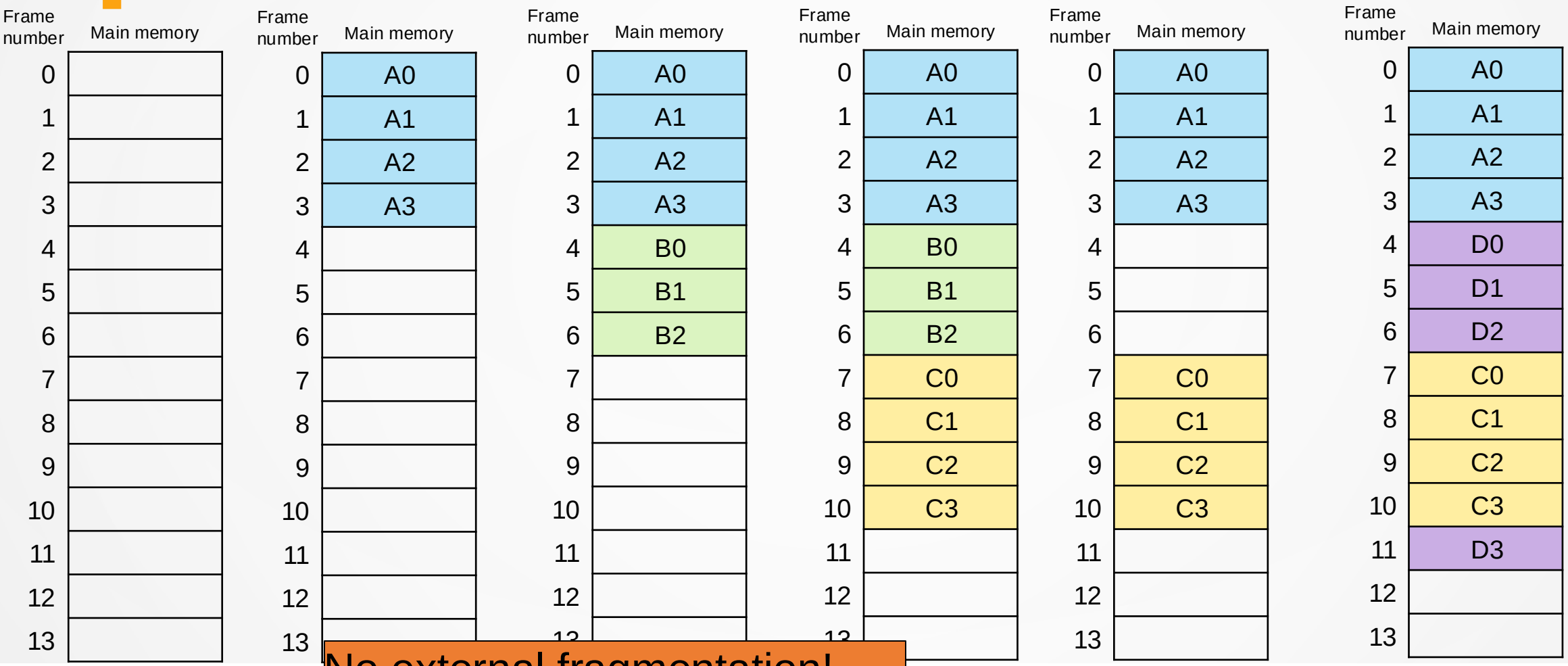
Frame number	Main memory
0	A0
1	A1
2	A2
3	A3
4	B0
5	B1
6	B2
7	C0
8	C1
9	C2
10	C3
11	
12	
13	

Frame number	Main memory
0	A0
1	A1
2	A2
3	A3
4	
5	
6	
7	C0
8	C1
9	C2
10	C3
11	
12	
13	

Frame number	Main memory
0	A0
1	A1
2	A2
3	A3
4	D0
5	D1
6	D2
7	C0
8	C1
9	C2
10	C3
11	D3
12	
13	

Figure adapted from Figure 7.9 in [Stallings, Operating systems: Internals and design principles, 9th ed]

PAGING: EXAMPLE

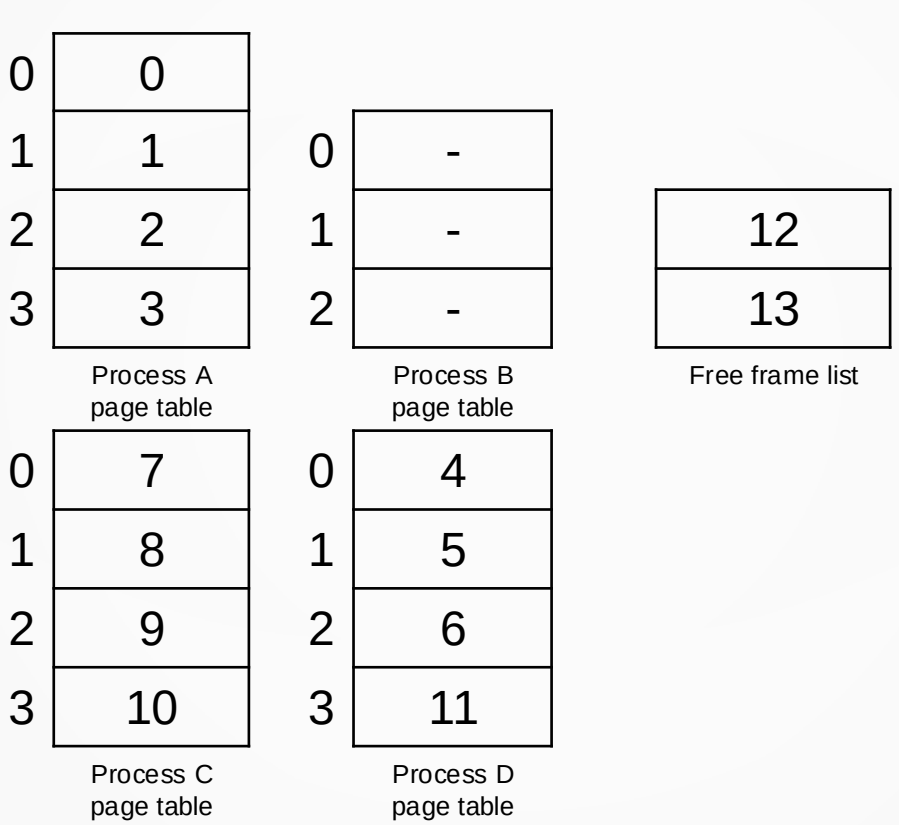


No external fragmentation!

Figure adapted from Figure 7.9 in [Stallings, Operating systems: Internals and design principles, 9th ed]

PAGING: EXAMPLE – PAGE TABLES

- Process page tables and the free space list for the example (last step)



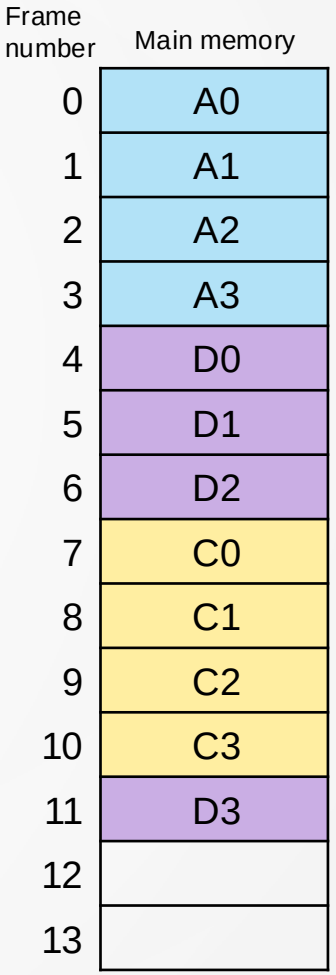


Figure adapted from Figure 7.9 in [Stallings, Operating systems: Internals and design principles, 9th ed]

PAGING: ADDRESS TRANSLATION EXAMPLE

- Virtual address
  0000 0101 1101 1110

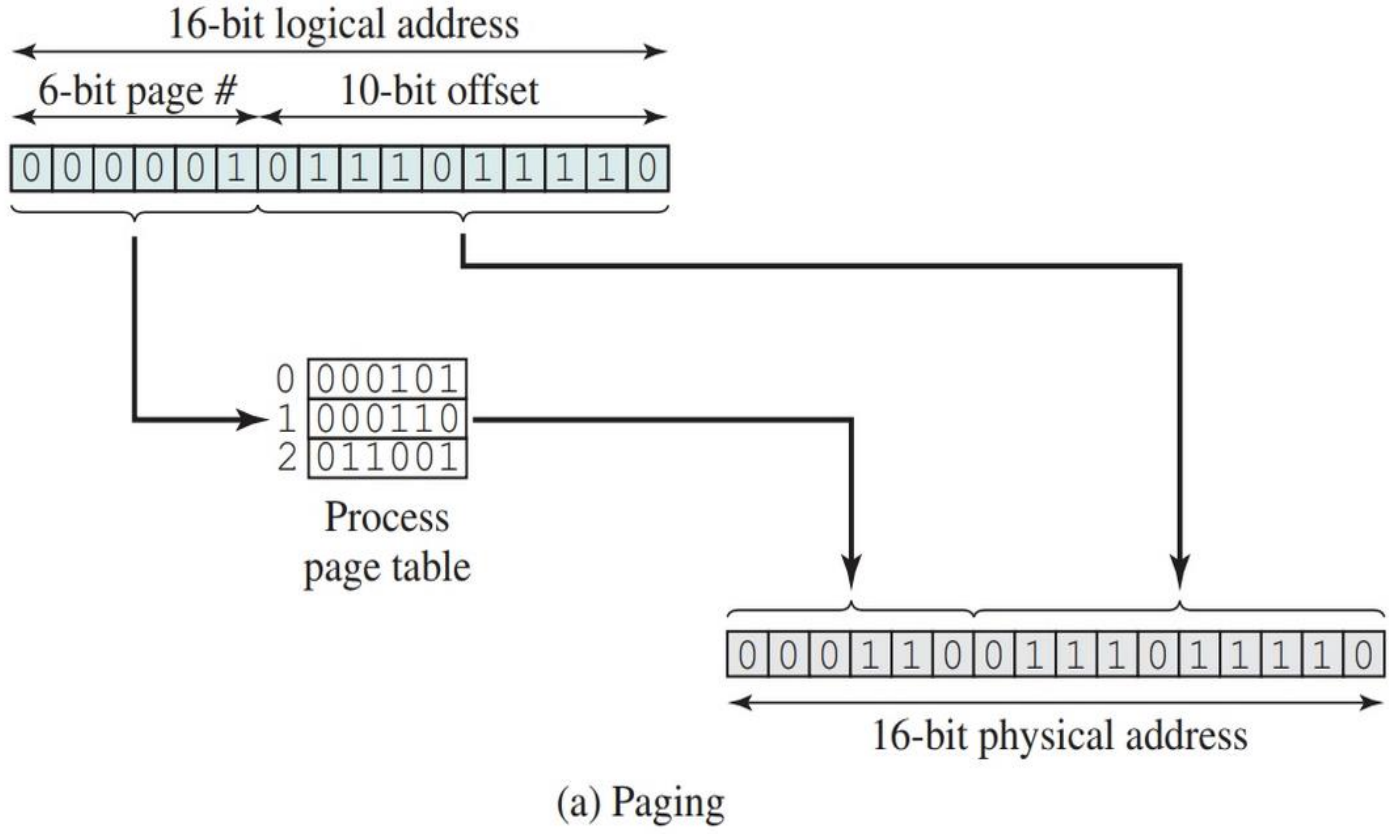


Figure 7.12 **Examples of Logical-to-Physical Address Translation**

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

PAGING: ADDRESS TRANSLATION EXAMPLE

- Virtual address
  0000 0101 1101 1110
  - Page number: 0000 01 = 1
  - Offset: 01 1101 1110 = 478
- According to the process page table page 1 resides in frame
  0001 10=6
- The physical address is then the concatenation of frame number and offset:
  0001 1001 1101 1110

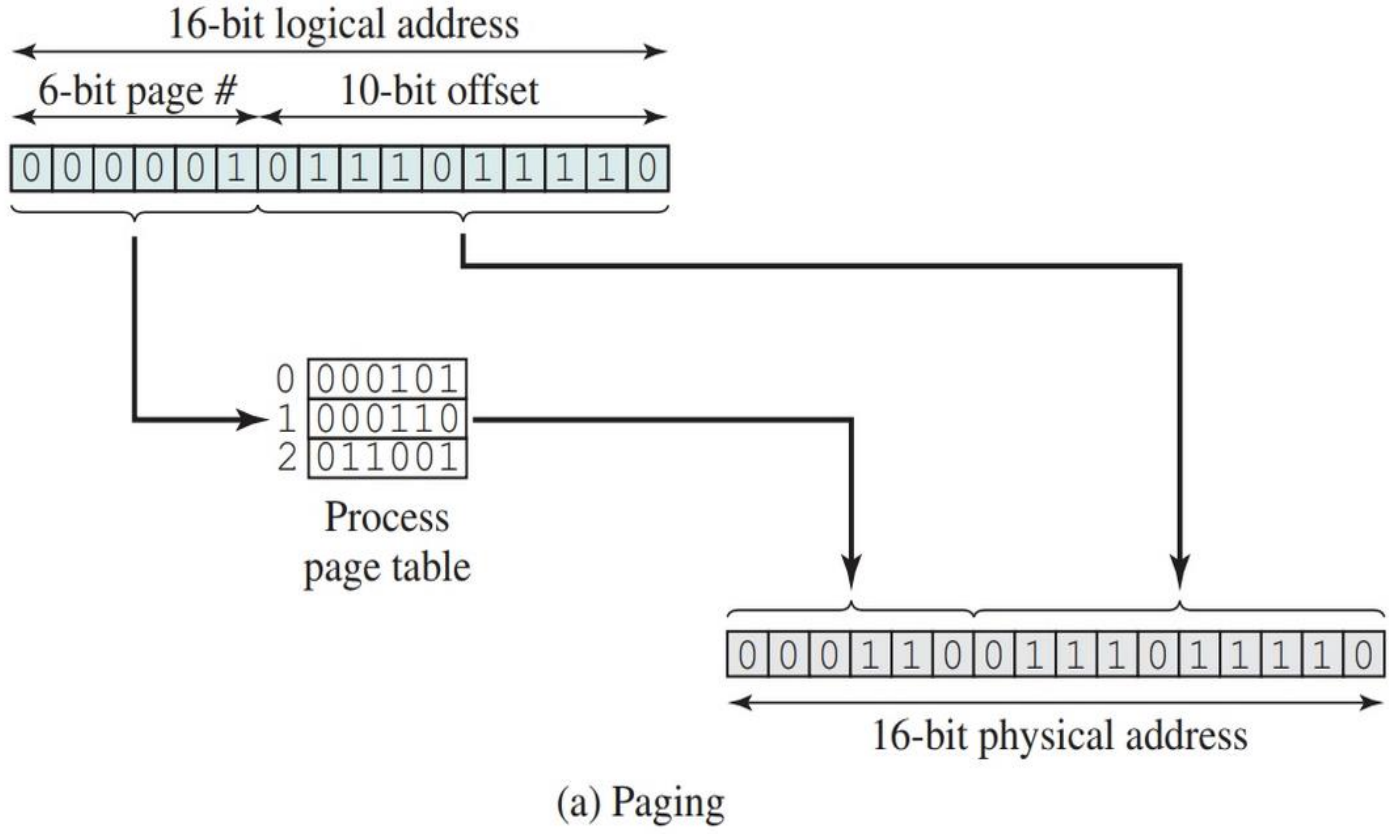


Figure 7.12 Examples of Logical-to-Physical Address Translation

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

PAGE SIZE: WHY A POWER OF 2?

- Relatively easy to **implement a function in hardware** to perform dynamic **address translation at run time**
- Consider address of *n*+*m* bits, where the leftmost *n* bits are the page number and the rightmost *m* bits are the offset. Then:
  1. **Extract the page number** (leftmost *n* bits of the address)
  2. **Use page number as index** into the process page table to find frame number *k*
  3. **Physical address** is $k*2^m$ + offset (**this needs not be calculated; the bits of the frame number and the bits of the offset are just concatenated**)

PAGING: HARDWARE SUPPORT

- MMU has a **page table pointer register**: points to the beginning of the page table of the current process
- Page table structure is dictated by the MMU
- MMU does address translation at run time

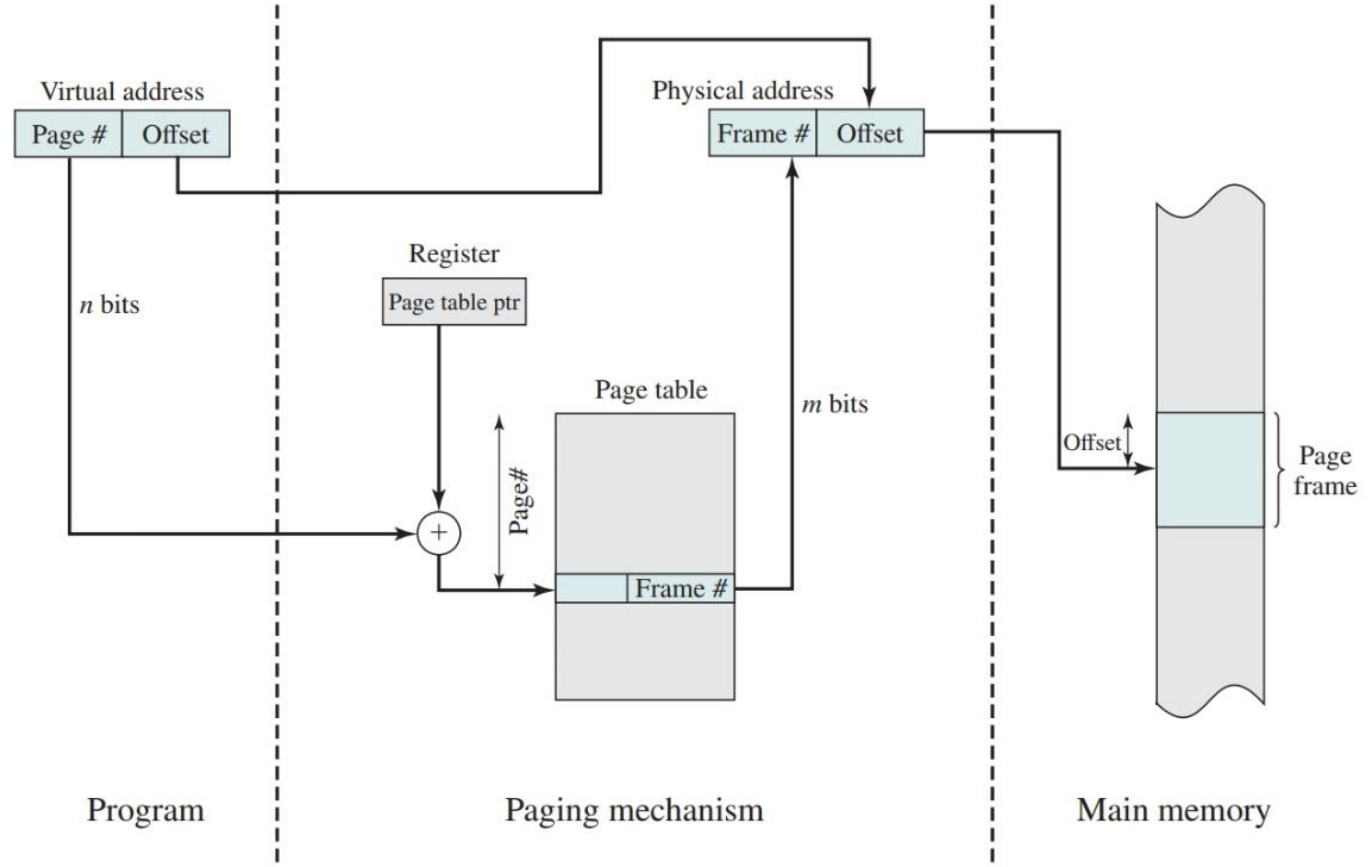


Figure 8.2 **Address Translation in a Paging System**

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

PAGING

- Allows processes to be relocated (swapped in and out of memory)
- Process image is divided into pages
  - Different permissions (execute-only, read-only) can be set for different pages
  - Allows sharing at page level
  - However, pages are not visible to the programmer!
- Pages and frames have a fixed size
  - Easy to manage free space
  - No **external fragmentation**
  - **Internal fragmentation:** wasted space due to a page not being fully used
- Provides protection from other processes: Each page is only visible to the process to which it belongs to

PAGE SIZE

- The smaller the page size, the less internal fragmentation
- However...
  - … more pages required per process
  - … more pages per process means larger page tables
  - … physical characteristics of most secondary-memory devices favor large page size for more **efficient block transfer of data**
- Typically, page size is 0.5 KB to 8 KB

HOW BIG IS A PAGE TABLE?

- Consider 32-bit virtual addresses with 4 KB pages
  - 20-bit virtual page numbers and 12-bit offsets
- OS needs a page table entry for each possible virtual page
  - Assuming 4 bytes per entry, the page table would take 2^{20}*4 B = 4 MB
  - Assuming system has 100 processes, this means 400 MB for page tables only!
- What about 64-bit virtual addresses with 4 KB pages?
  - 52-bit virtual page numbers and 12-bit offsets
  - Assuming 8 bytes per entry, the page table would take 2^{52}*8 B = 32768 TB

TWO-LEVEL HIERARCHICAL PAGE TABLE

- In practice, even 5-level page tables exist in today's systems

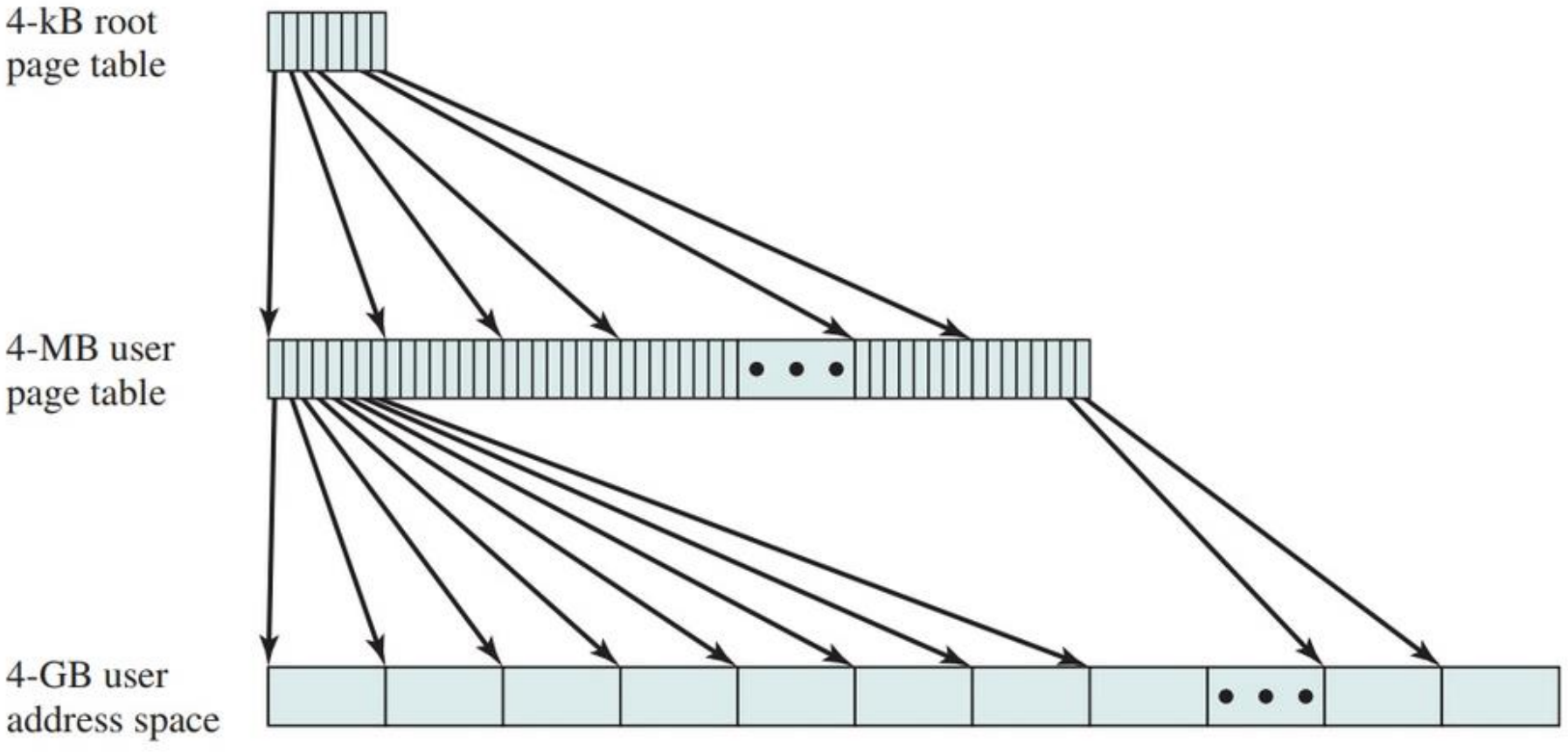


Figure 8.3 A Two-Level Hierarchical Page Table

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

TWO-LEVEL HIERARCHICAL PAGE TABLE: ADDRESS TRANSLATION

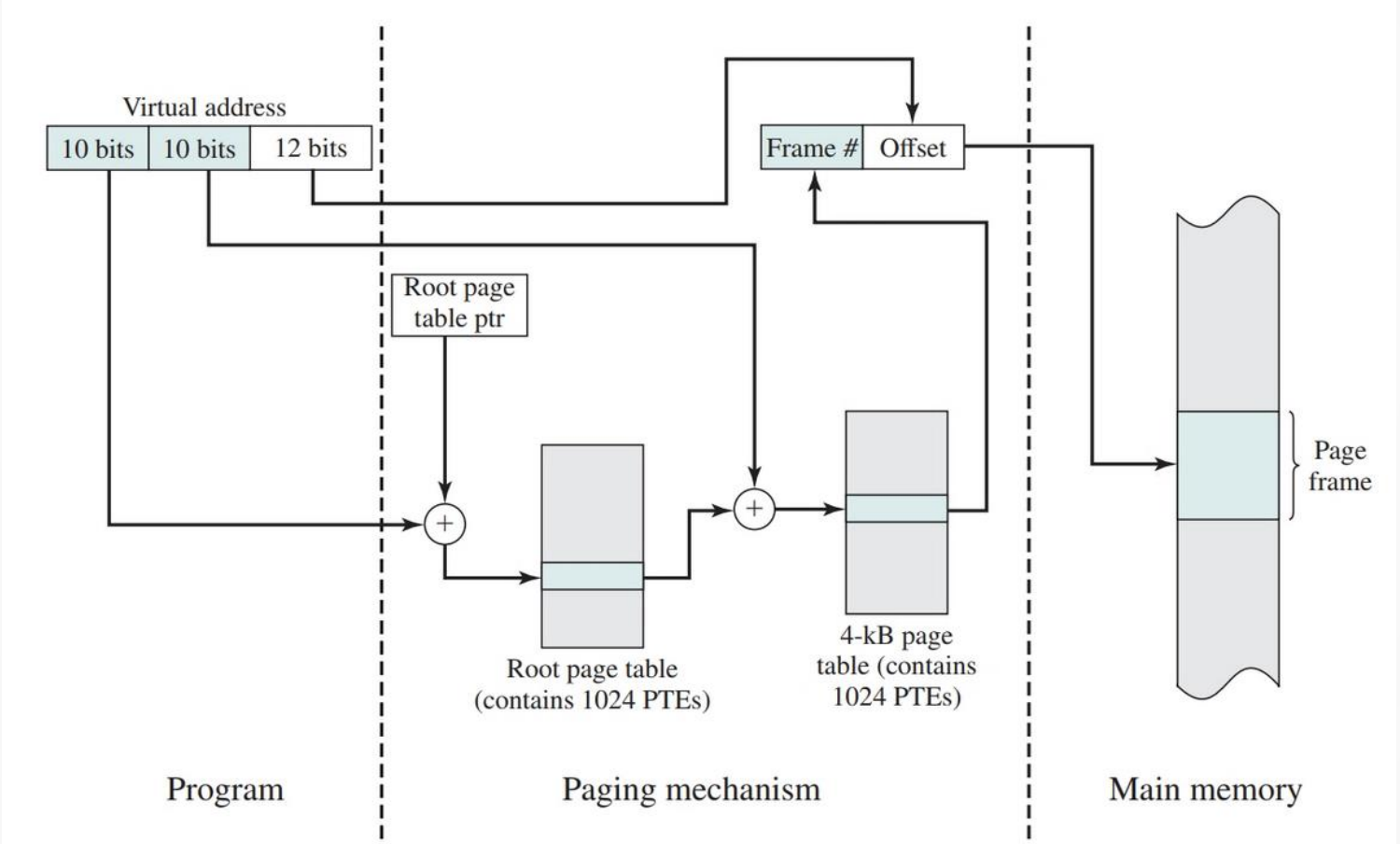


Figure 8.4 Address Translation in a Two-Level Paging System

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

TWO-LEVEL HIERARCHICAL PAGE TABLE: ADDRESS TRANSLATION EXAMPLE

- Virtual addess:
  0000 0001 1000 0000 1000 0101 0111 0010

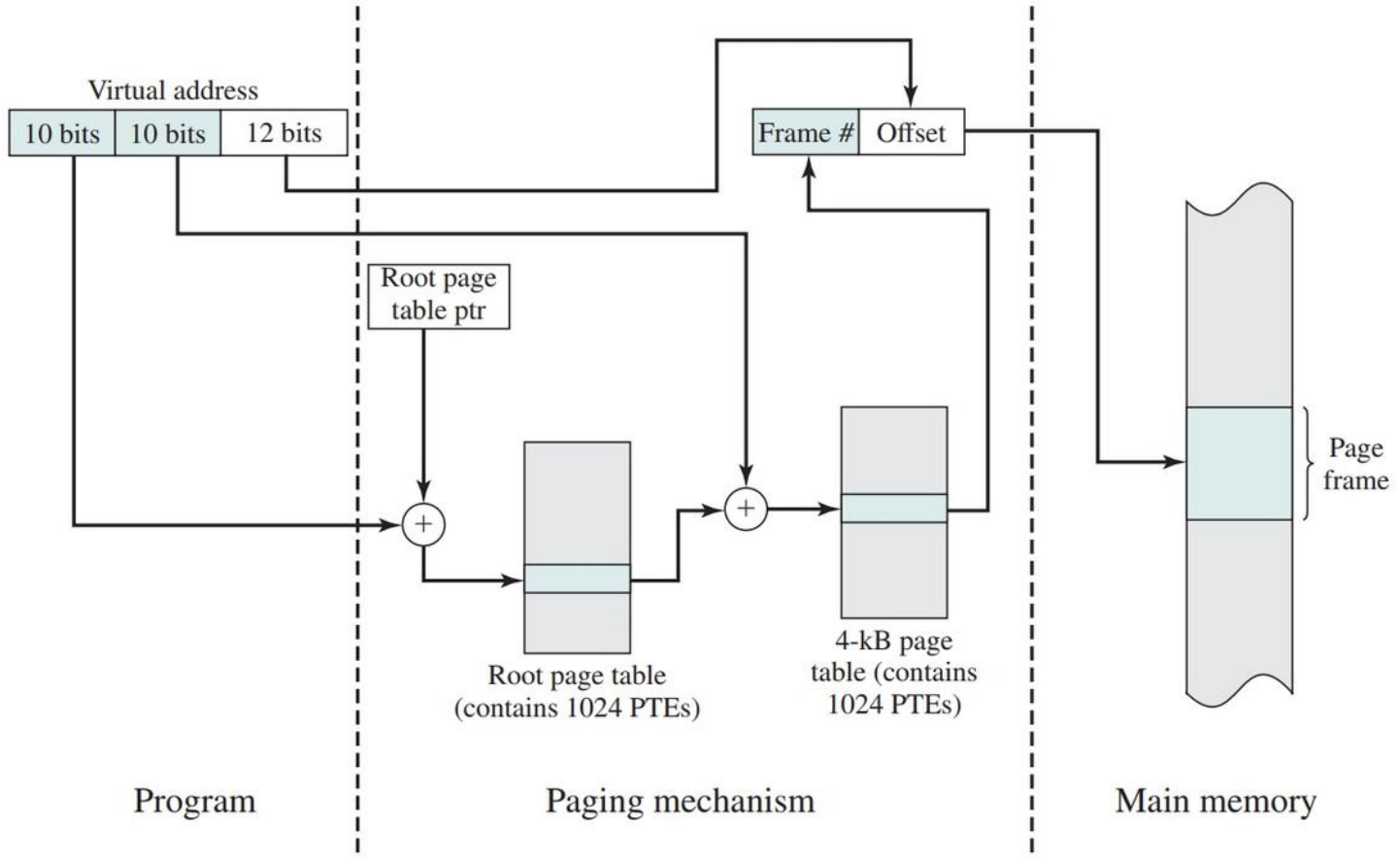


Figure 8.4 **Address Translation in a Two-Level Paging System**

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

TWO-LEVEL HIERARCHICAL PAGE TABLE: ADDRESS TRANSLATION EXAMPLE

- Virtual addess:
  0000 0001 1000 0000 1000 0101 0111 0010
  - Index to root page table: 00 0000 0110 = 6
  - Index to 4-kB page table: 00 0000 1000 = 8
  - Offset: 0101 0111 0010
- 6th entry in root page table gives the address of the 4-kB page table
- 8th entry in 4-kB page table gives the frame # (assume it is 3 = 0x0000 0000 0000 0011)
- Physical address:
  0x0000 0000 0000 0011 0101 0111 0010

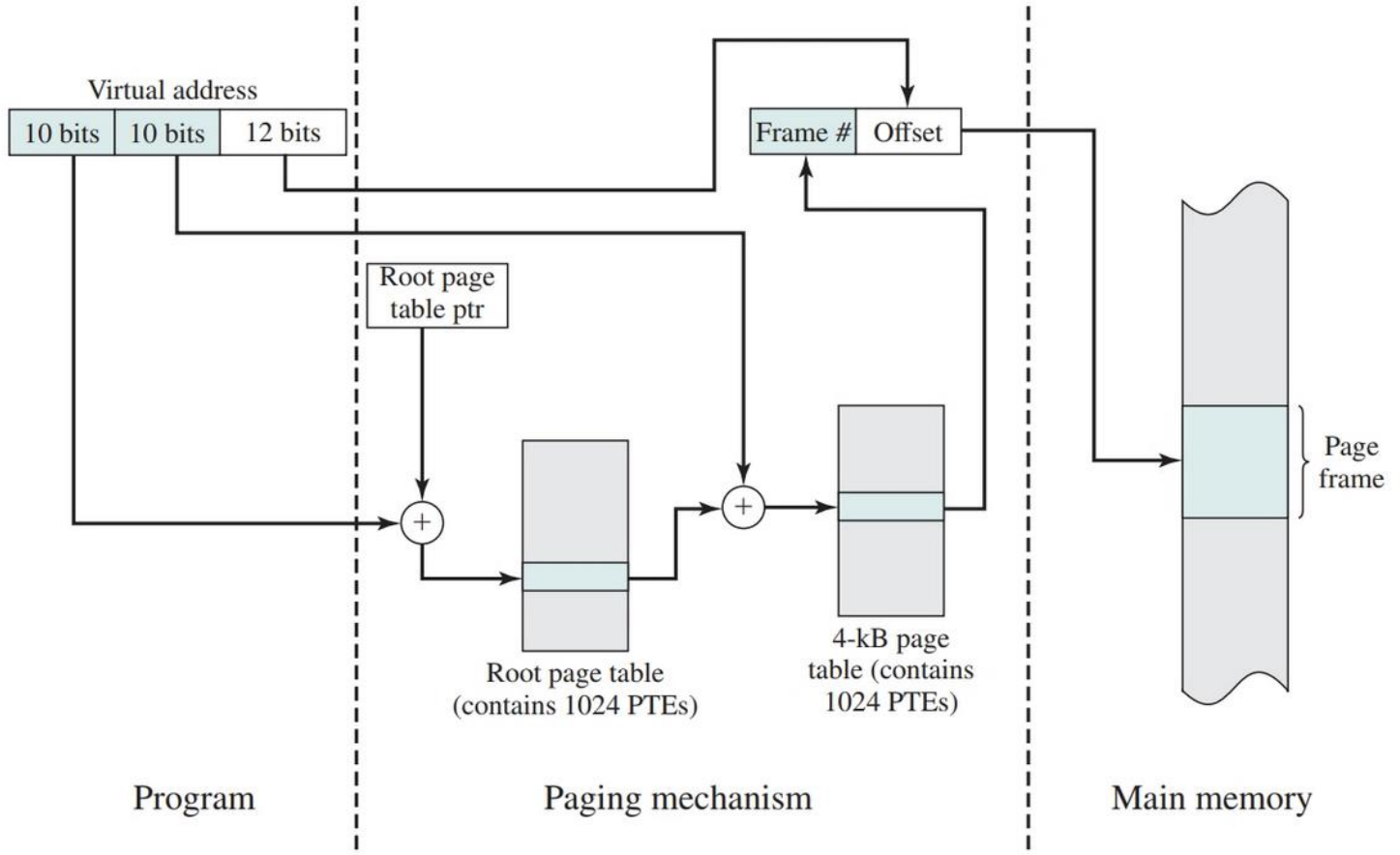


Figure 8.4 **Address Translation in a Two-Level Paging System**

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

ALTERNATIVE APPROACH: INVERTED PAGE TABLE

- Instead of having a page table entry of each possible virtual page, implement a hash table mapping the virtual pages in use to physical frames
  - Virtual page number is mapped to a hash value
  - The hash value is used to index the inverted page table
- Fixed proportion of real memory is needed for page tables regardless of the number of processes or virtual page numbers supported
  - There is one entry in the inverted page table for each real memory frame, rather than one per virtual page
  - More than one virtual page may map into the same hash value; A chaining technique used for managing the overflow

INVERTED PAGE TABLE STRUCTURE

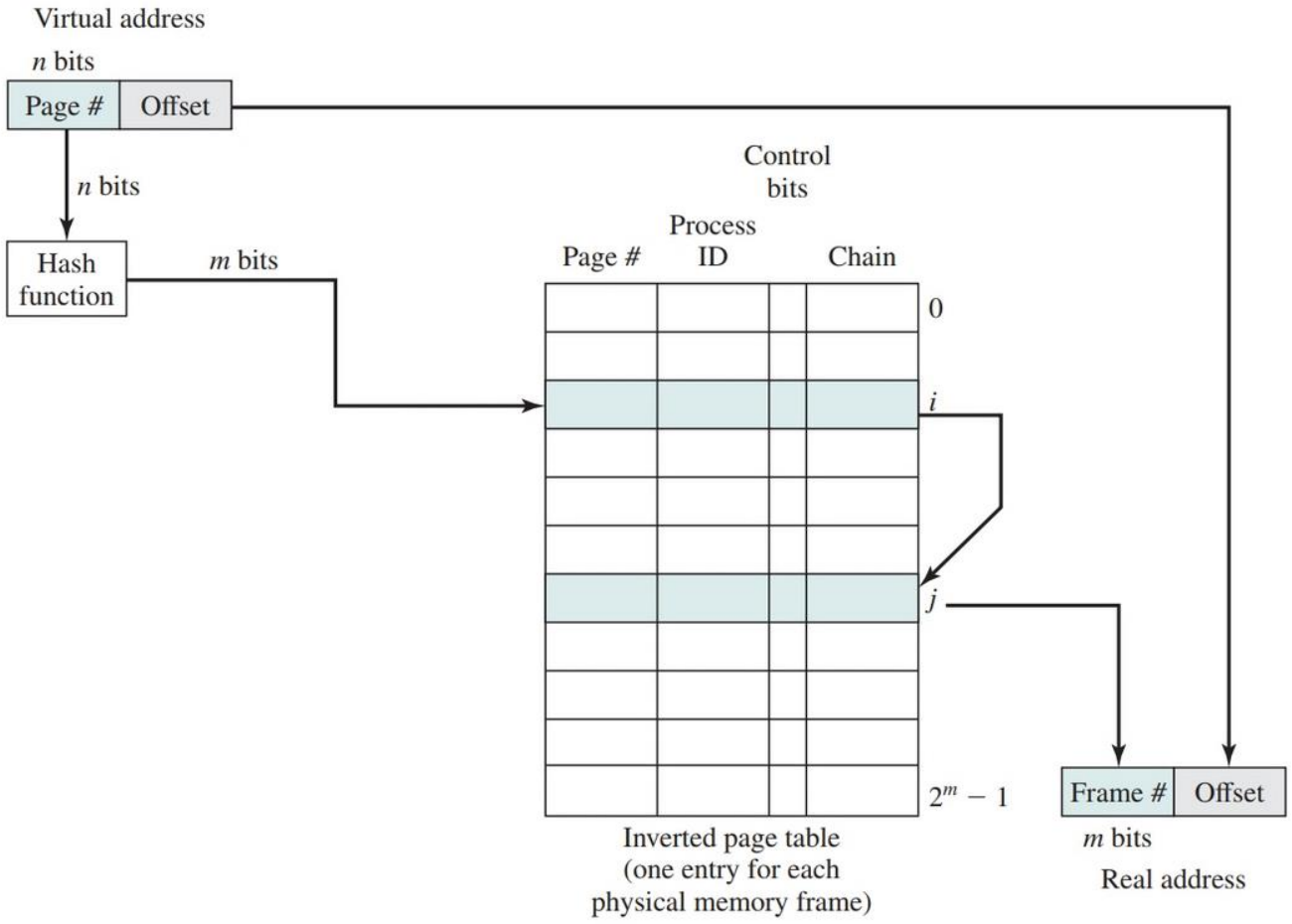


Figure 8.5 **Inverted Page Table Structure**

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

INVERTED PAGE TABLE: EXAMPLE

- Virtual addess (n=8):
  0000 0111 0000 0001 1001

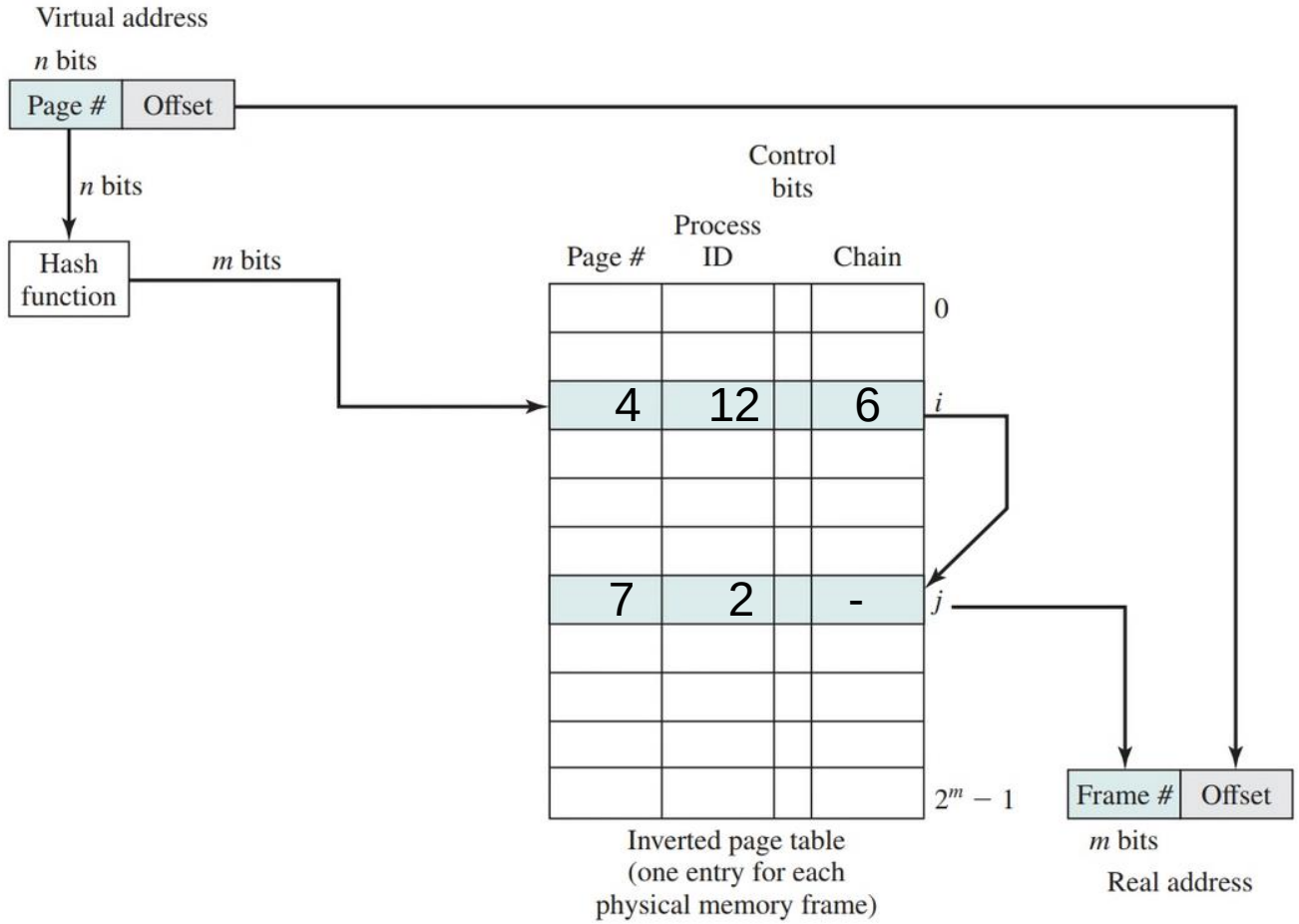


Figure 8.5 **Inverted Page Table Structure**

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

INVERTED PAGE TABLE: EXAMPLE

- Virtual addess (n=8):
  0000 0111 0000 0001 1001
  - Page #: 0000 0111 = 7
  - Offset: 0000 0001 1001
- Hash(Page #) = Hash(7) = 2
  - This entry contains a different page -> follow chain to find the correct entry
  - Correct entry found on row 6 -> page stored in frame 6
- Virtual address (6 = 0000 0110):
  0000 0110 0000 0001 1001

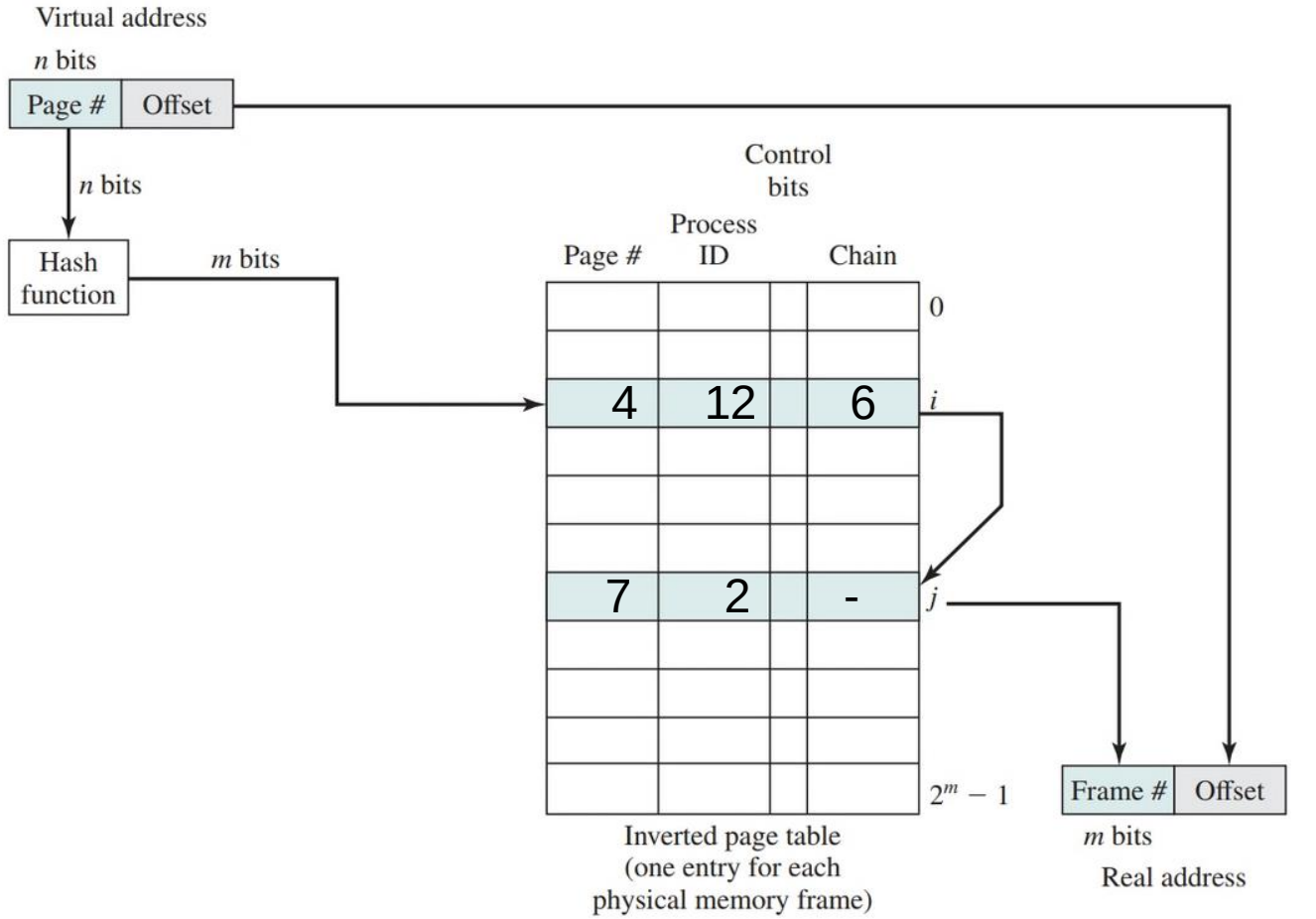


Figure 8.5 **Inverted Page Table Structure**

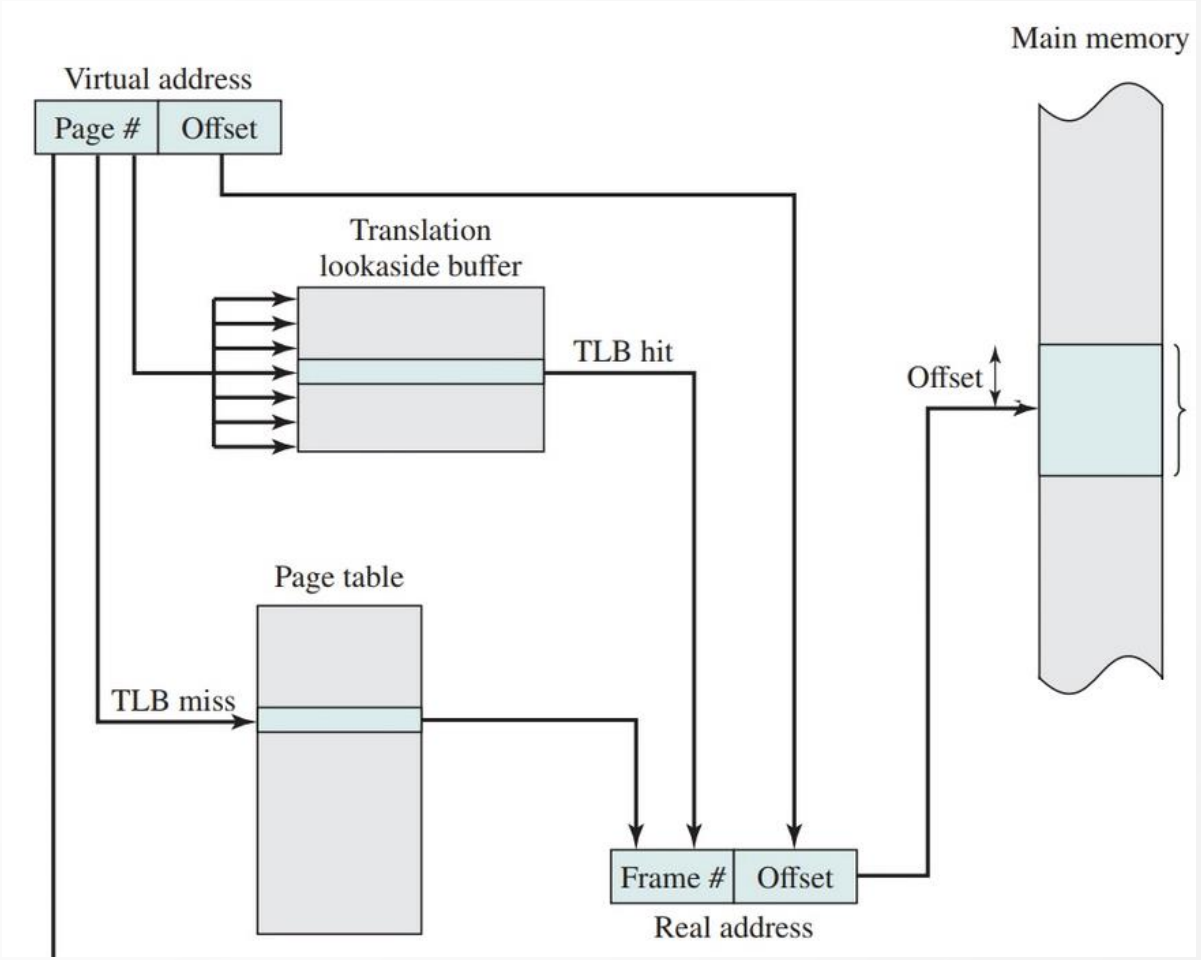
Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

TRANSLATION LOOKASIDE BUFFER (TLB)

- Every virtual memory reference can cause **two physical memory accesses**
  - One to fetch the page table entry
  - Another to fetch the actual data
- To overcome the effect of doubling the memory access time, most systems employ a **special high-speed cache** called a **translation lookaside buffer (TLB)**
  - Hardware assistance!
  - Contains the page table entries that have been most recently used

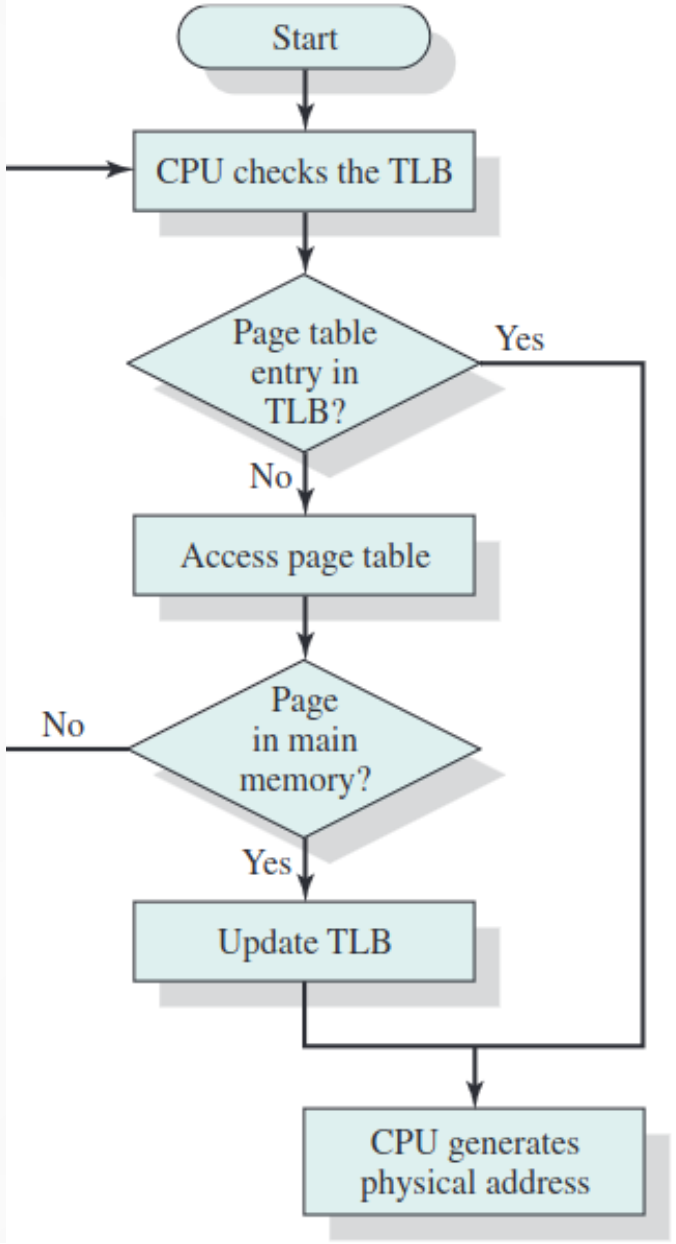
TRANSLATION LOOKASIDE BUFFER (TLB)

- TLB only contains some of the page table entries
  - We cannot simply index the TLB with page numbers
  - Each TLB entry must include the **page number** as well as the **complete page table entry**
- **Associative mapping**: Processor is equipped with hardware that allows to interrogate **simultaneously multiple TLB entries** to determine if there is a match on the page number



(Part of) Figure 8.6 [Stallings, Operating systems: Internals and design principles, 9th ed]

ADDRESS TRANSLATION WITH PAGING AND TLB



(Part of) Figure 8.7 [Stallings, Operating systems: Internals and design principles, 9th ed]

COMBINED SEGMENTATION AND PAGING

- In combined paging / segmentation system, the u**ser process address space is broken up into several segments**, and each segment is further broken up into several fixed-size pages (same size as frames in main memory)
- Segmentation is visible to the programmer
- Paging is transparent to the programmer
- From programmer's point of view: A virtual address consists of a segment number and segment offset
- From system's point of view: the segment offset is viewed as a page number and a page offset for a page specified within that segment

COMBINED SEGMENTATION AND PAGING

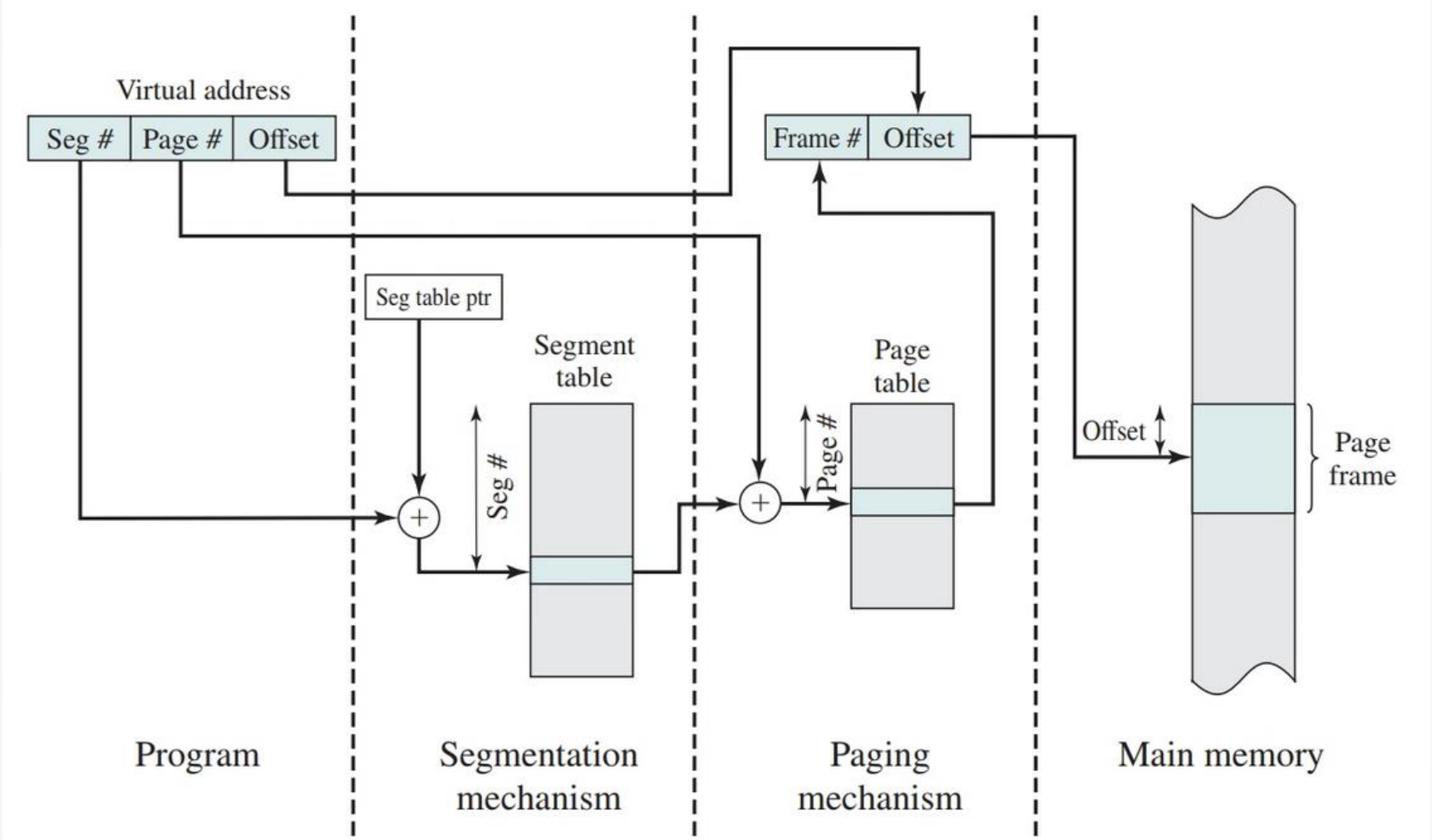


Figure 8.12 Address Translation in a Segmentation/Paging System

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

SEGMENTATION AND PAGING: EXAMPLE

- Virtual addess:
  1000 0110 1001 0010
  - Segment #: 10 = 2
  - Page #: 00 0110 = 6
  - Offset: 1001 0010
- 2nd entry in segment table gives the address of the page table
- 6th entry in the page table gives the frame number
  (assume this is 0001 0000 = 16)
- Physical address:
  0001 0000 1001 0010

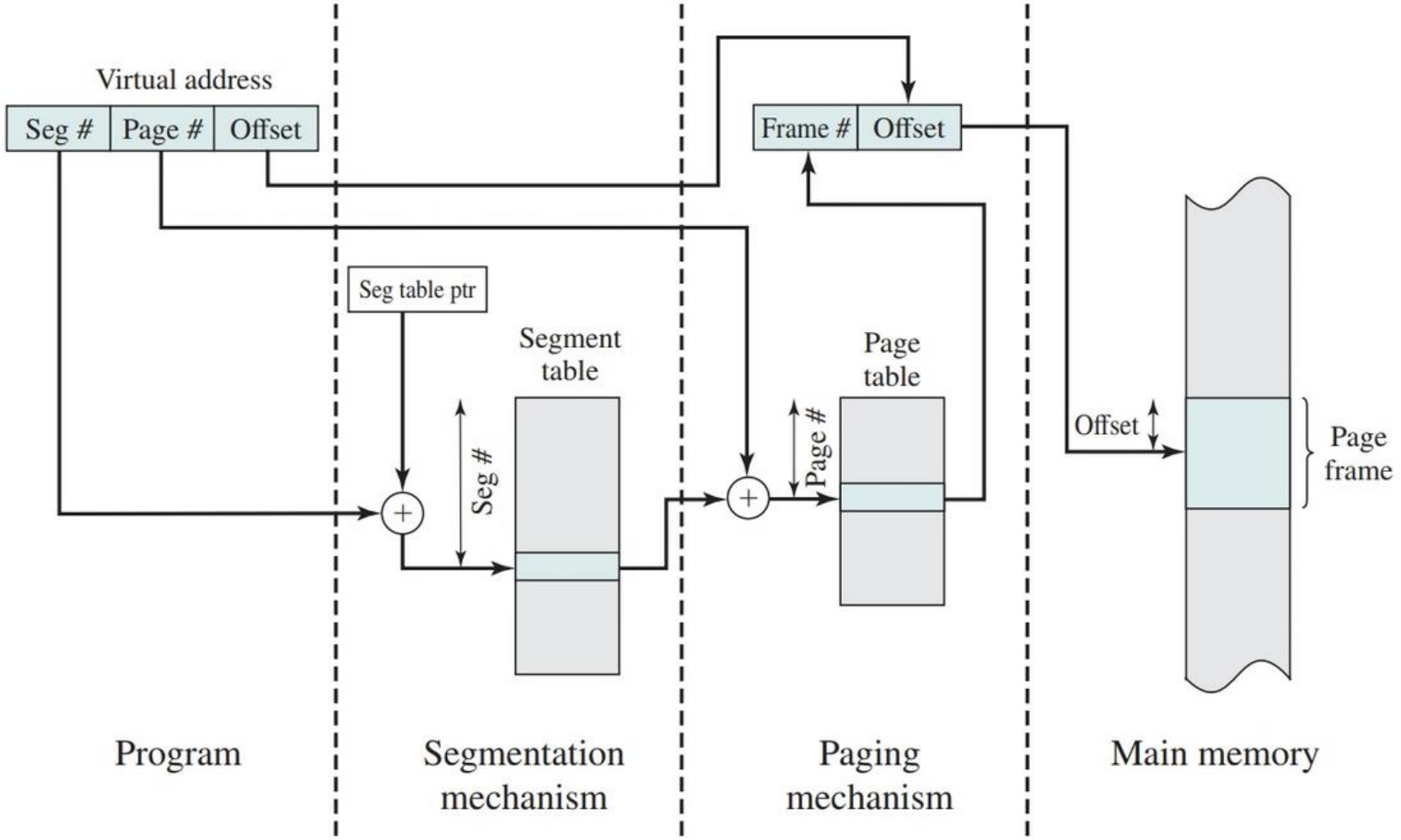


Figure 8.12 Address Translation in a Segmentation/Paging System

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

SUMMARY

- Memory management and virtual memory is one of the most important and complex tasks of the OS
- Memory is a resource that needs to be allocated to and shared among processes
- Memory management needs to allow **relocation** of processes, be **efficient**, and allow **protection and sharing** among processes
- Basic tools: Paging and segmentation (can also be combined!)
  - **Paging**: Small fixed-size pages
  - **Segmentation**: Pieces of variable size
- Implementation requires hardware assistance (e.g. TLB, address translation) and software support (e.g. segment tables, page tables)