CSE 120: Principles of Operating Systems
Lecture 9: Logical Memory

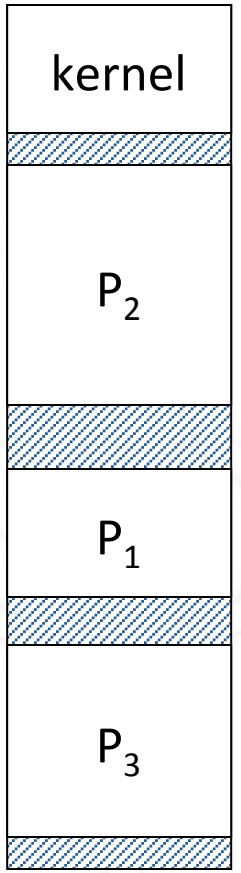
Prof. Joseph Pasquale
University of California, San Diego
February 27, 2023

What is Logical Memory

- Logical memory = a process's memory
- As viewed (referenced) by a process
- Allocated without regard to physical memory

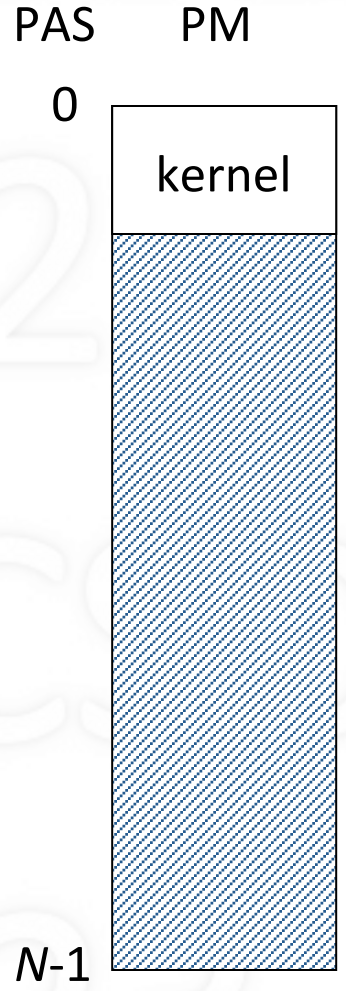
Problems with Sharing Memory

- The Addressing Problem
  - Compiler generates memory references
  - Unknown where process will be located
- The Protection Problem
  - Modifying another process's memory
- The Space Problem
  - The more processes there are, the less memory each individually can have



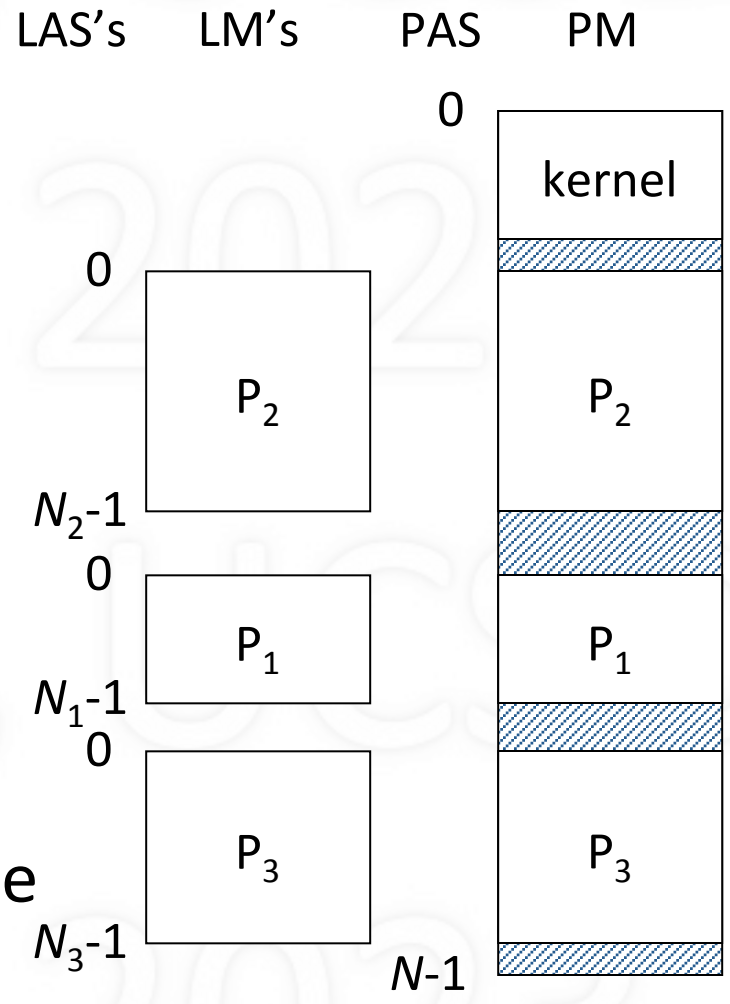
Address Spaces

- Address space
  - Set of addresses for memory
- Usually linear: 0 to *N*-1 (size *N*)
- Physical Address Space
  - 0 to *N*-1, *N* = size
  - Kernel occupies lowest addresses (typically)



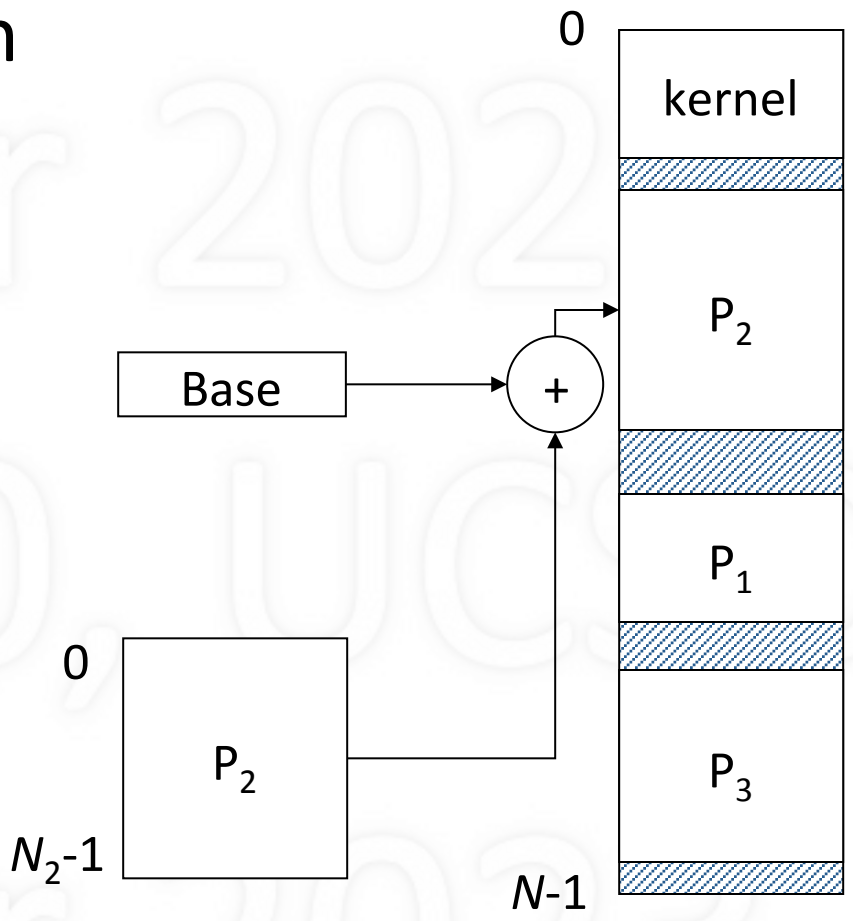
Logical vs. Physical Addressing

- Logical addresses
  - Assumes separate memory starting at 0
  - Compiler generated
  - Independent of location in physical memory
- Convert logical to physical
  - Via software: at load time
  - Via hardware: at access time



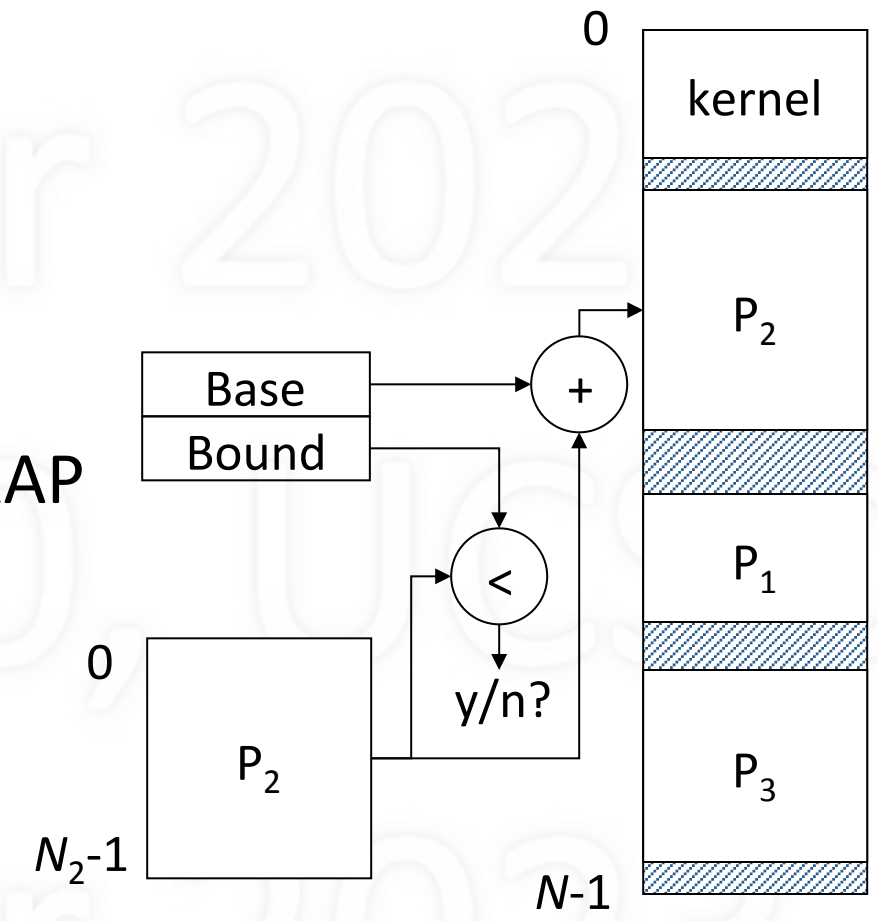
Hardware for Logical Addressing

- Base register filled with start address
- To translate logical address, add base
- Achieves relocation
- To move process: change base



Protection

- Bound register works with base register
- Is address < bound
  - Yes: add to base
  - No: invalid address, TRAP
- Achieves protection

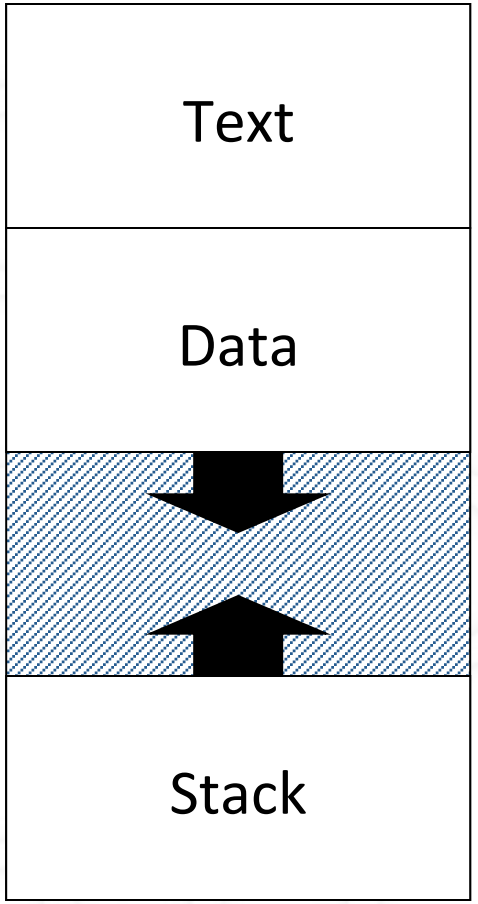


Memory Registers Part of Context

- On Every Context Switch
  - Load base/bound registers for selected process
  - Only kernel does loading of these registers
  - Kernel must be protected from all processes
- Benefit
  - Allows each process to be separately located
  - Protects each process from all others

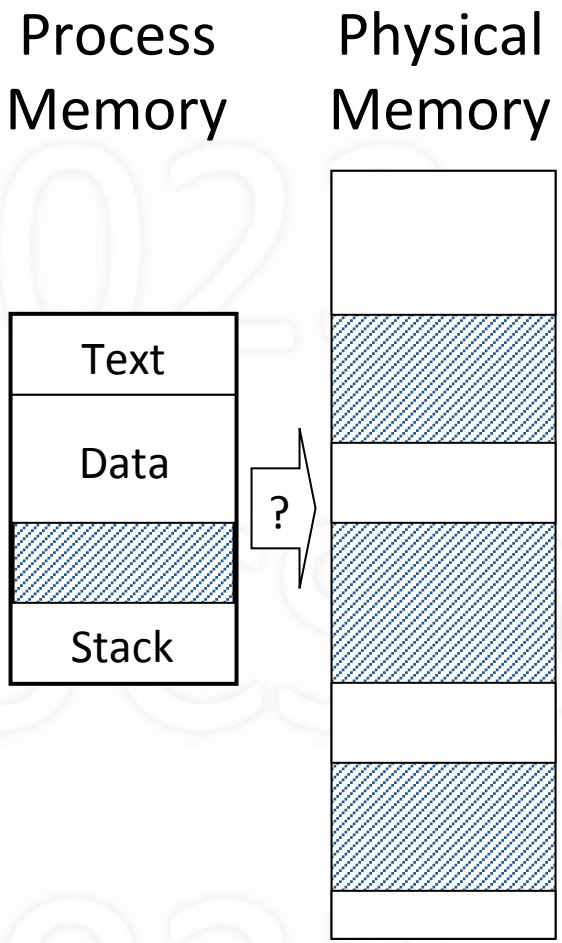
Recall Process Address Space

- Text: program instructions
  - Execute-only, fixed size
- Data: variables (static, heap)
  - Read/write, variable size
  - Dynamic allocation by request
- Stack: activation records (auto)
  - Read/write, variable size
  - Automatic growth/shrinkage



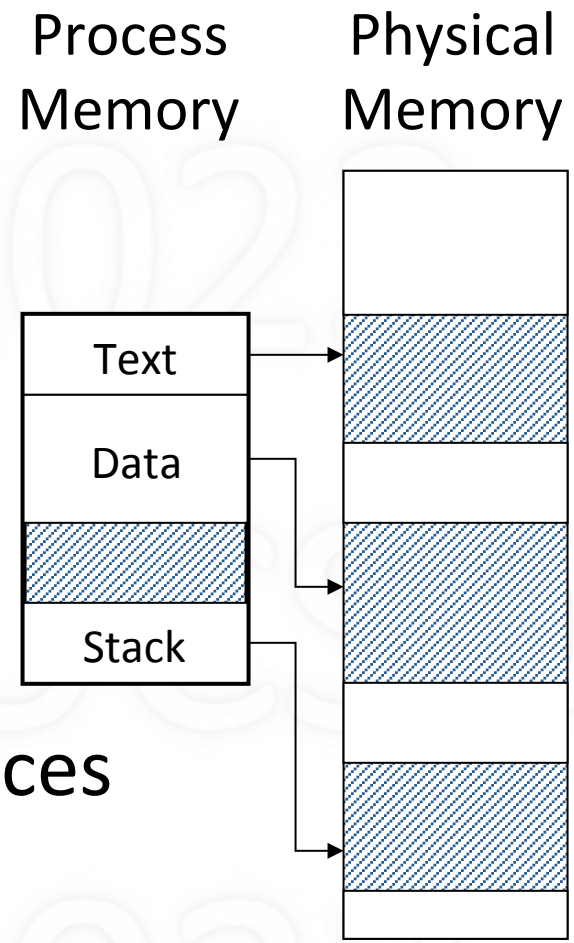
Fitting Process into Memory

- Must find large enough hole
- May not succeed
  - even if enough fragment space
- Even if successful, inefficient
  - Space must be allocated for potential growth areas



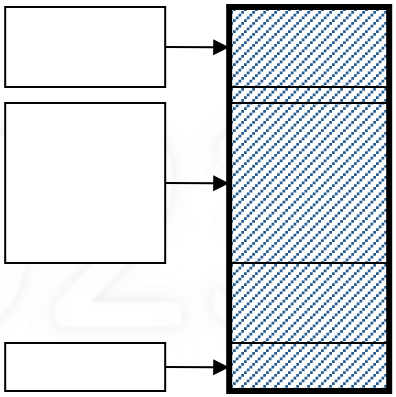
Fitting Process into Memory

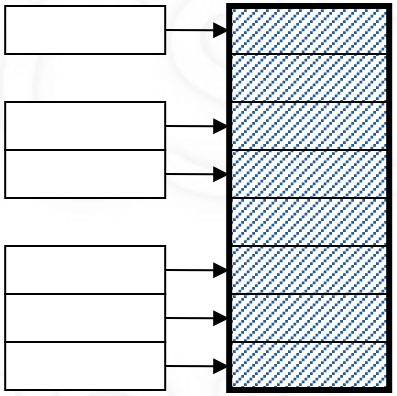
- Must find large enough hole
- May not succeed
  - even if enough fragment space
- Even if successful, inefficient
  - Space must be allocated for potential growth areas
- Solution: break process into pieces
  - Distribute into available holes



Two Approaches

- Segmented address space
  - Partition into segments
  - Segments can be different sizes
- Paged address space
  - Partition into pages
    - Corresponding memory: frames
  - Pages are the same size

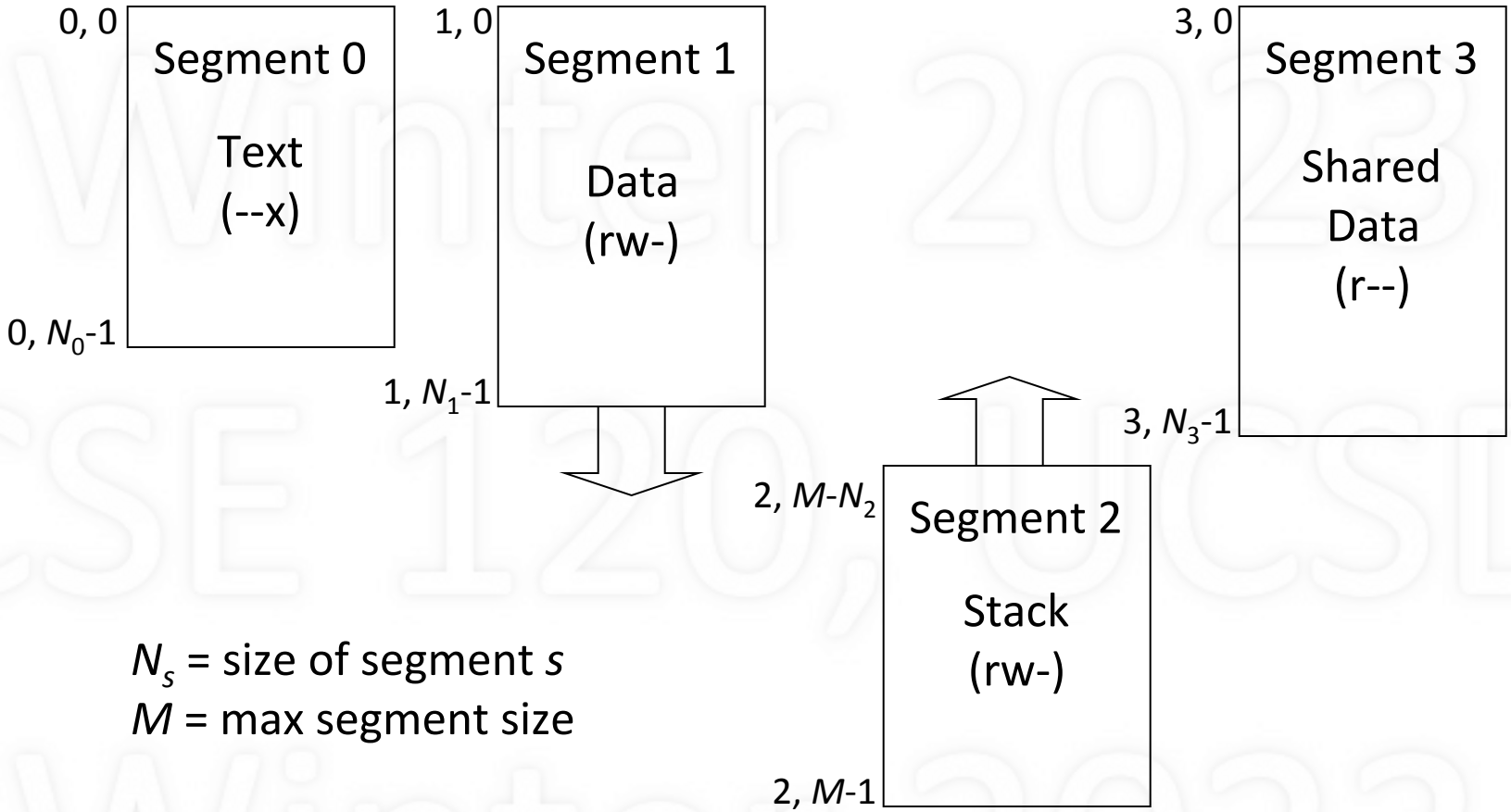




Segmented Address Space

- Address space is a set of segments
- Segment: a linearly addressed memory
  - Typically contains logically-related information
  - Examples: program code, data, stack
- Each segment has an identifier *s*, and a size N_s
  - *s* between 0 and *S*-1, *S* = max number of segments
- Logical addresses are of the form (*s*, *i*)
  - Offset *i* within segment *s*, *i* must be less than N_s

Example: Segmented Address Space



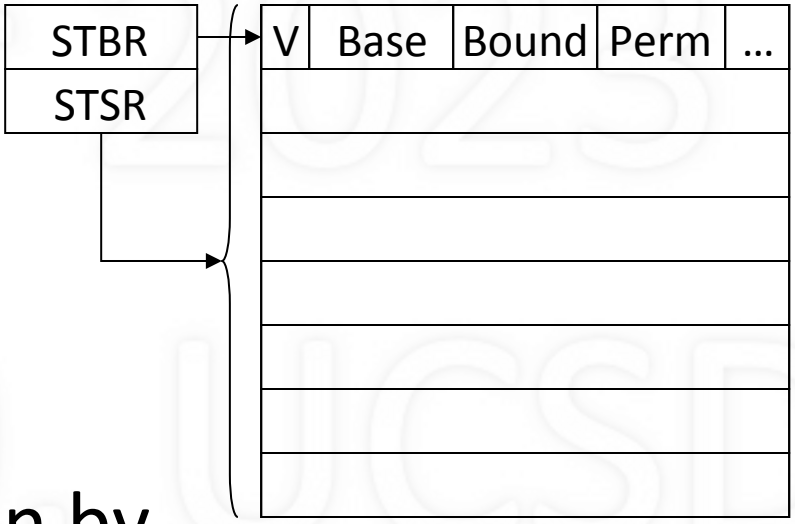
N_s = size of segment s
M = max segment size

Segment-based Address Translation

- Problem: how to translate
  - a logical address (*s*, *i*) into physical address *a*?
- Solution: use a segment (translation) table *ST*
  - to segment *s* into base physical address *b* = *ST* (*s*)
  - then add *b* and *i*
- Summary: physical address = *ST* (*s*) + *i*

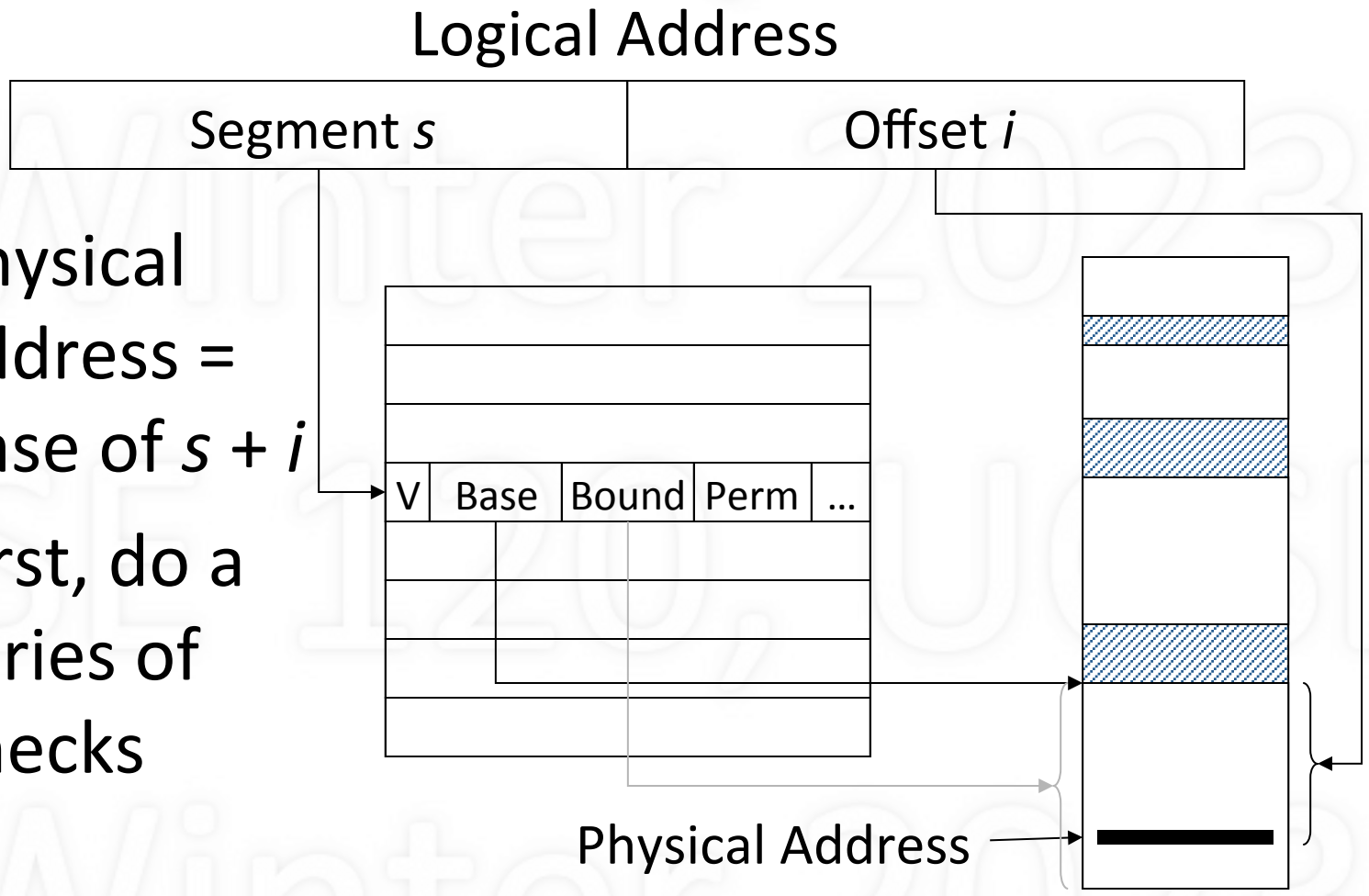
Segment Table

- One per process (typically)
- Table entry elements
  - V: valid bit
  - Base: segment location
  - Bound: segment size
  - Perm: permissions
- Location in memory given by
  - Segment table base register (hardware)
  - Segment table size register (hardware)

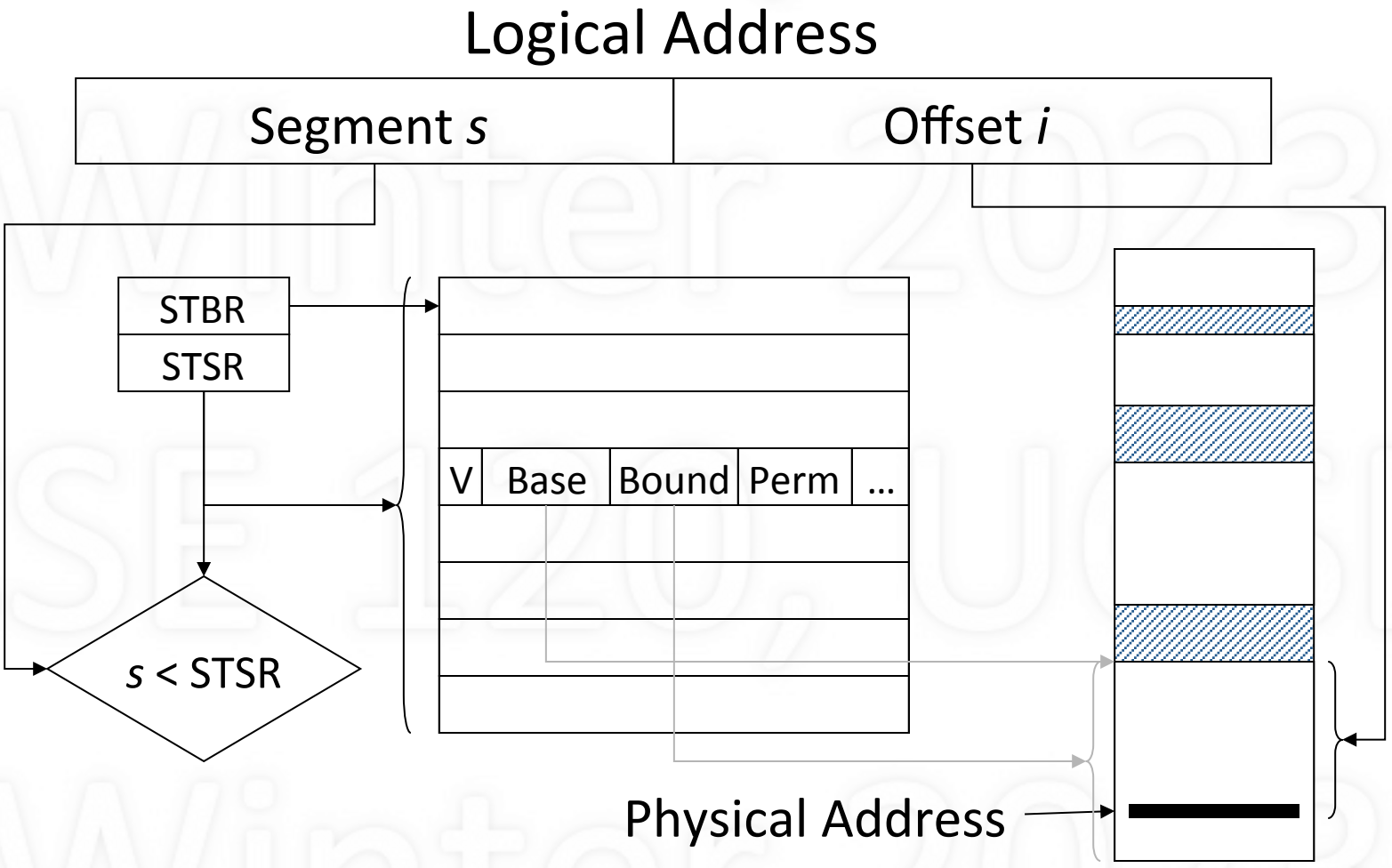


Address Translation

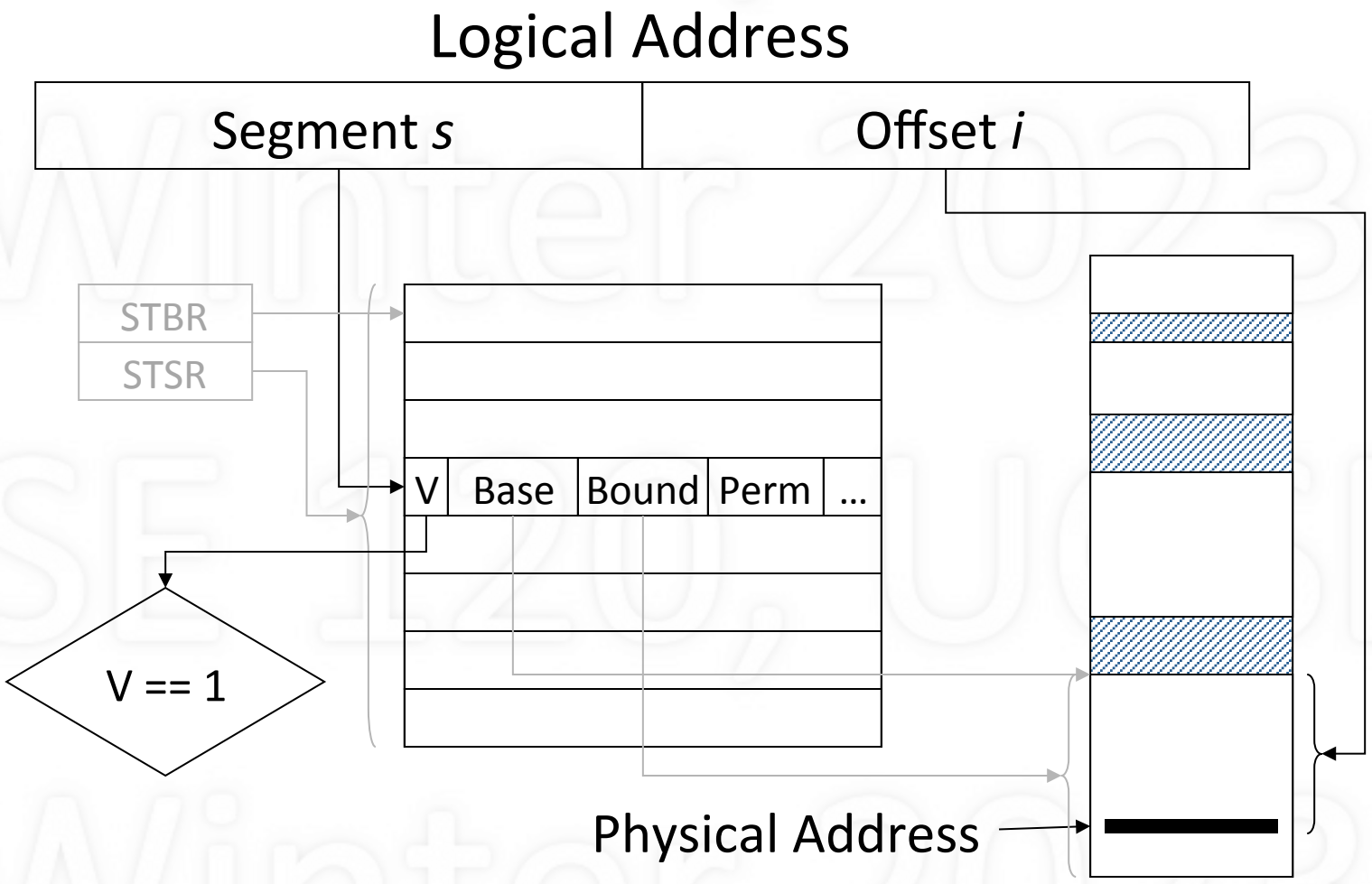
- Physical address = base of *s* + *i*
- First, do a series of checks



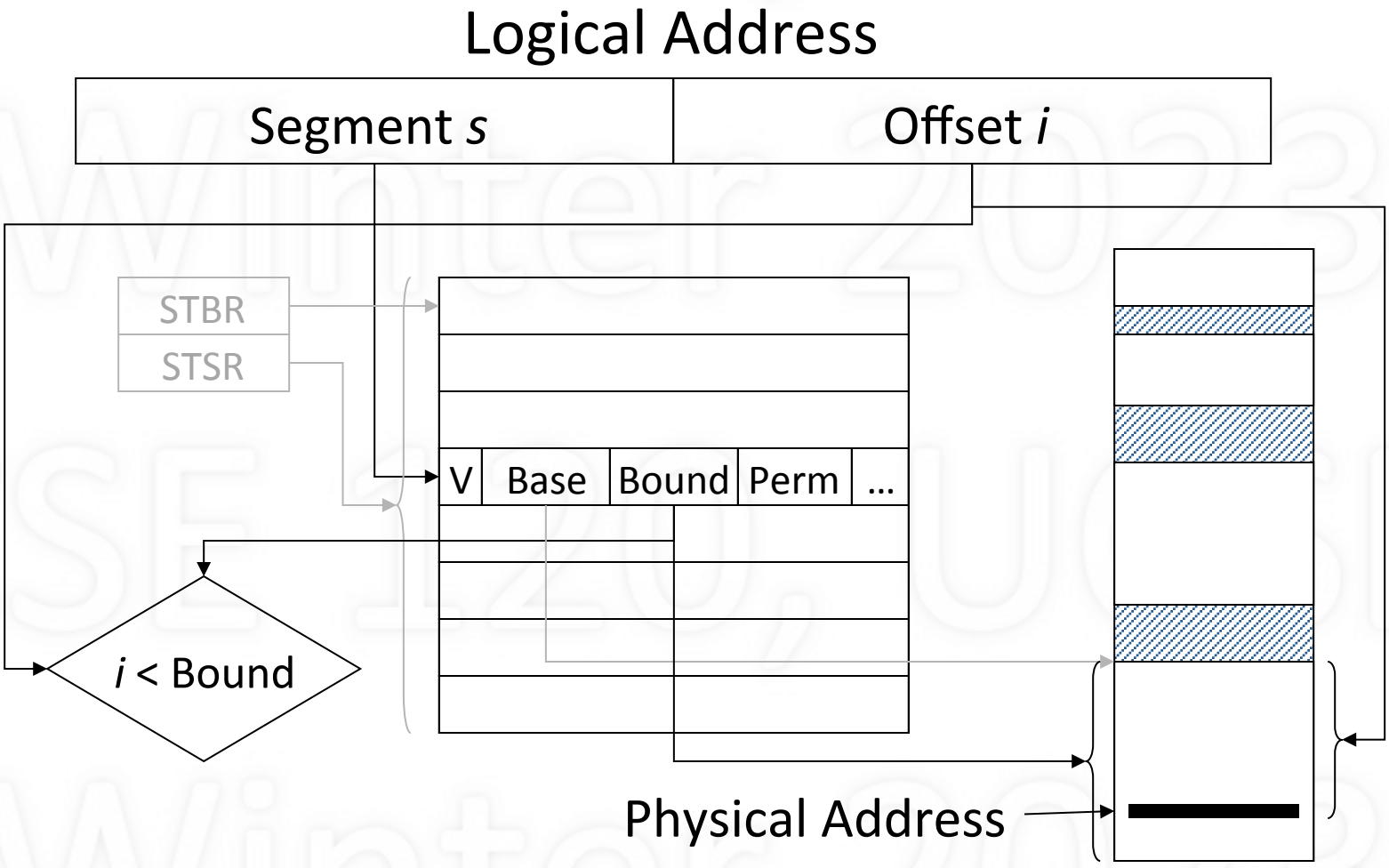
Check if Segment *s* is within Range



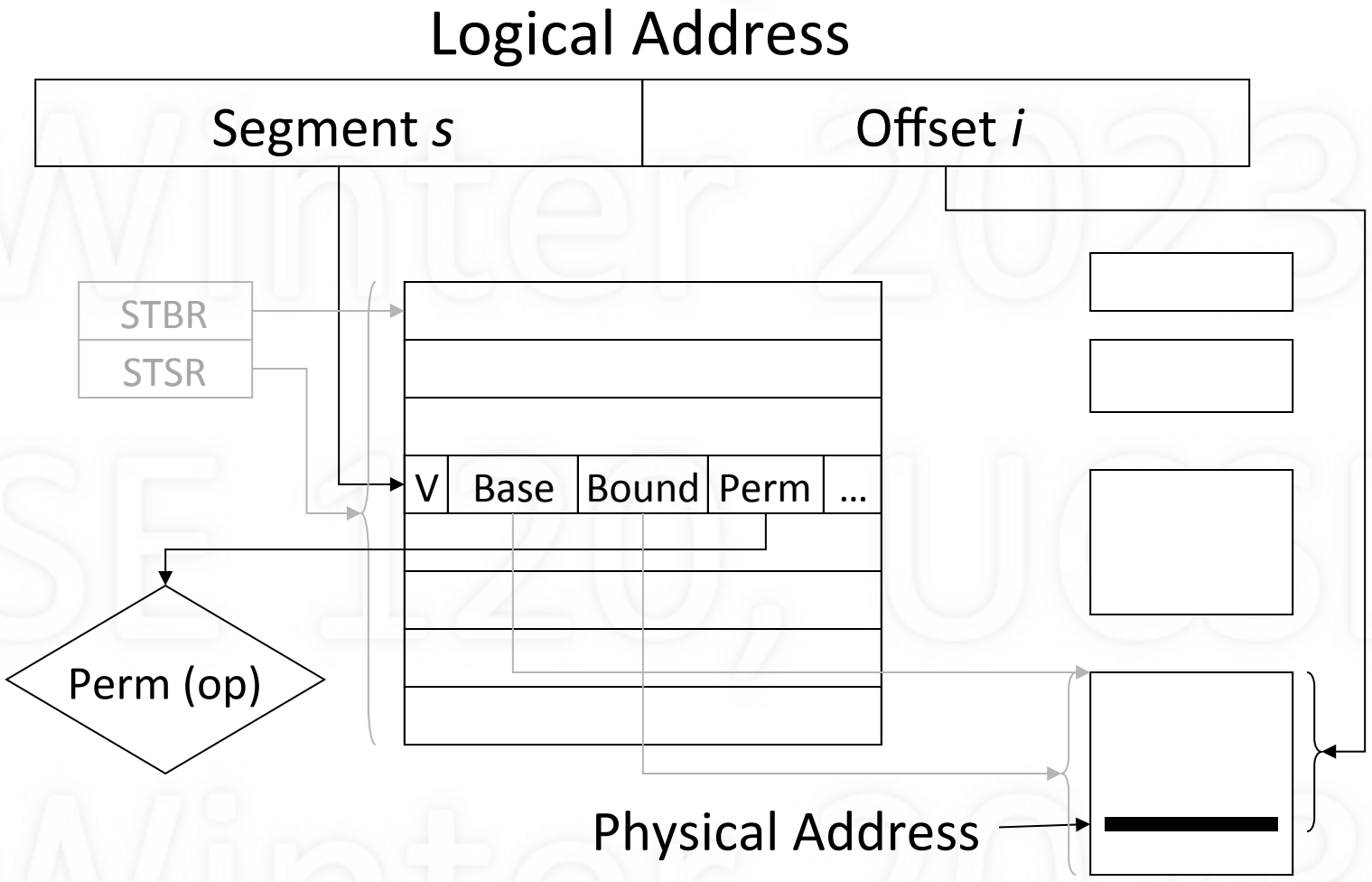
Check if Segment Entry *s* is Valid



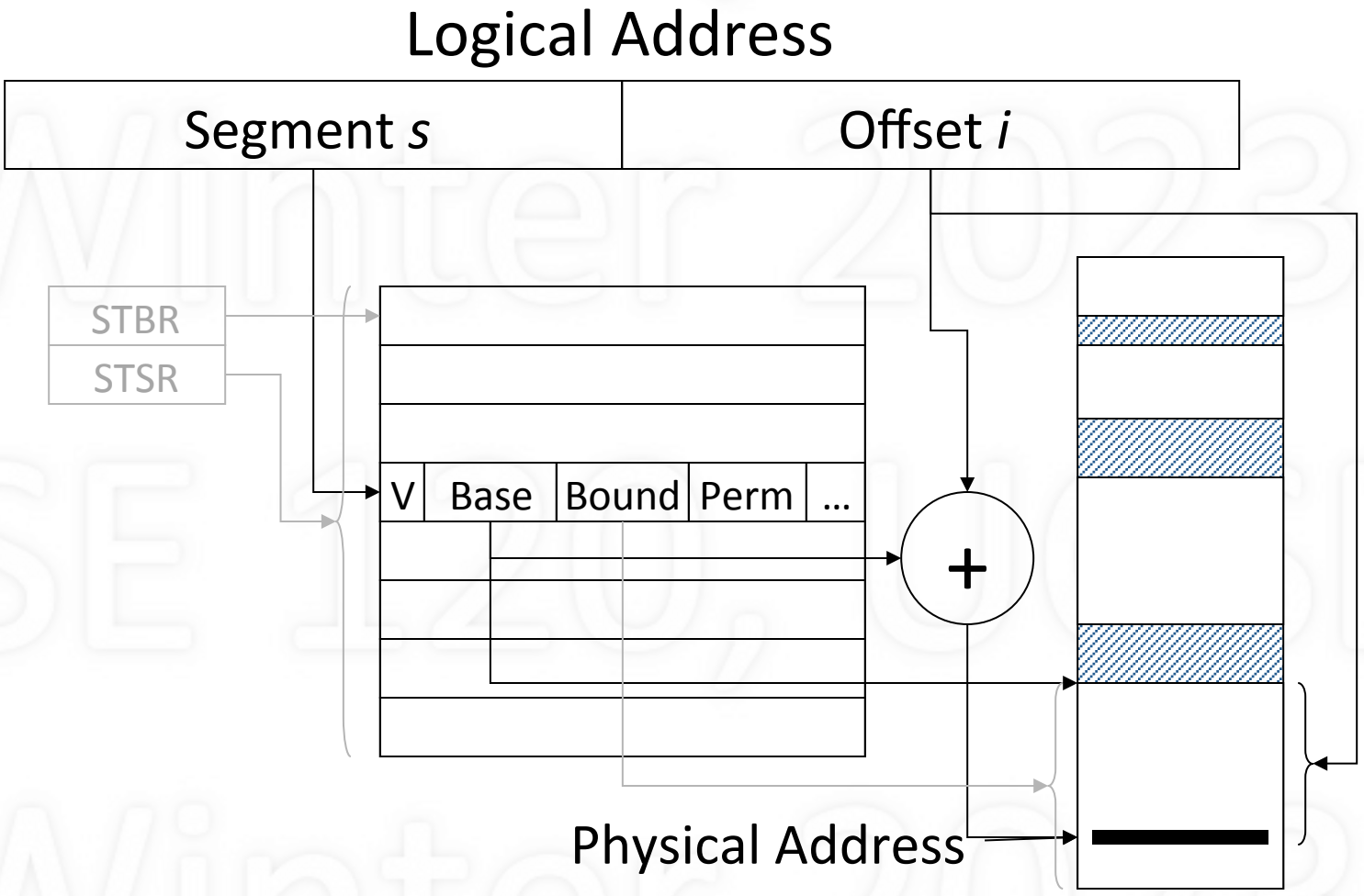
Check if Offset *i* is within Bounds



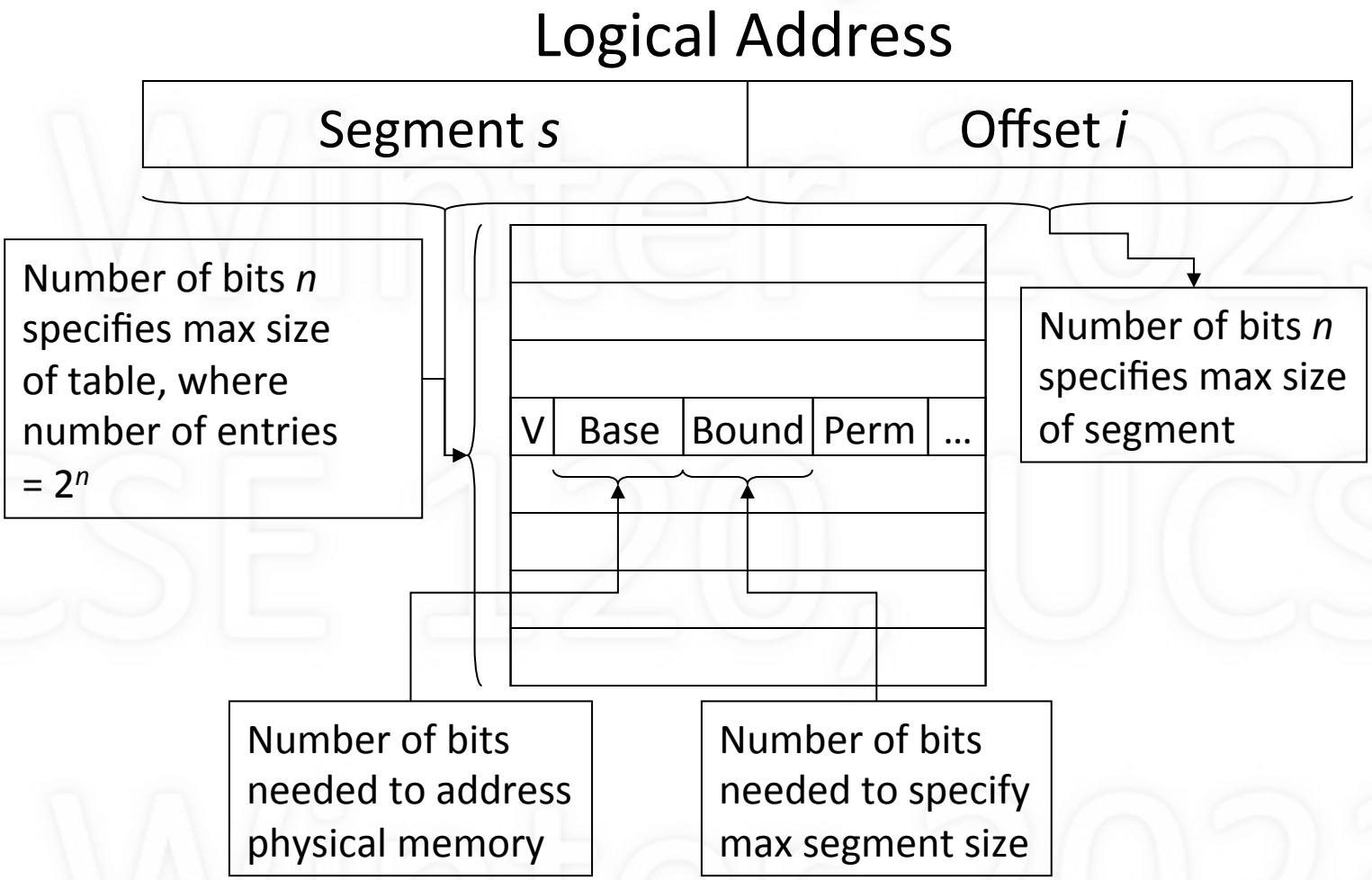
Check if Operation is Permitted



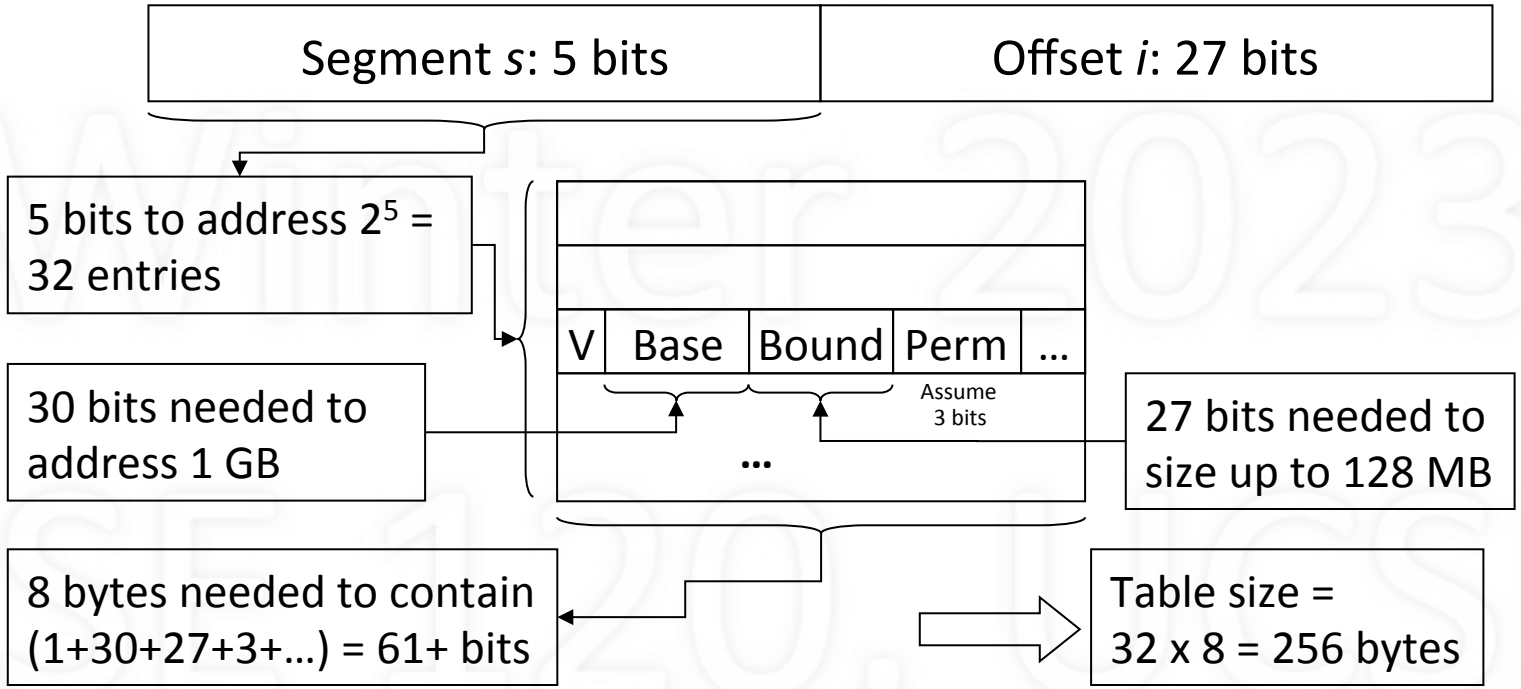
Translate Address



Sizing the Segment Table



Example of Sizing the Segment Table



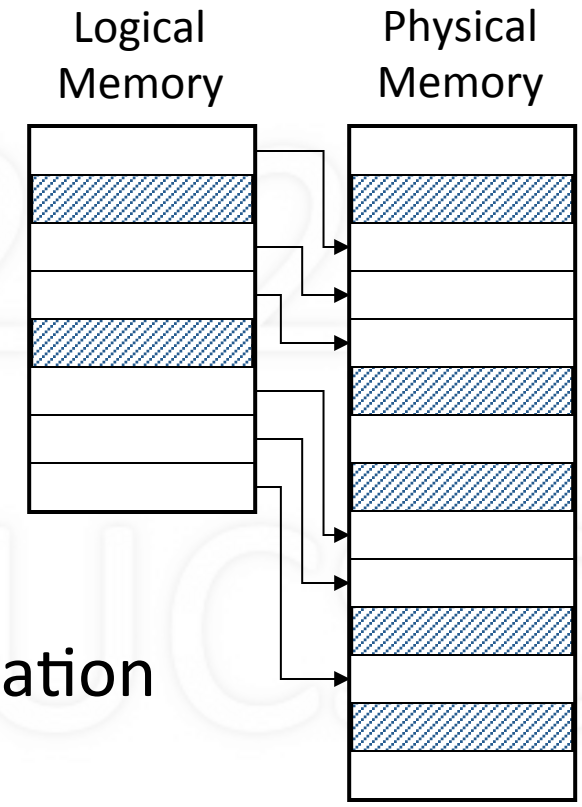
- Given 32 bit logical, 1 GB physical memory (max)
  - 5 bit segment number, 27 bit offset

Pros and Cons of Segmentation

- Pro: Each segment can be
  - located independently
  - separately protected
  - grown/shrunk independently
- Pro: Segments can be shared by processes
- Con: Variable-size allocation
  - Difficult to find holes in physical memory
  - External fragmentation

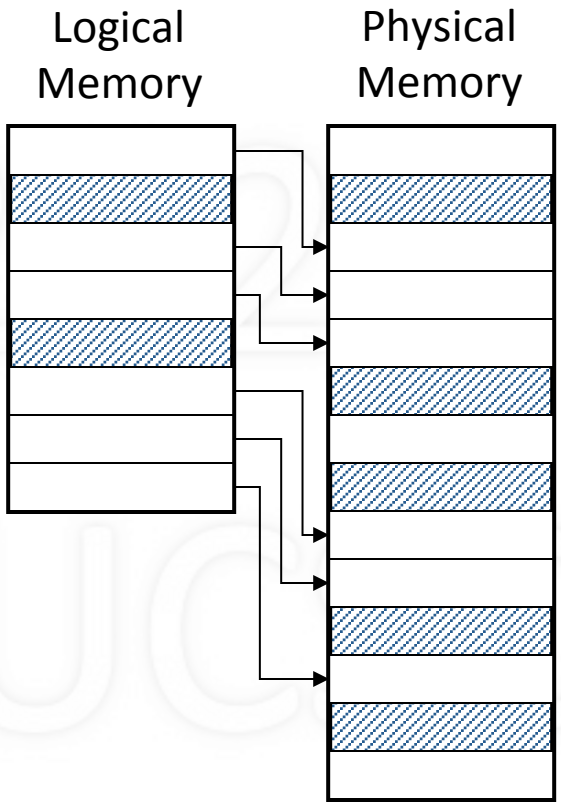
Paged Address Space

- Logical (process) memory
  - Linear sequence of pages
- Physical memory
  - Linear sequence of frames
- Pages and frames
  - Frame: a physical unit of information
  - A page fits exactly into a frame
  - Fixed size, all pages/frames same size



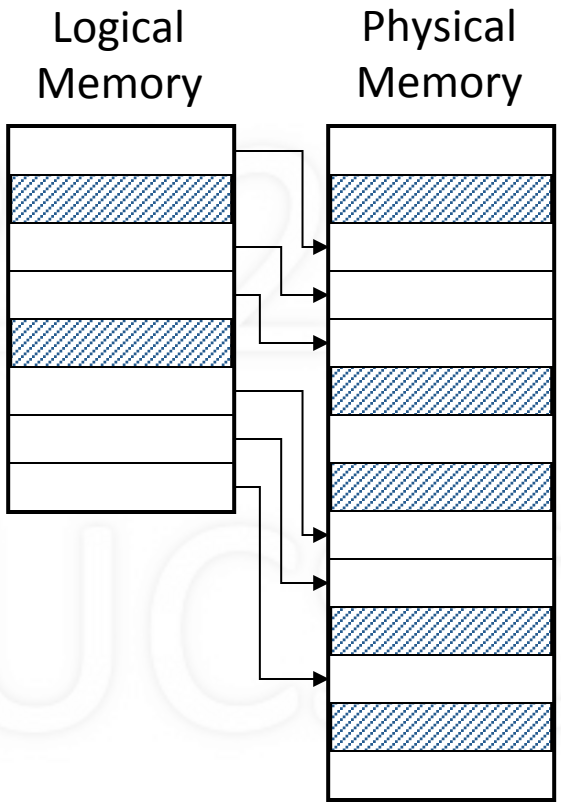
Page-based Logical Addressing

- Form of logical address: (*p*, *i*)
  - *p* is page number, 0 to N_L-1
  - *i* is offset within page
  - Note: *i* is less than page size
    - no need to check
- Size of logical address space
  - N_L = max number of pages
  - N_L × page size = size of logical address space



Frame-based Physical Addressing

- Form of physical address: (*f*, *i*)
  - *f* is frame number, 0 to N_P-1
  - *i* is offset within frame
  - Note: *i* is less than frame size
    - since page size = frame size
- Size of physical address space
  - N_P = max number of frames
  - N_P × frame size = size of physical address space

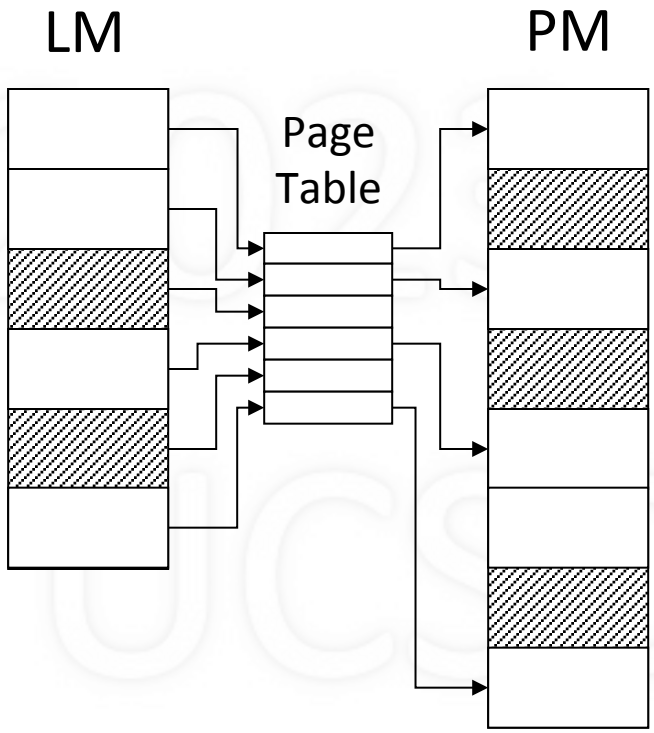


Page-based Address Translation

- Problem: how to translate
  - logical address (*p*, *i*) into physical address (*f*, *i*)?
- Solution: use a page (translation) table *PT*
  - to translate page *p* into frame *f* = *PT* (*p*)
  - then concatenate *f* and *i*
- Summary: physical address = *PT* (*p*) || *i*

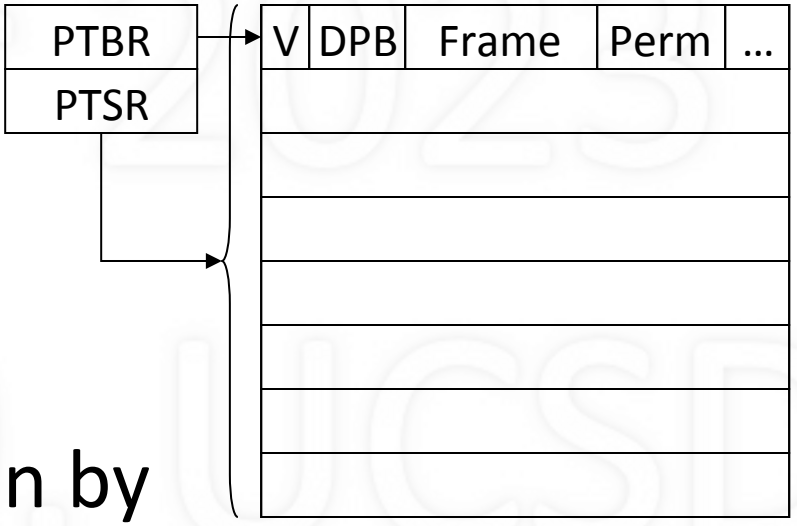
Logical Pages to Physical Frames

- Each page of logical memory corresponds to entry in page table
- Page table “maps” logical page into frame of physical memory



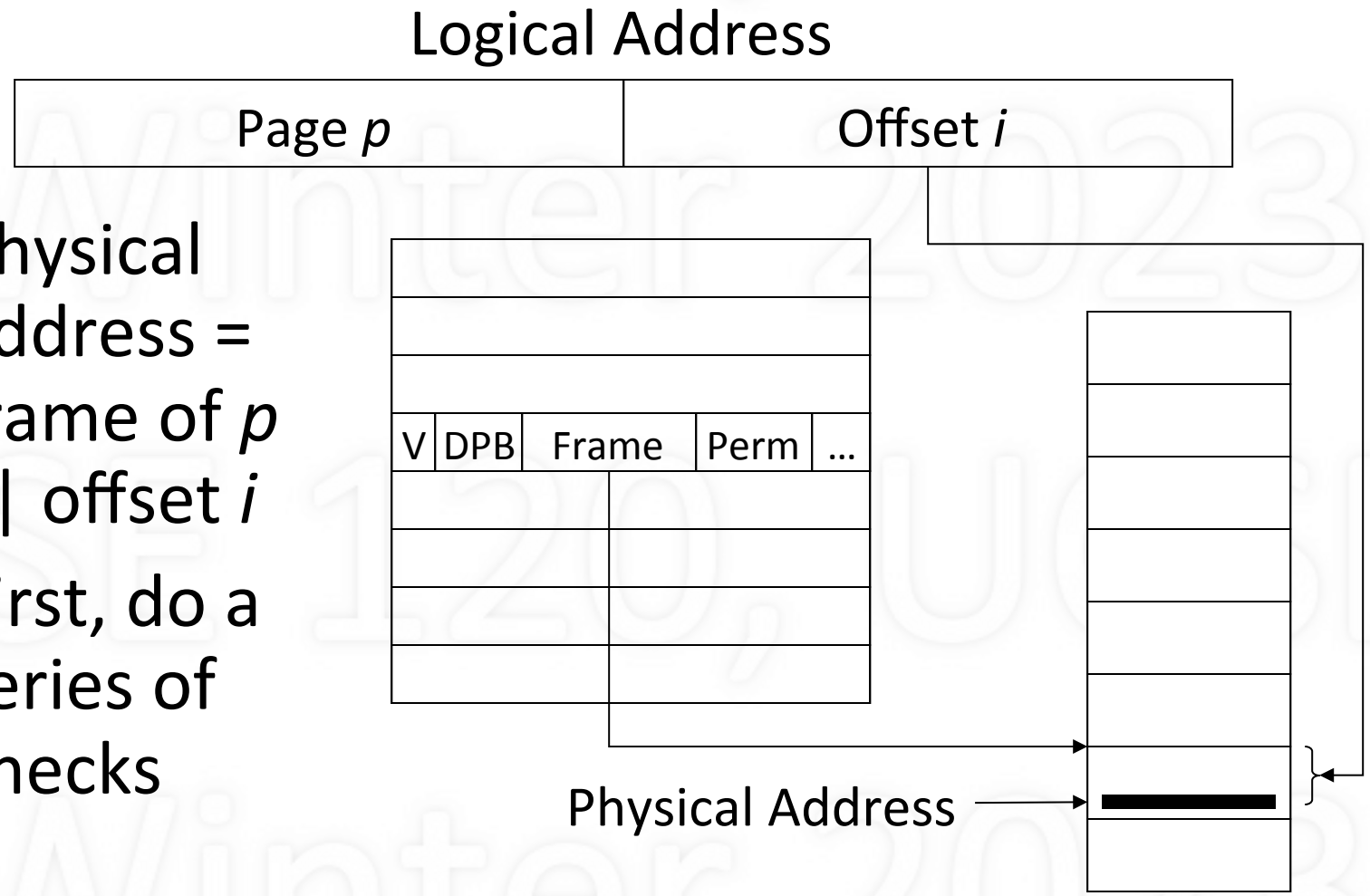
Page Table

- One per process (typically)
- Table entry elements
  - V: valid bit
  - Demand paging bits
  - Frame: page location
- Location in memory given by
  - Page table base register (hardware)
  - Page table size register (hardware)

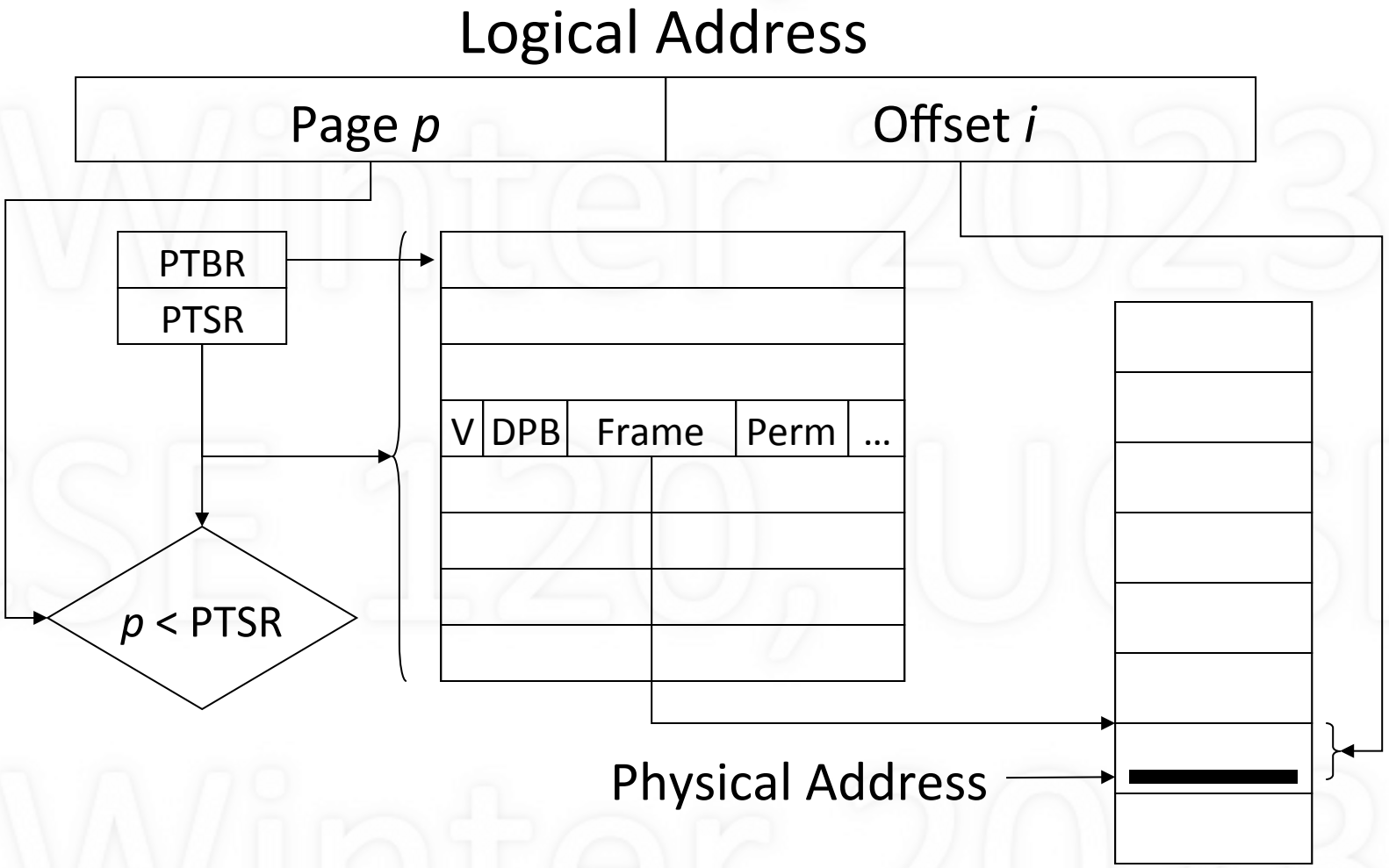


Address Translation

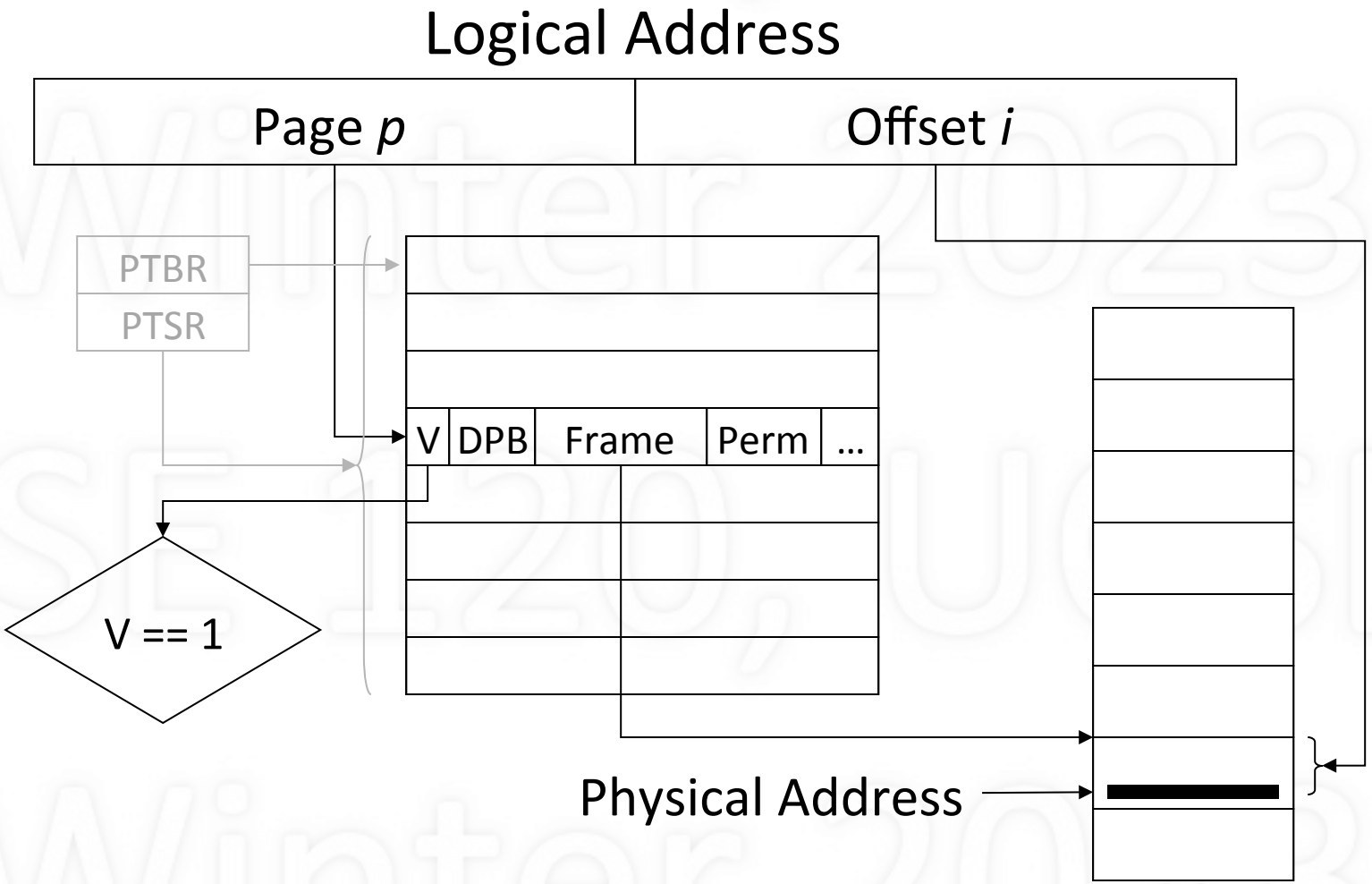
- Physical address = frame of *p* || offset *i*
- First, do a series of checks



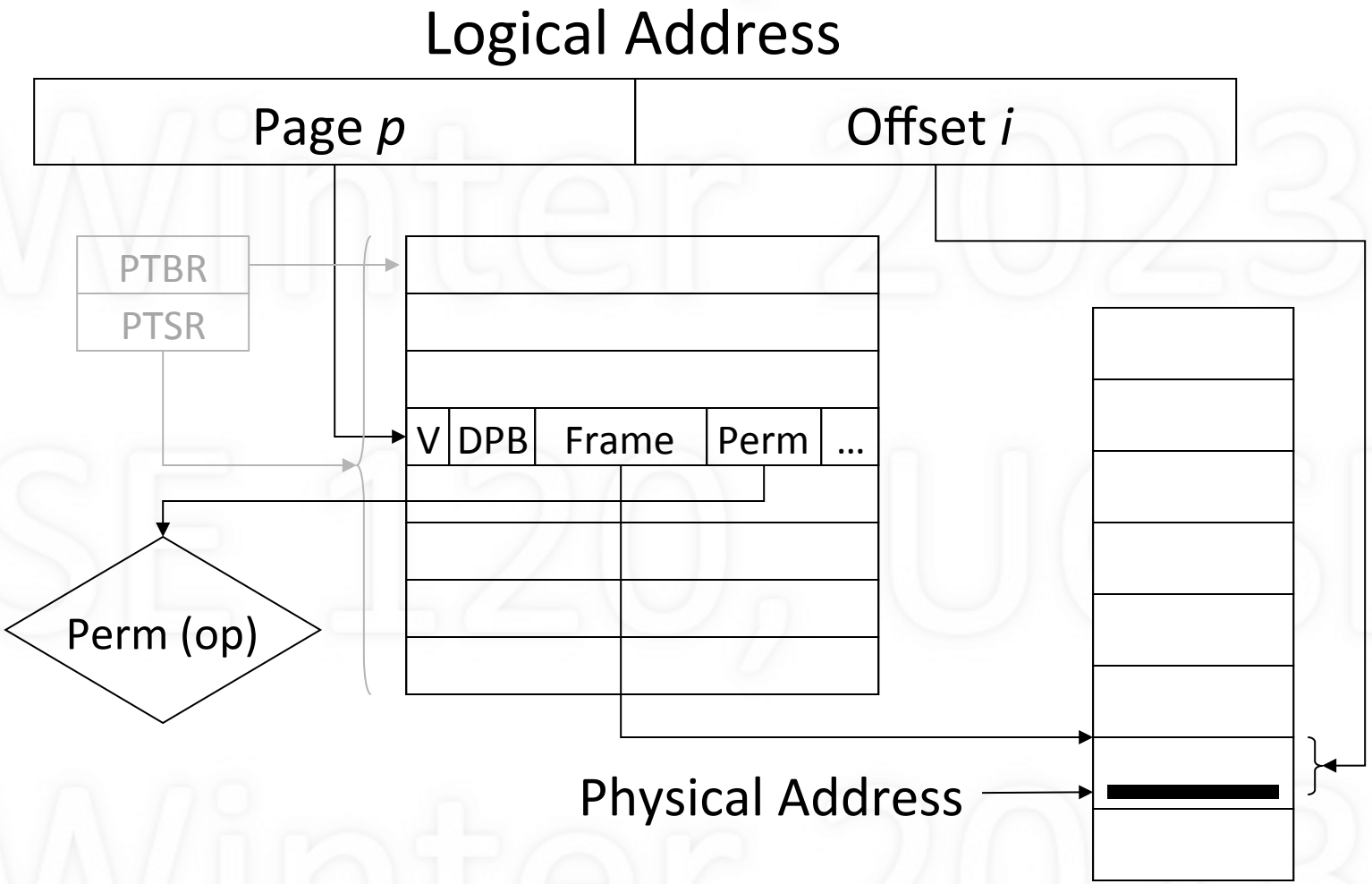
Check if Page *p* is Within Range



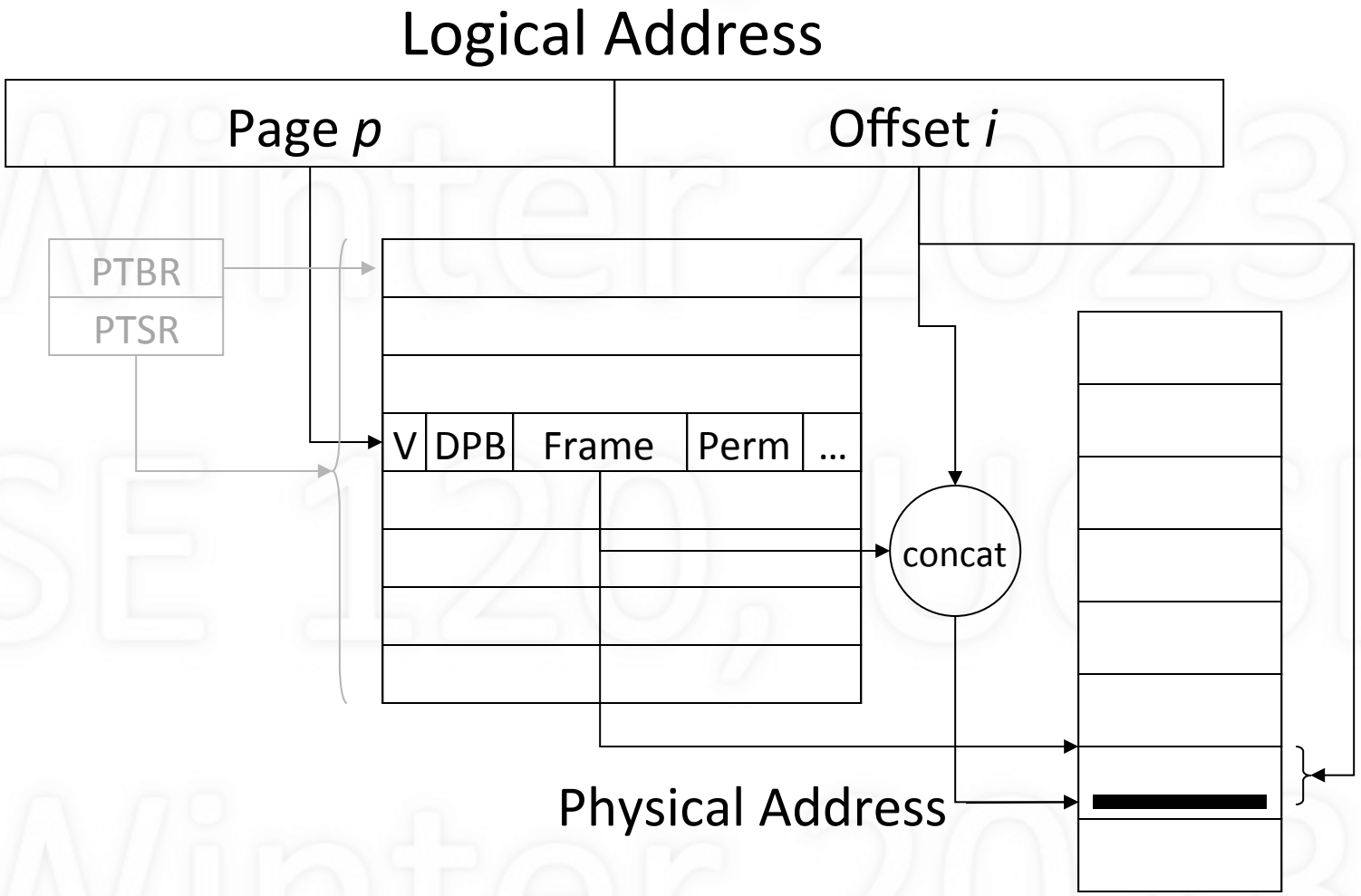
Check if Page Table Entry *p* is Valid



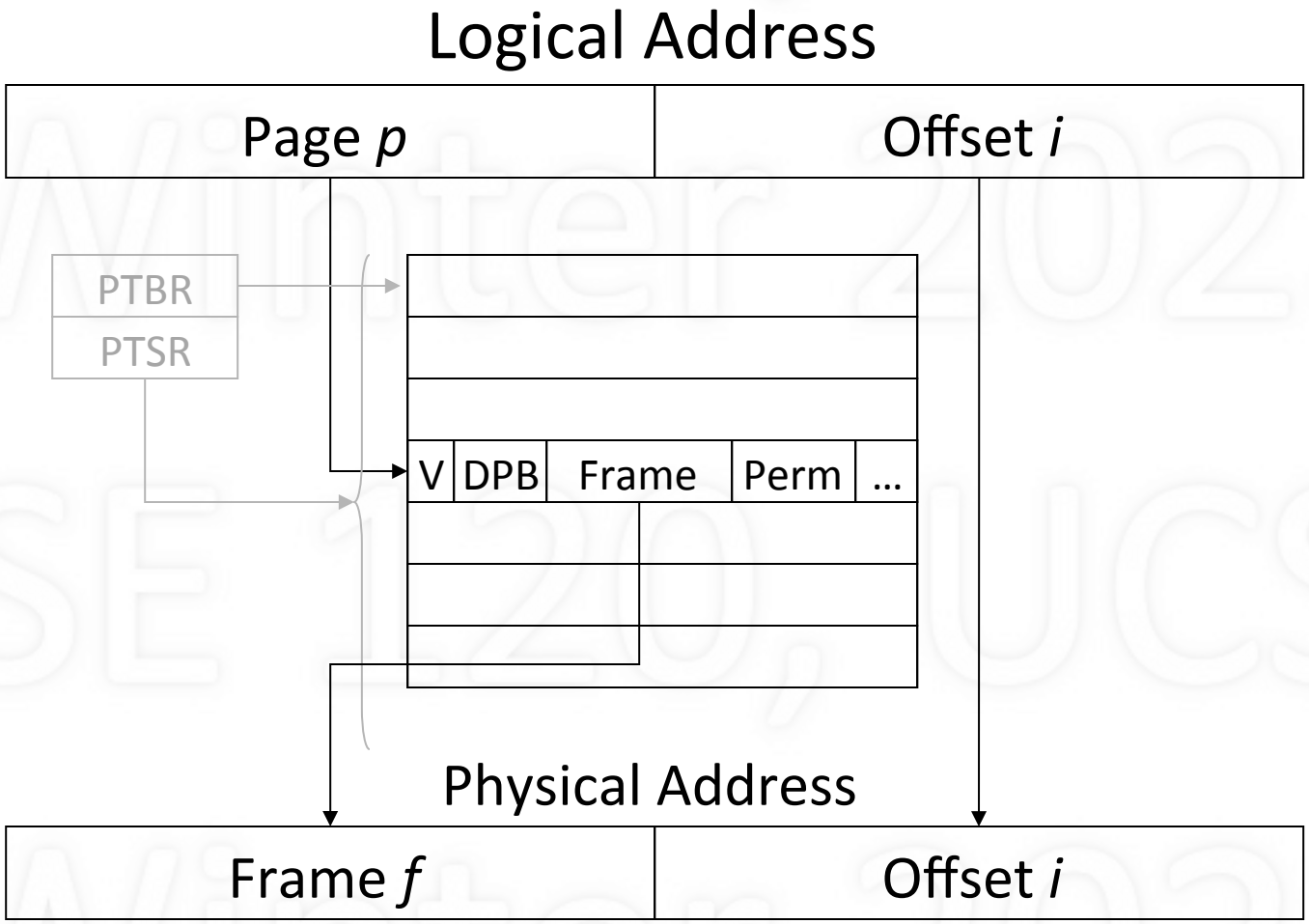
Check if Operation is Permitted



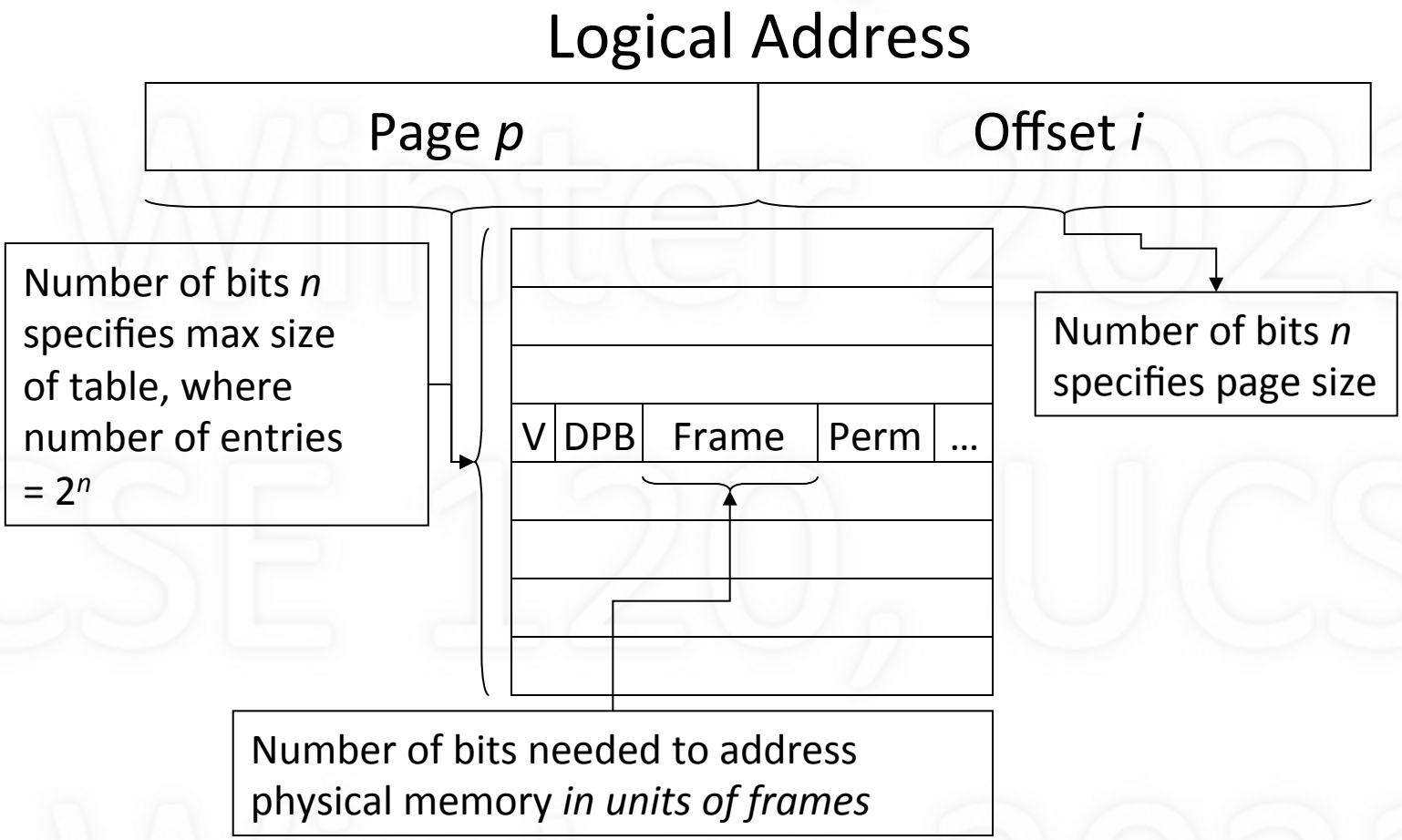
Translate Address



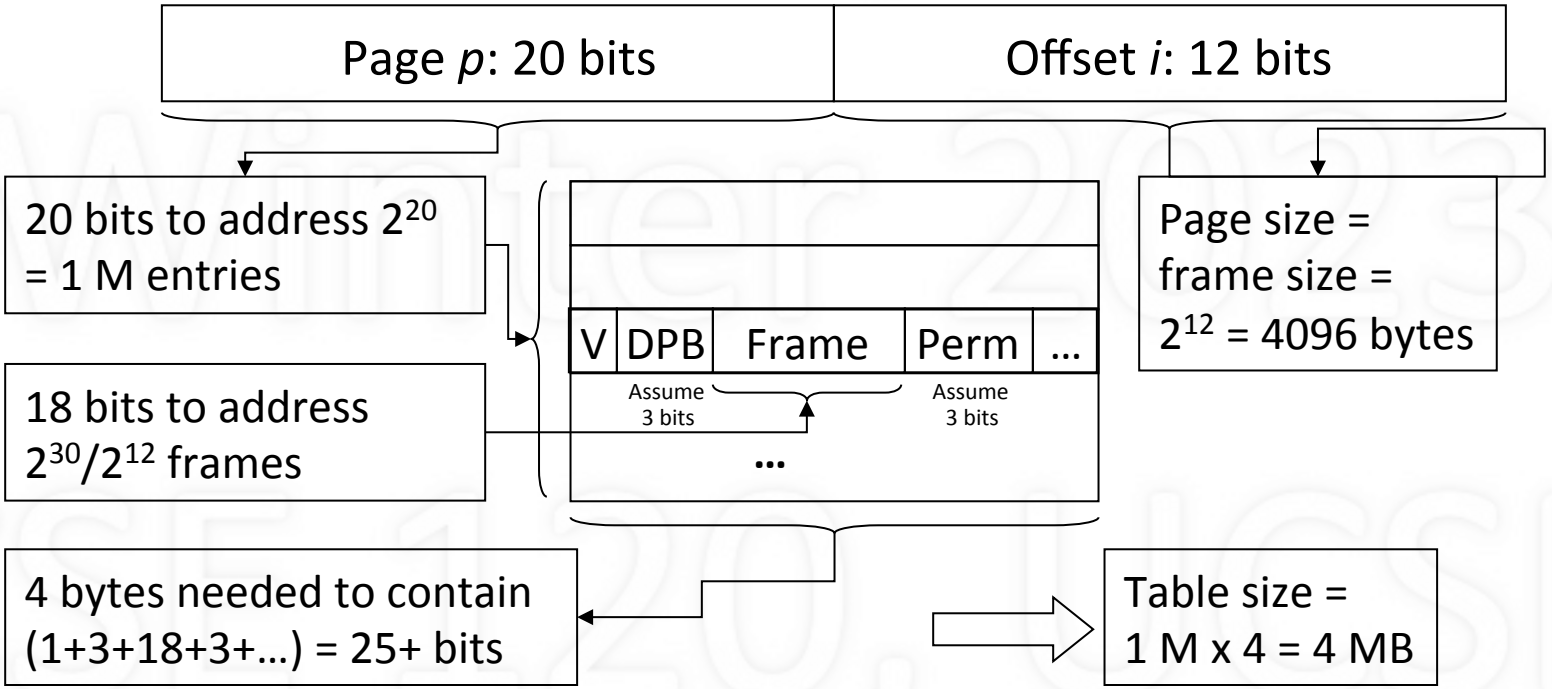
Physical Address by Concatenation



Sizing the Page Table



Example of Sizing the Page Table



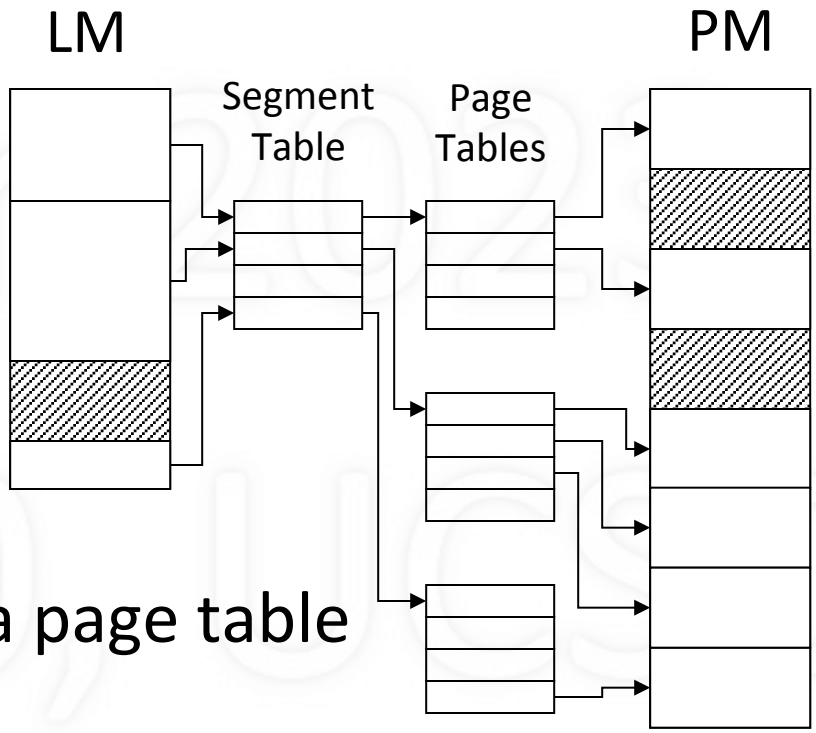
- Given 32 bit logical, 1 GB physical memory (max)
  - 20 bit page number, 12 bit offset

Segments vs. Pages

- Segment is good "logical" unit of information
  - Can be sized to fit any contents
  - Makes sense to share (e.g., code, data)
  - Can be protected according to contents
- Page is good "physical" unit of information
  - Simple memory management
- Why not have best of both
  - Segments, each a set of pages

Combining Segments and Pages

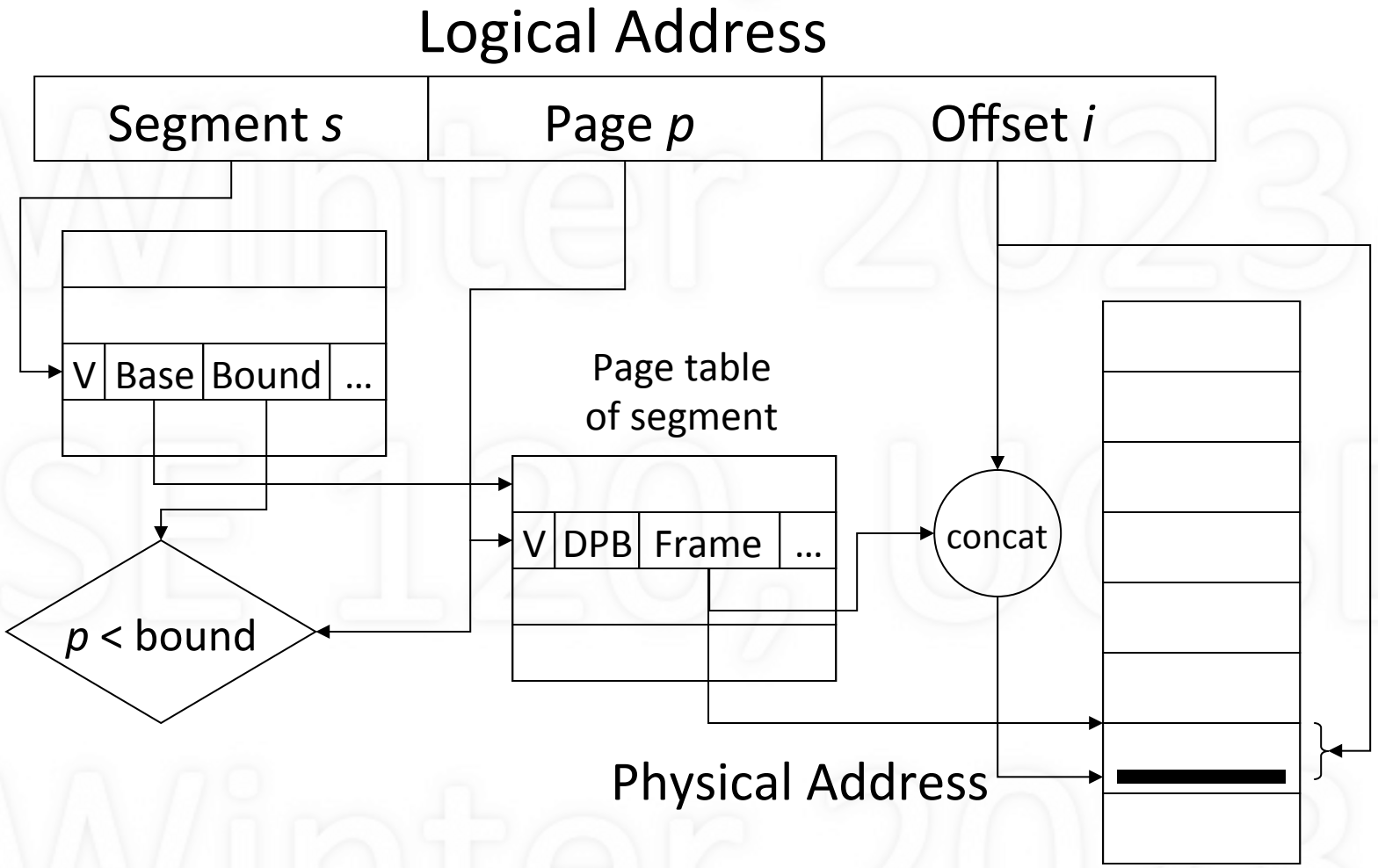
- Logical memory
  - composed of segments
- Each segment
  - composed of pages
- Segment table
  - Maps each segment to a page table
- Page tables
  - Maps each page to physical page frames



Address Translation

- Logical address: [segment *s*, page *p*, offset *i*]
- Do various checks
  - *s* < STSR, V == 1, *p* < bound, perm (op)
  - May get a segmentation violation
- Use *s* to index segment table to get page table
- Use *p* to index page table to get frame *f*
- Physical address = *concatenate* (*f*, *i*)

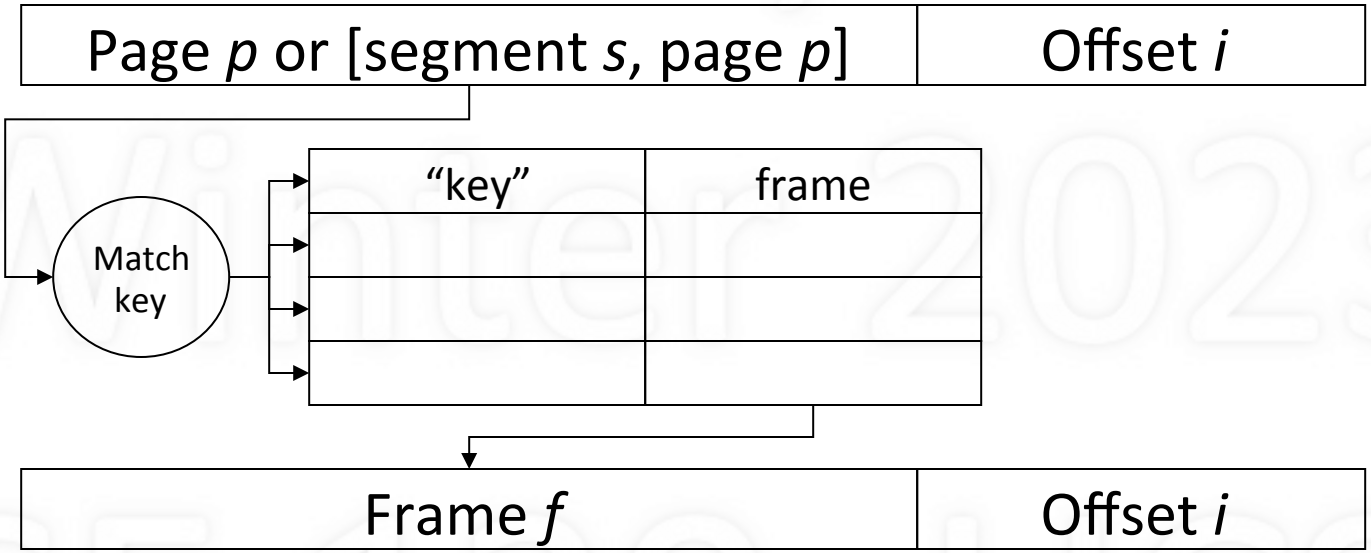
Segment/Page Address Translation



Cost of Translation

- Each lookup costs another memory reference
  - For each reference, additional references required
  - Slows machine down by factor of 2 or more
- Take advantage of locality of reference
  - Most references are to a small number of pages
  - Keep translations of these in high-speed memory
- Problem: don't know which pages till accessed

TLB: Translation Look-aside Buffer



- Fast memory keeps most recent translations
- If key matches, get frame number
- else wait for normal translation (in parallel)

Translation Cost with TLB

- Cost is determined by
  - Speed of memory: ~ 100 nsec
  - Speed of TLB: ~ 5 nsec
  - Hit ratio: fraction of refs satisfied by TLB, ~99%
- Speed with no address translation: 100 nsec
- Speed with address translation
  - TLB miss: 200 nsec (100% slowdown)
  - TLB hit: 105 nsec (5% slowdown)
  - Average: 105 x 0.99 + 200 x 0.01 ≈ 106 nsec

TLB Design Issues

- The larger the TLB
  - the higher the hit rate
  - the slower the response
  - the greater the expense
- TLB has a major effect on performance!
  - Must be flushed on context switches
  - Alternative: tagging entries with PIDs

Summary

- Logical memory
  - The Addressing Problem
  - The Protection Problem
- Organization of logical address space
  - Segmented
  - Paged
- Logical-to-physical address translation
- High cost of translation: locality, TLB

Textbook

- OSP: Chapter 8
- OSC: Chapter 9 (Main Memory)
  - Lecture-related: 9.1-9.3, 9.8
  - Recommended: 9.4, 9.6-9.7