Virtual Memory

Kevin Webb

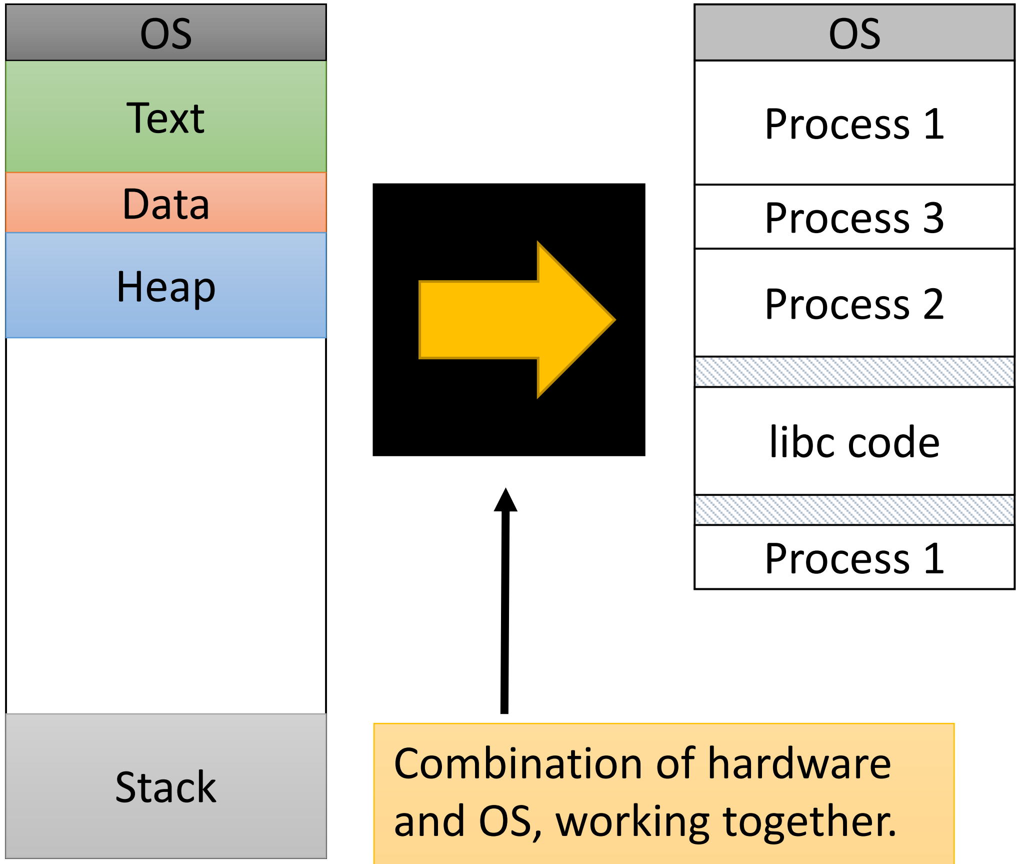
Swarthmore College

February 27, 2020

Today's Goals

- Describe the mechanisms behind address translation.
- Analyze the performance of address translation alternatives.
- ~~Explore page replacement policies for disk swapping.~~

Address Translation: Wish List



Combination of hardware and OS, working together.

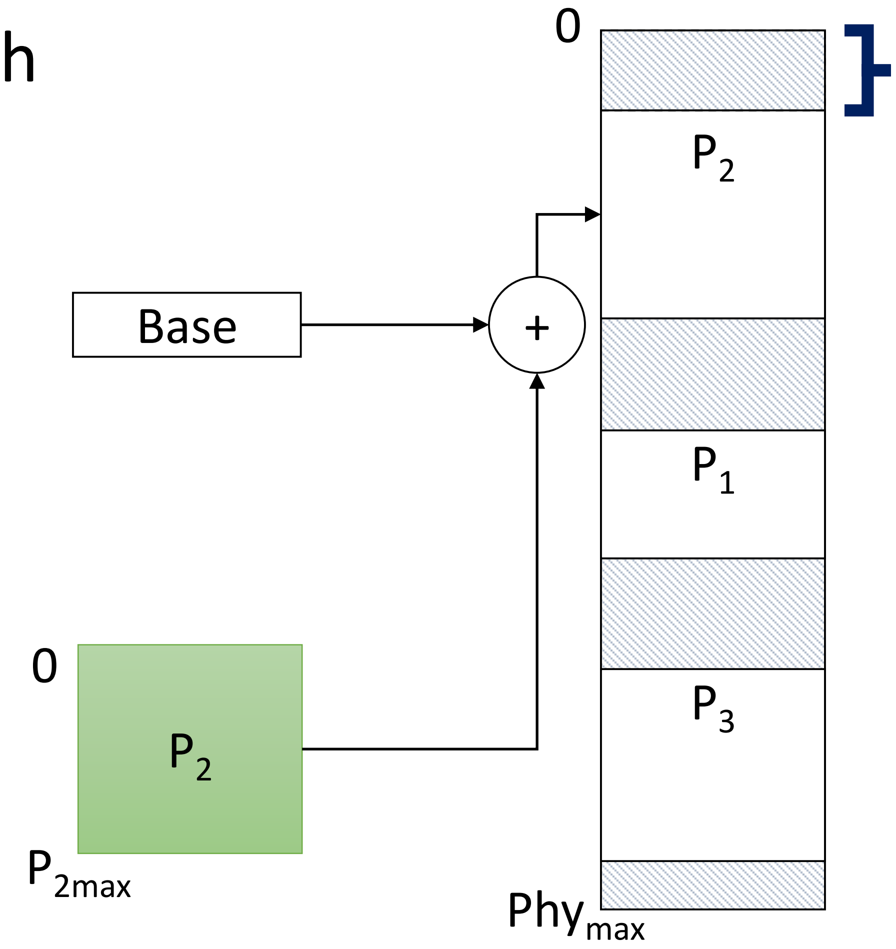
In hardware, MMU:
Memory Management Unit

- Map virtual addresses to physical addresses.
- Allow multiple processes to be in memory at once, but isolate them from each other.
- Determine which subset of data to keep in memory / move to disk.
- Allow the same physical memory to be mapped in multiple process VASes.
- Make it easier to perform placement in a way that reduces fragmentation.
- Map addresses quickly with a little HW help.

Simple (Unrealistic) Translation Example

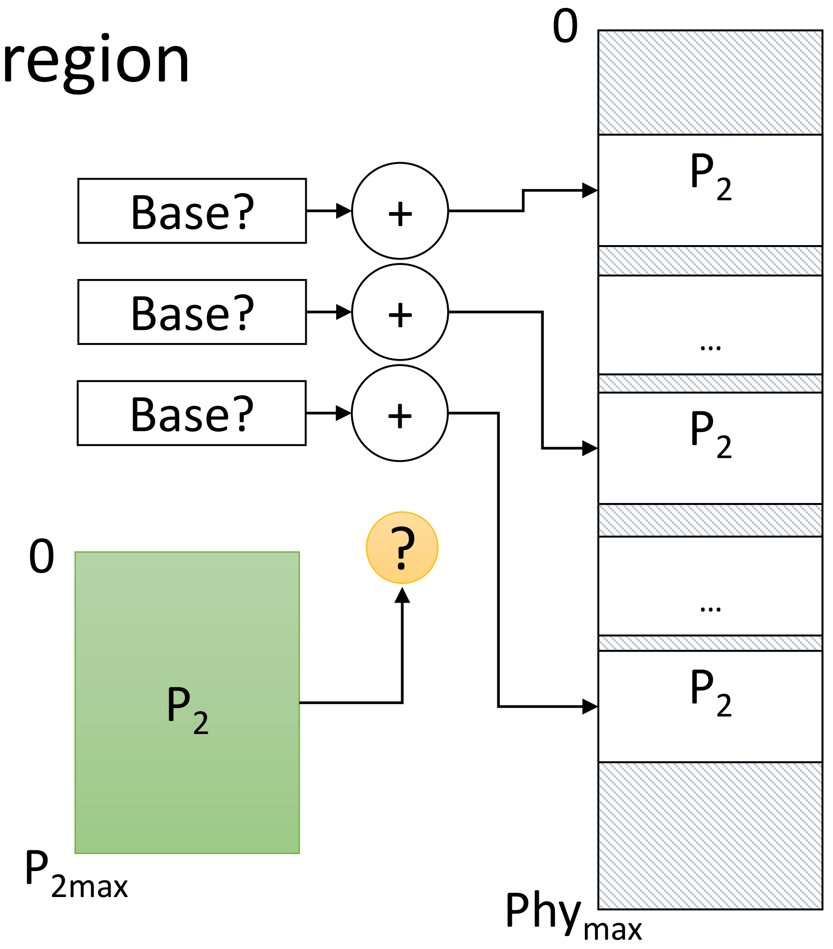
- Process P_2's virtual addresses don't align with physical memory's addresses.

- Determine offset from physical address 0 to start of P_2, store in *base*.



Generalizing

- Problem: process may not fit in one contiguous region

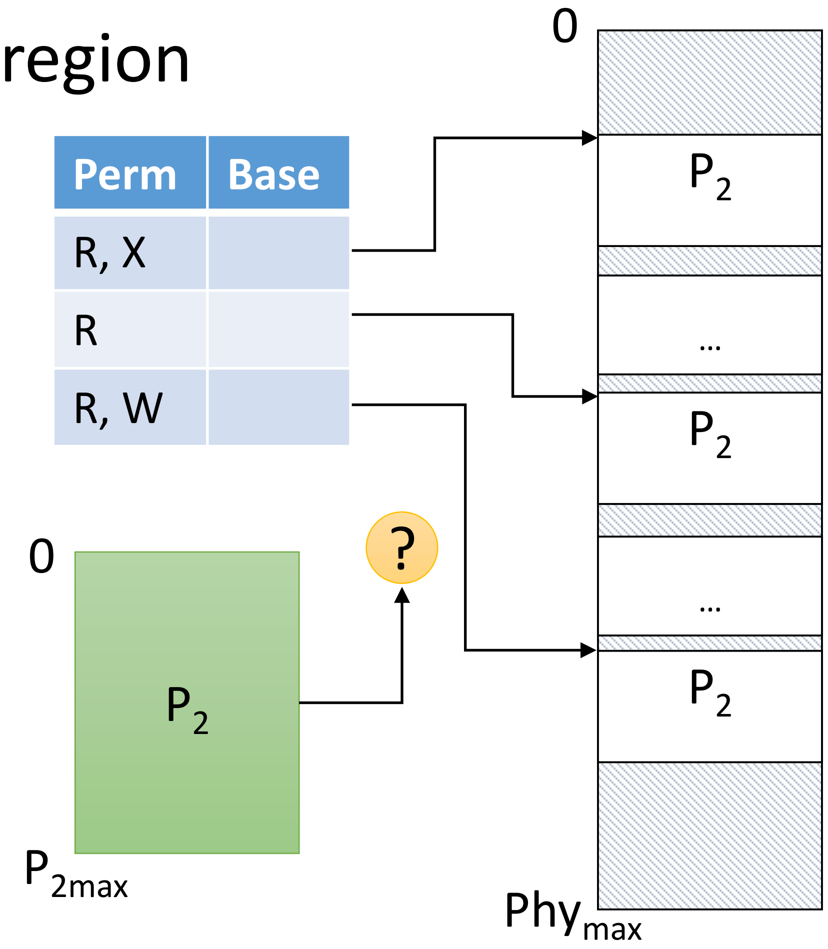


Generalizing

- Problem: process may not fit in one contiguous region

- Solution: keep a table (one for each process)
  - Keep details for each region in a row
  - Store additional metadata (ex. permissions)

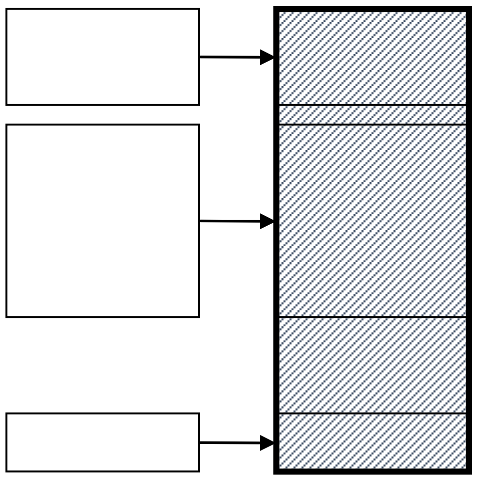
- Interesting questions:
  - How many regions should there be (and what size)?
  - How to determine which row we should use?



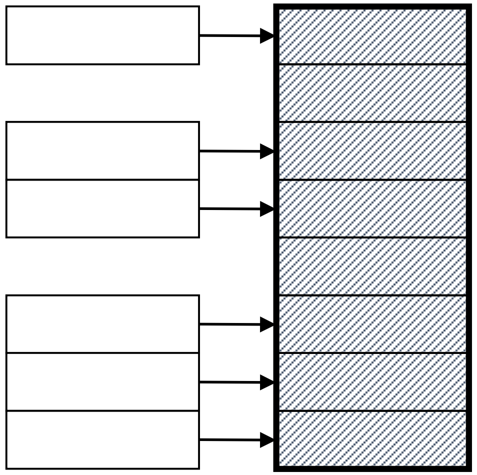
Perm	Base
R, X	
R	
R, W	

Defining Regions - Two Approaches

- Segmentation:
  - Partition address space and memory into segments
  - Segments have varying sizes



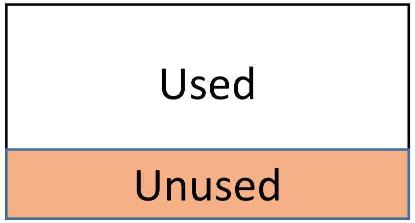
- Paging:
  - Partition address space and memory into pages
  - Pages are a constant, fixed size



Fragmentation

Internal

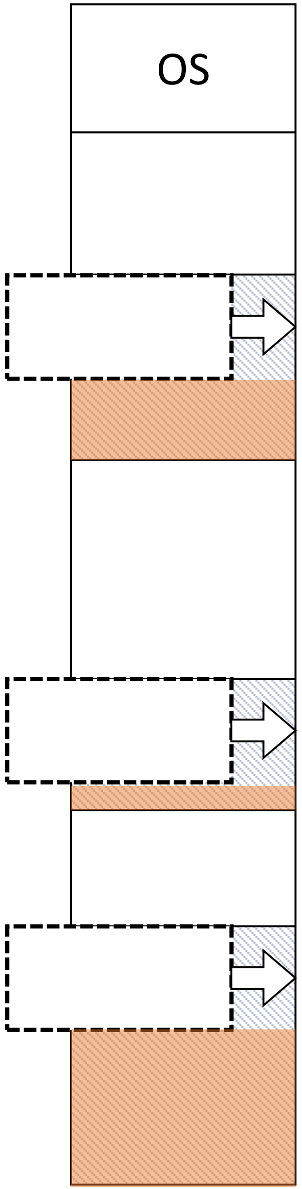
- Process asks for memory, doesn't use it all.
- Possible reasons:
  - Process was wrong about needs
  - OS gave it more than it asked for
- *internal*: within an allocation



Memory allocated to process

External

- Over time, we end up with these small gaps that become more difficult to use (eventually, wasted).
- *external*: unused memory between allocations



Which scheme is better for reducing internal and external fragmentation. Why?

A. Segmentation is better than paging for both forms of fragmentation.

B. Segmentation is better for *internal* fragmentation, and paging is better for *external* fragmentation.

C. Paging is better for *internal* fragmentation, and segmentation is better for *external* fragmentation.

D. Paging is better than segmentation for both forms of fragmentation.

Segmentation vs. Paging

- A segment is good *logical* unit of information
  - Can be sized to fit any contents
  - Easy to share large regions (e.g., code, data)
  - Protection requirements correspond to logical data segment

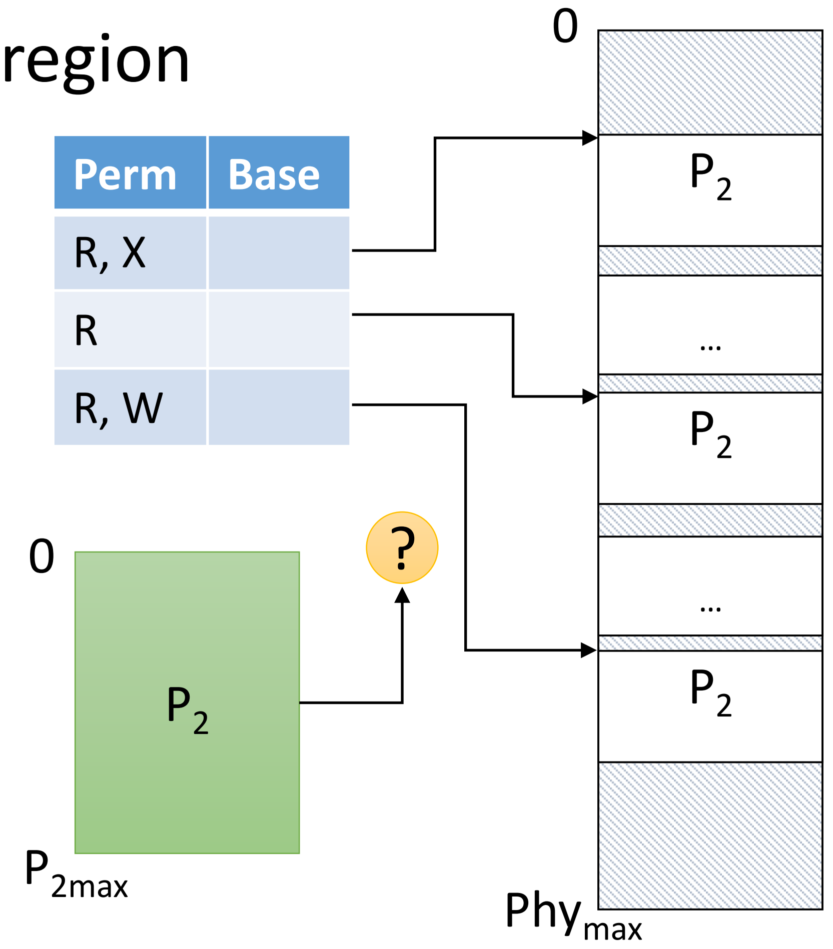
- A page is good *physical* unit of information
  - Simple physical memory placement
  - No external fragmentation
  - Constant sizes make it easier for hardware to help

Generalizing

- Problem: process may not fit in one contiguous region

- Solution: keep a table (one for each process)
  - Keep details for each region in a row
  - Store additional metadata (ex. permissions)

- Interesting questions:
  - How many regions should there be (and what size)?
  - **How to determine which row we should use?**



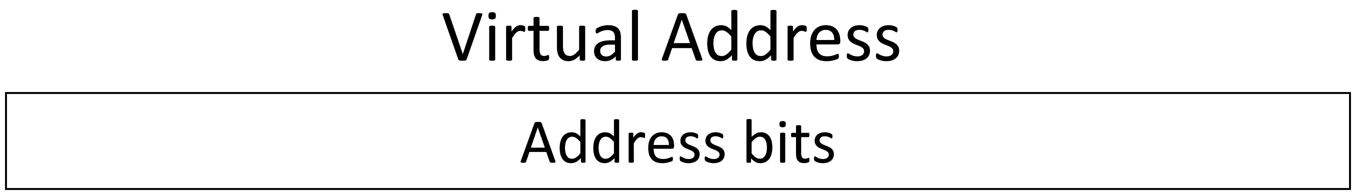
Perm	Base
R, X	
R	
R, W	

For **both** segmentation and paging…

- **Each process** gets a table to track memory address translations.

- When a process attempts to read/write to memory:
    - It attempts to access a virtual address from its virtual address space

Address Translation



Virtual Address

Address bits

- Userspace process accesses memory by supplying an address:
  - `movl (%eax), %ecx`

- Send the bits held in register %eax to memory to retrieve contents.

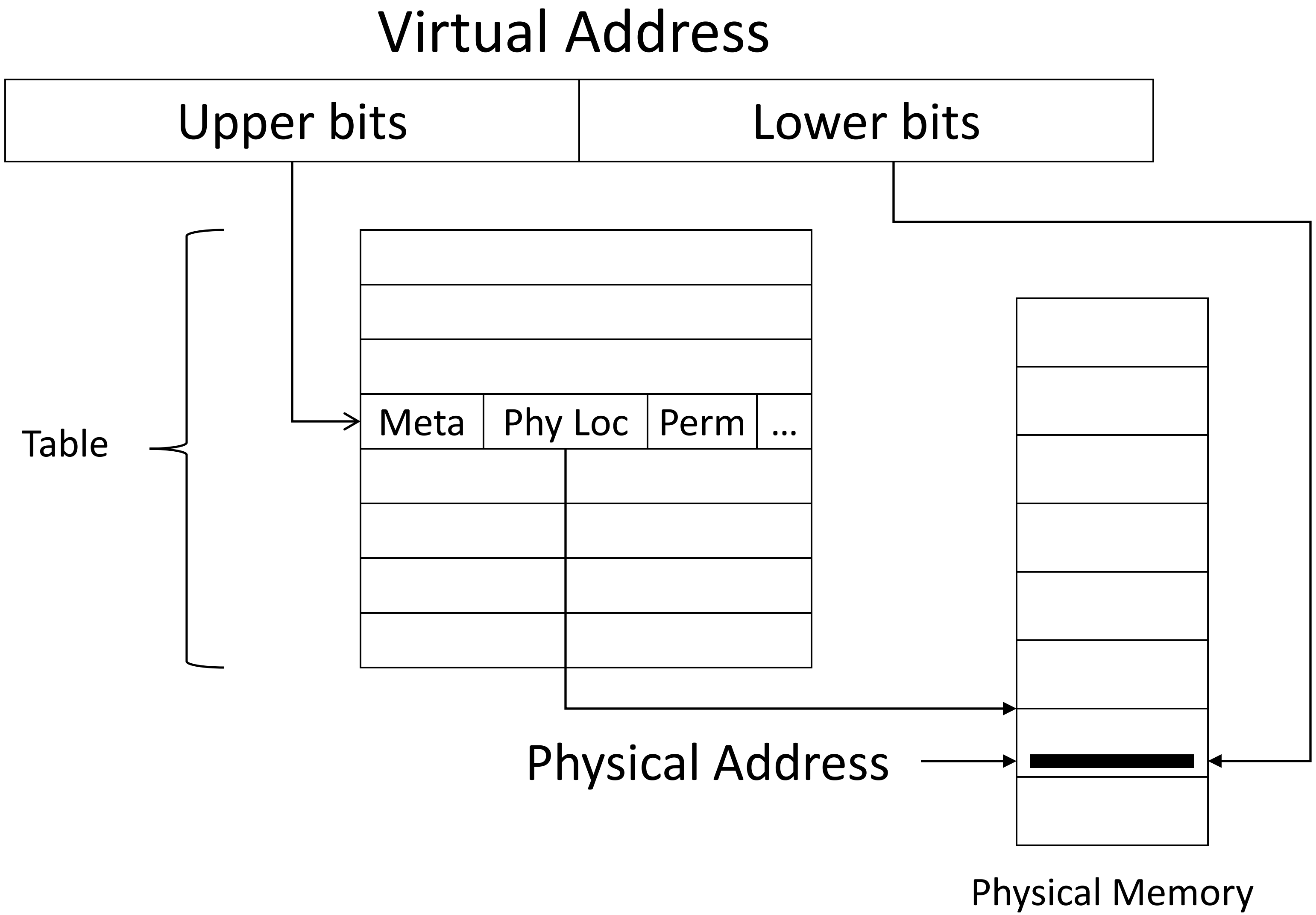
Address Translation

Virtual Address

Upper bits	Lower bits

- Insight: we can use the address itself to make translation easier
    - Break the address into two (or more) regions
    - Interpret one (or more) regions as an index into the table

Address Translation



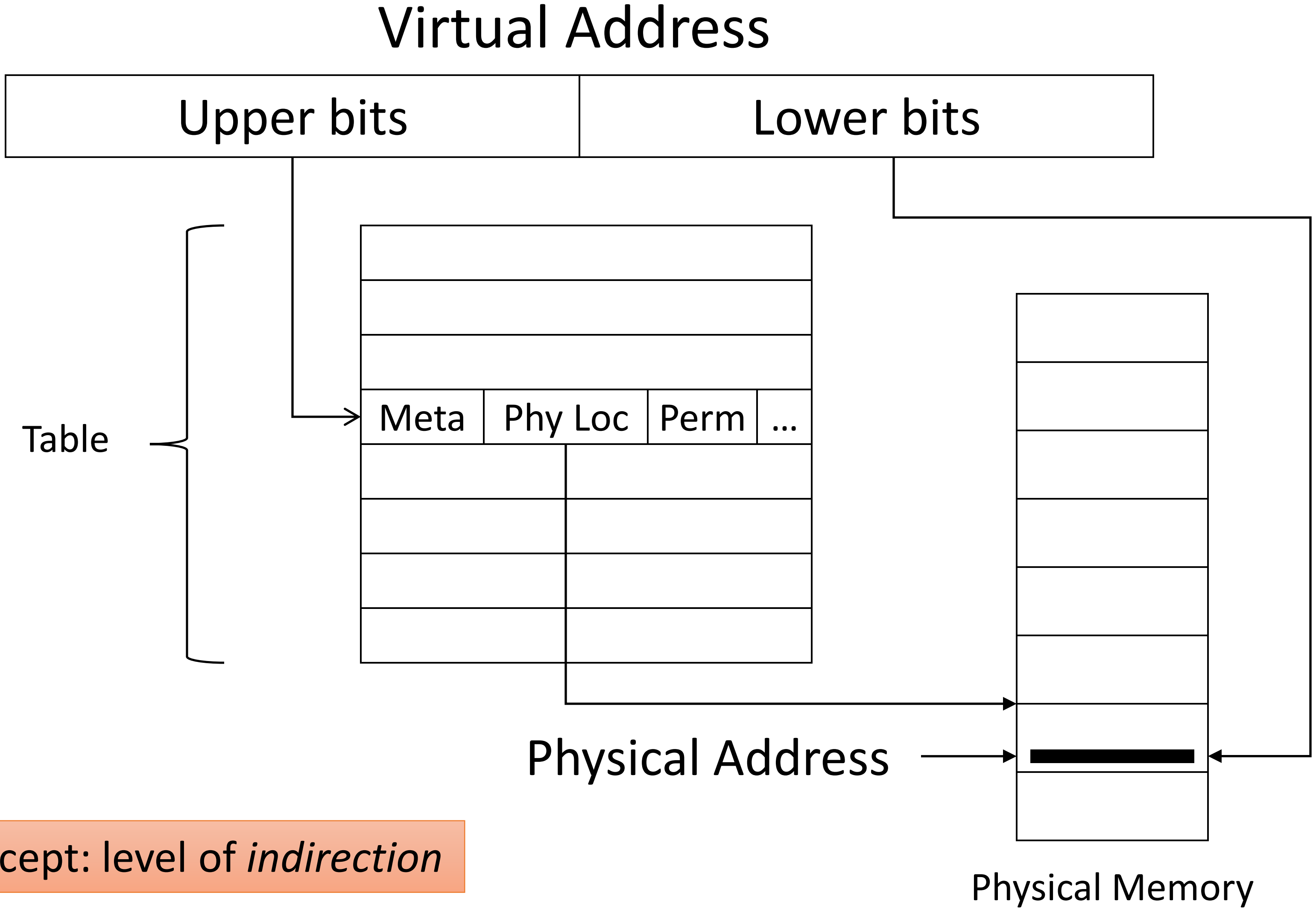
Performance Implications

Without VM:

Go directly to address in memory.

With VM:

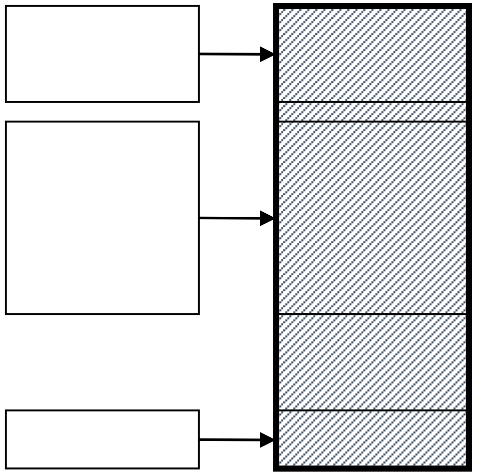
Do a lookup in memory to determine which address to use.



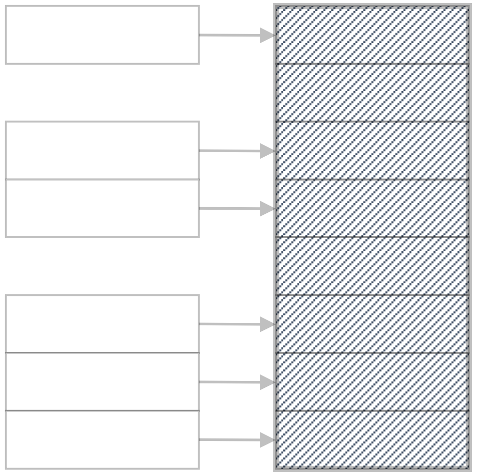
Concept: level of *indirection*

Defining Regions - Two Approaches

- Segmentation:
  - Partition address space and memory into segments
  - Segments have varying sizes

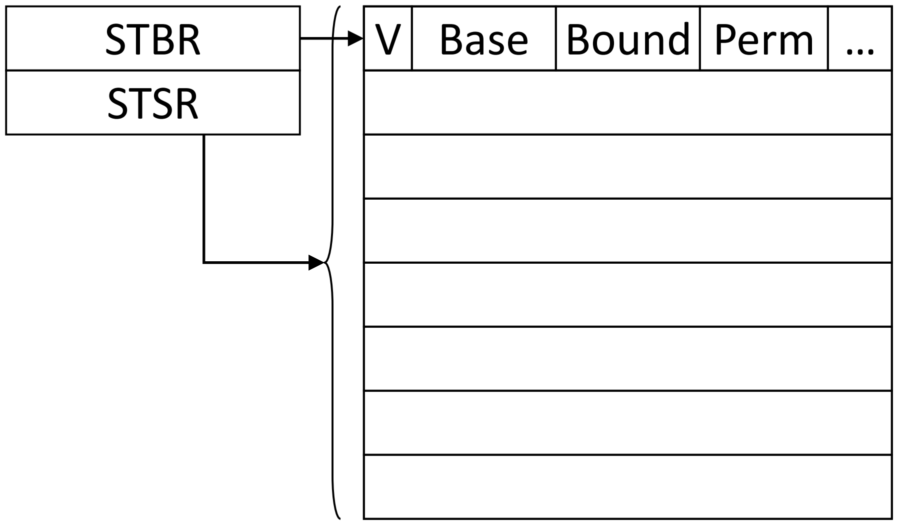


- Paging:
  - Partition address space and memory into pages
  - Pages are a constant, fixed size



Segment Table

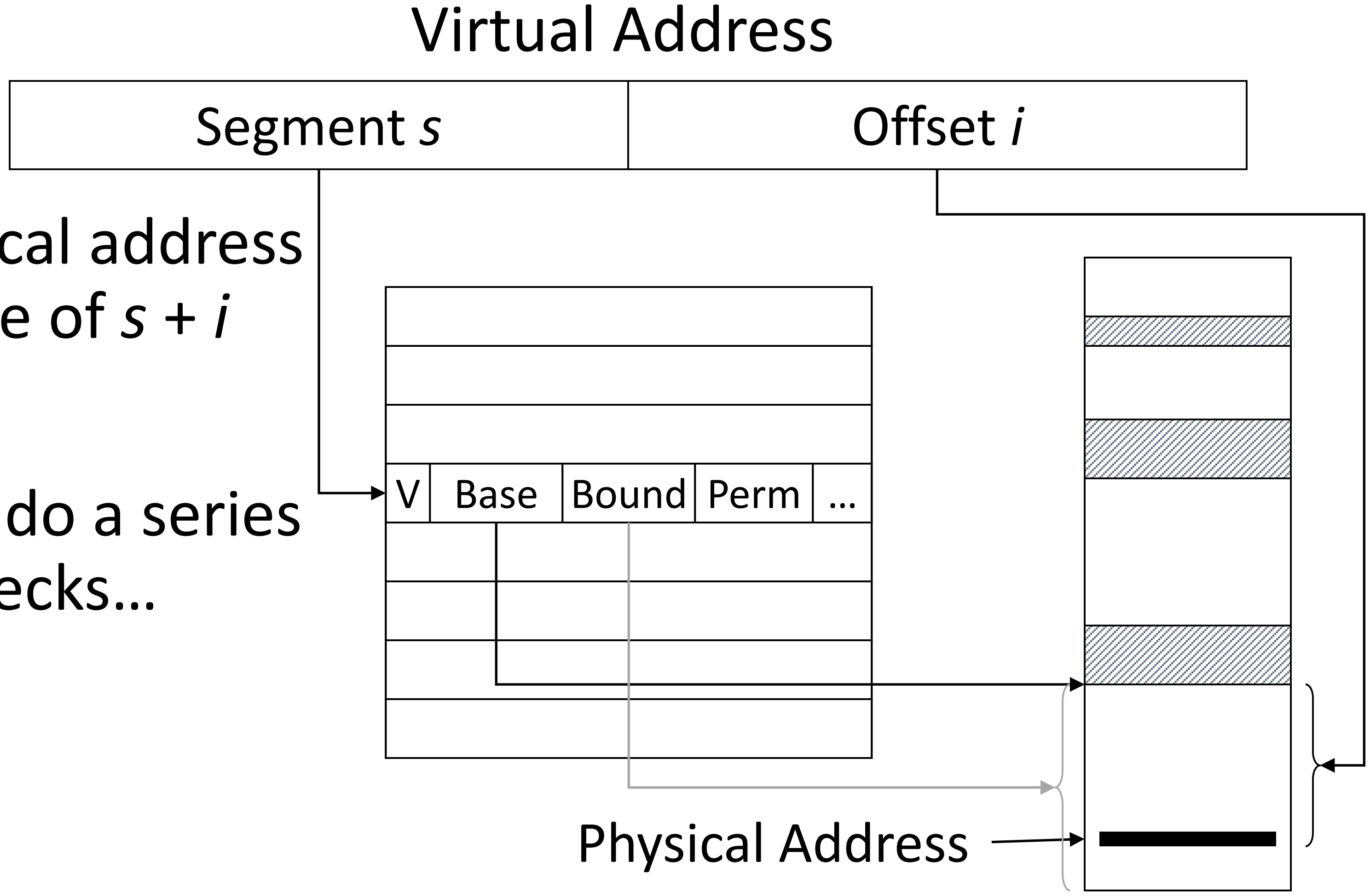
- One table per process
- Where is the *table* located in memory?
  - Segment table base register (STBR)
  - Segment table size register (STSR)
- Table entry elements
  - V: valid bit (does it contain a mapping?)
  - Base: segment location in physical memory
  - Bound: segment size in physical memory
  - Permissions



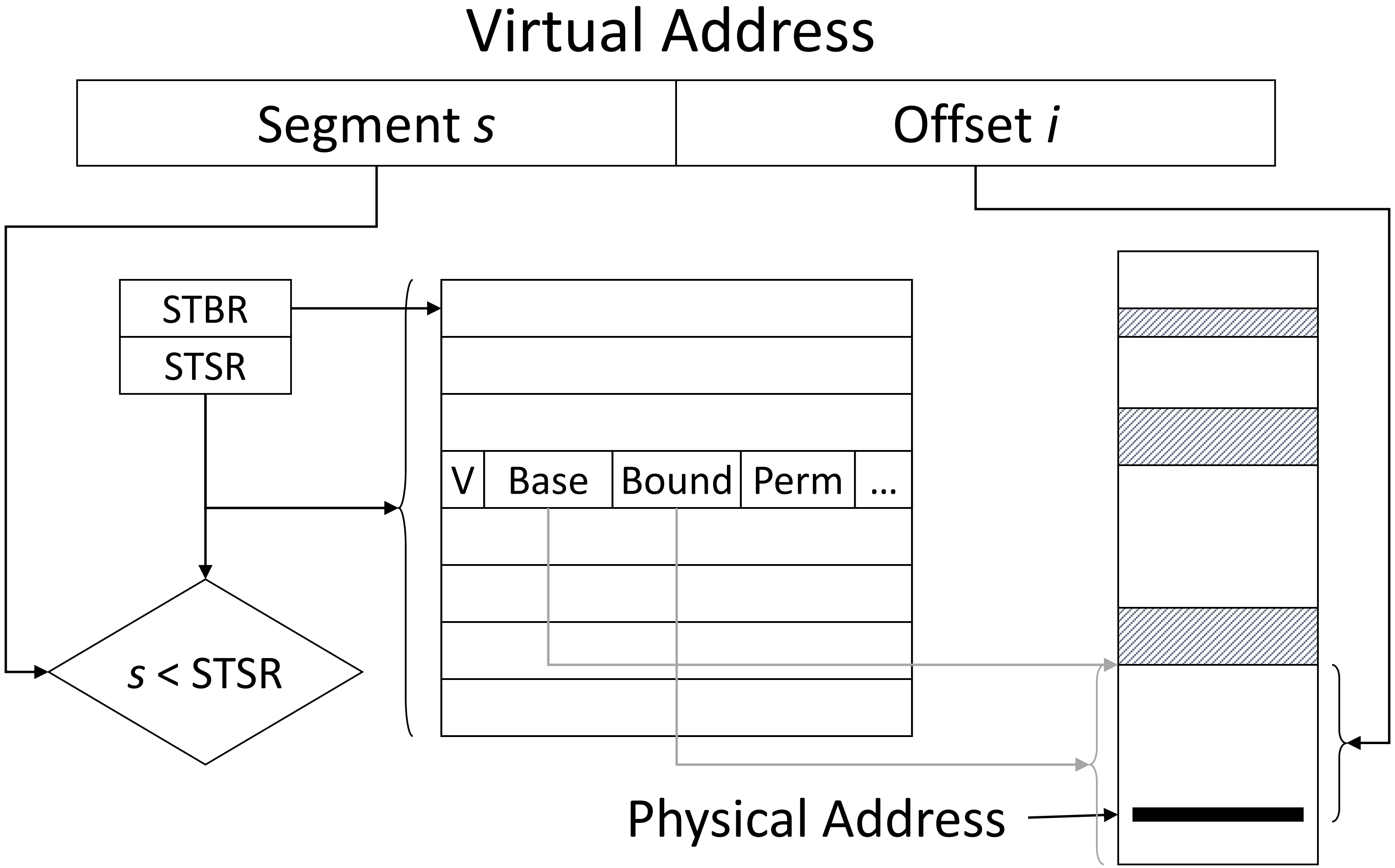
Address Translation

- Physical address = base of *s* + *i*

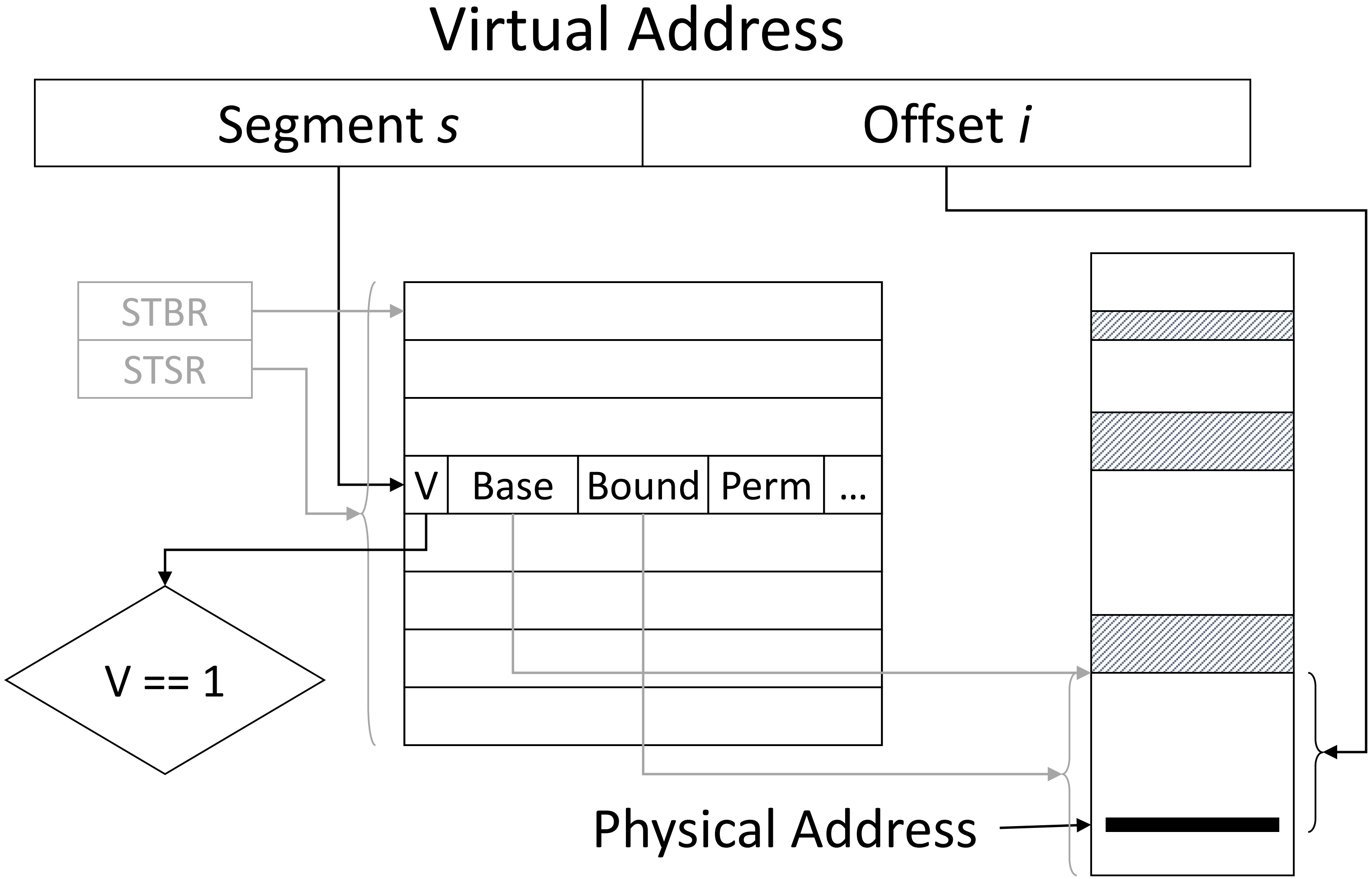
- First, do a series of checks…



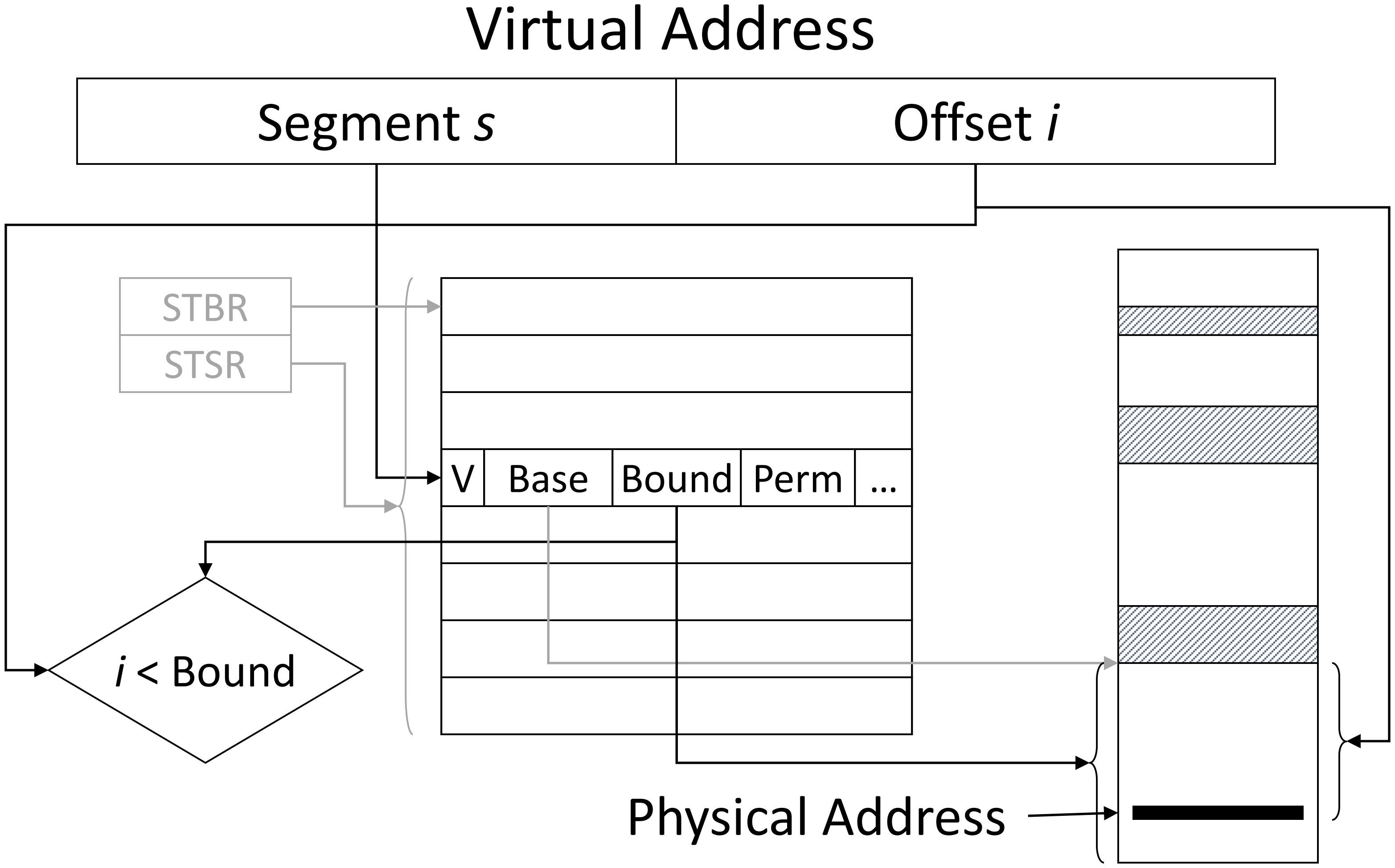
Check if Segment *s* is within Range



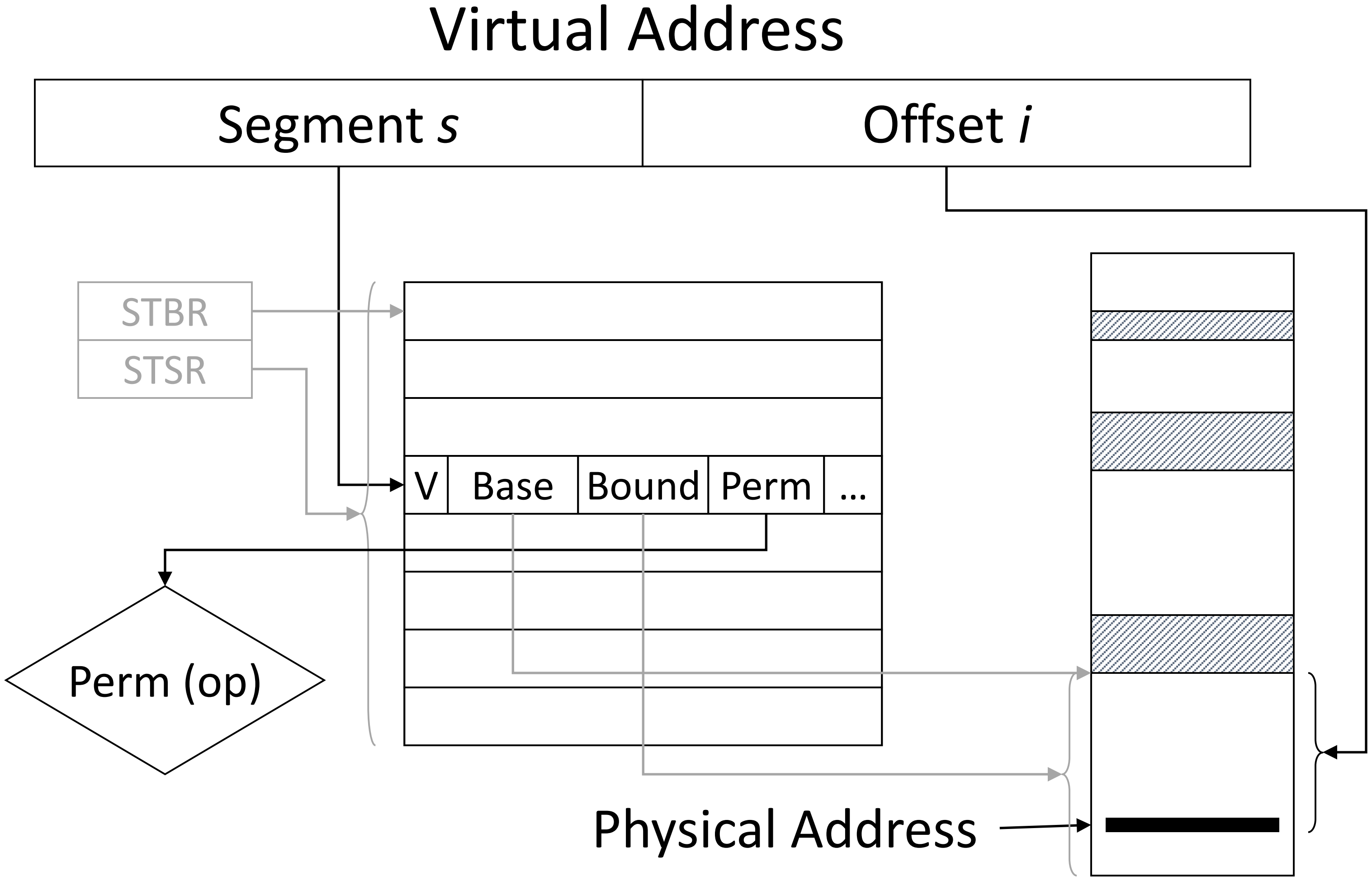
Check if Segment Entry *s* is Valid



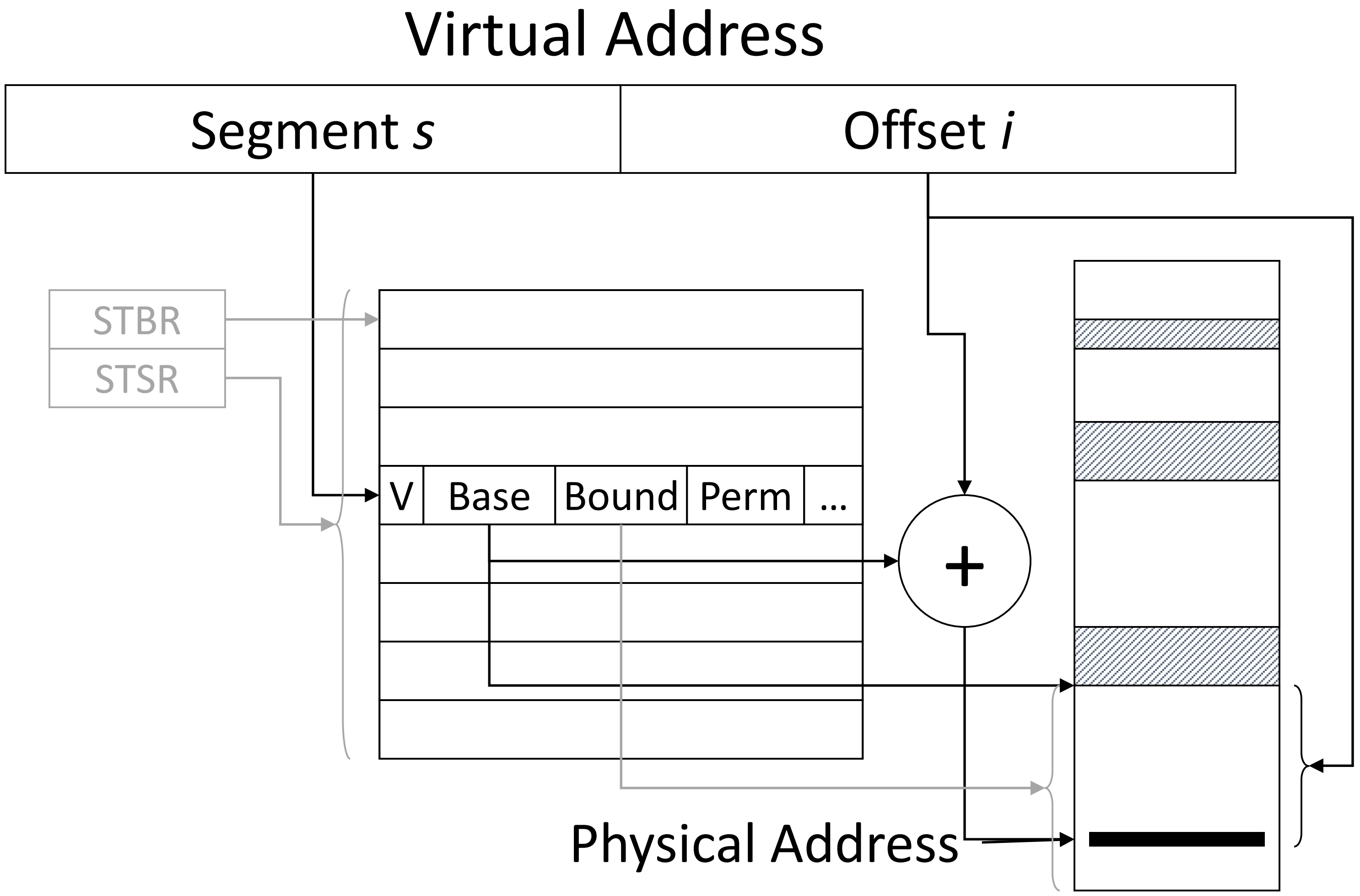
Check if Offset *i* is within Bounds



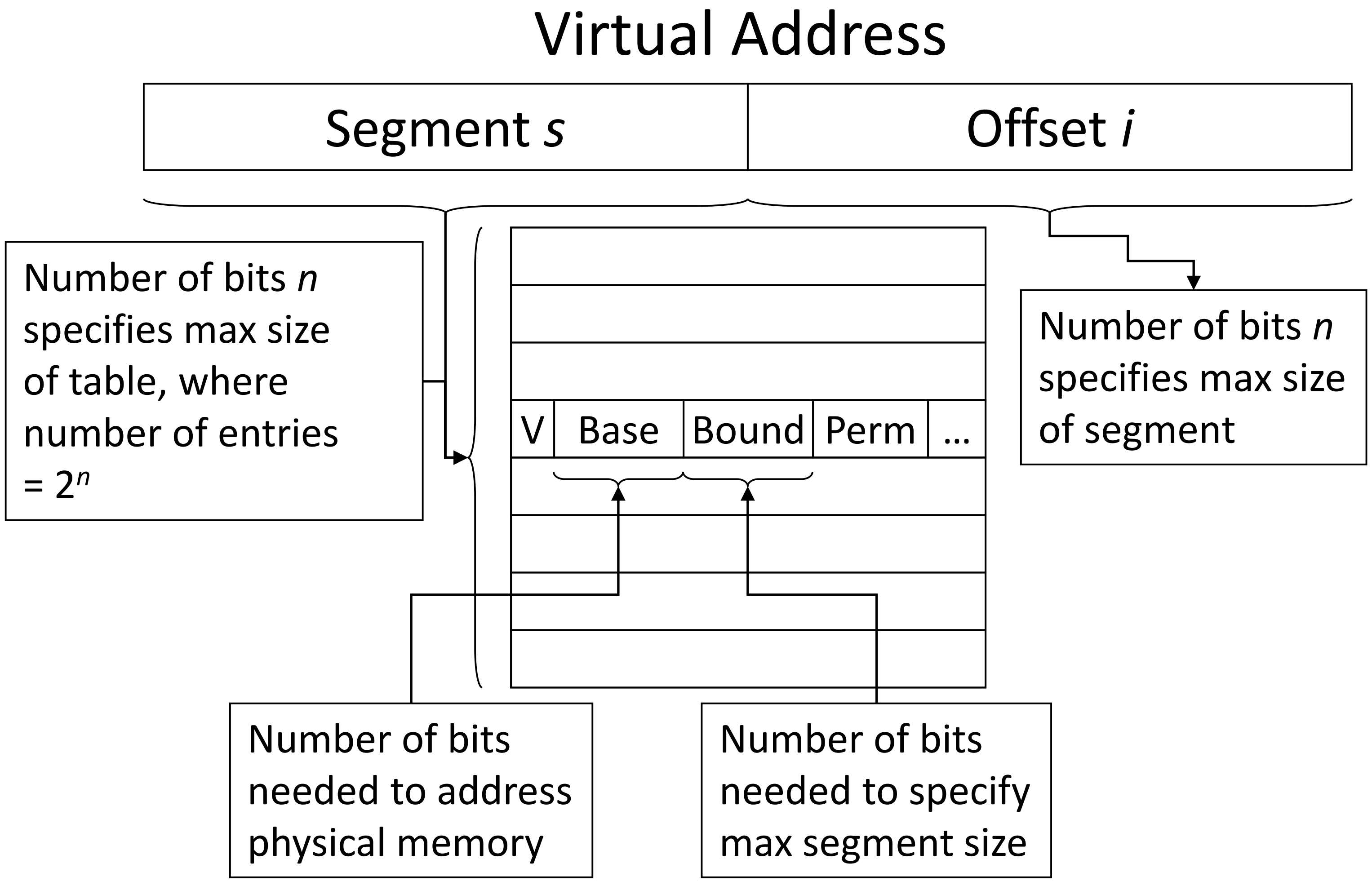
Check if Operation is Permitted



Translate Address



Sizing the Segment Table



Virtual Address

Segment *s*	Offset *i*

Number of bits *n* specifies max size of table, where number of entries = 2^n

V	Base	Bound	Perm	…

Number of bits *n* specifies max size of segment

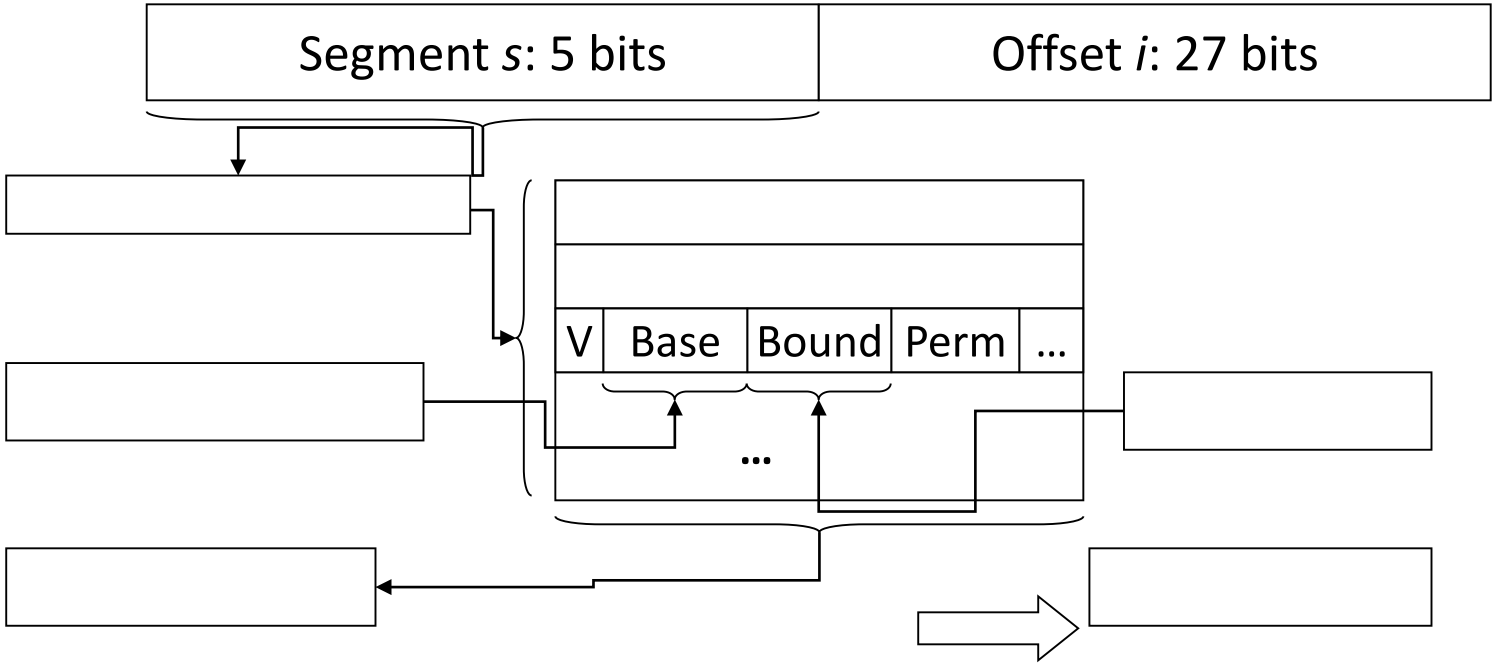
Number of bits needed to address physical memory

Number of bits needed to specify max segment size

Helpful reminder:

2^{10} => Kilobyte
2^{20} => Megabyte
2^{30} => Gigabyte

Example of Sizing the Segment Table



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

5 bit segment address, 32 bit logical address, 1 GB Physical memory.
How many entries (rows) will we have in our segment table?

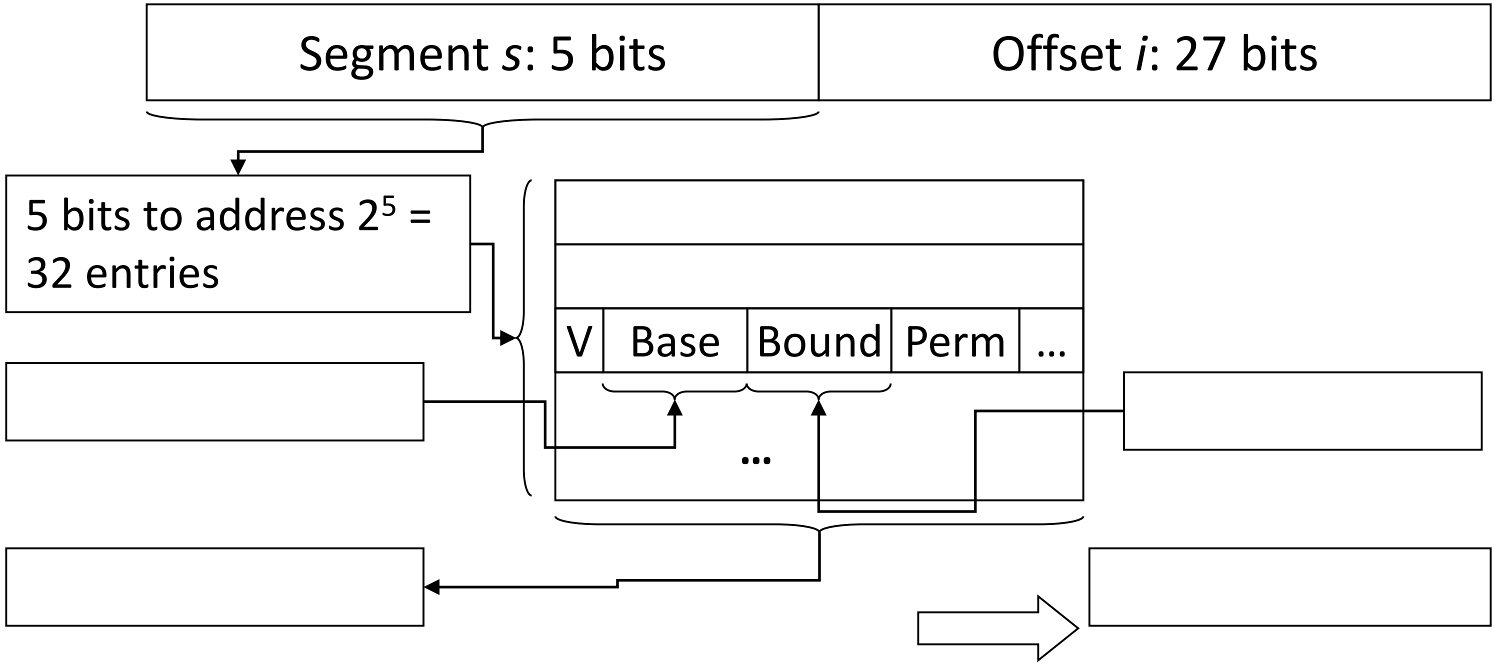
A. 32: The logical address size is 32 bits

B. 32: The segment address is five bits

C. 30: We need to address 1 GB of physical memory

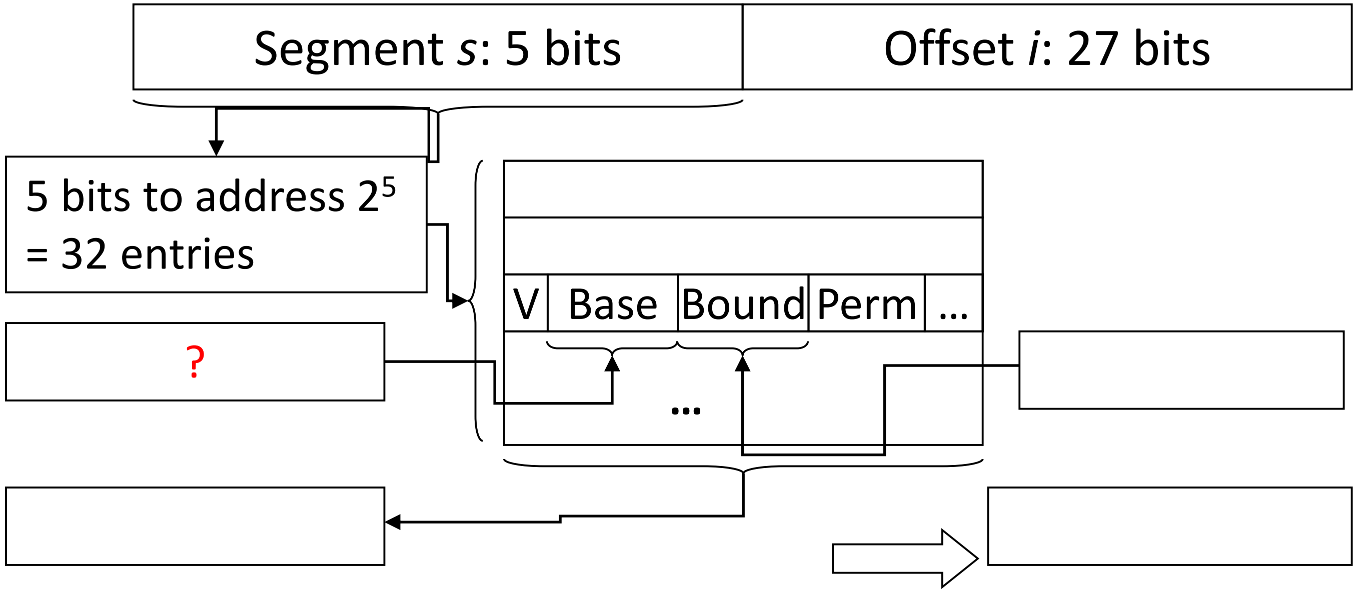
D. 27: We need to address up to the maximum offset

Example of Sizing the Segment Table



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

How many bits do we need for the base?

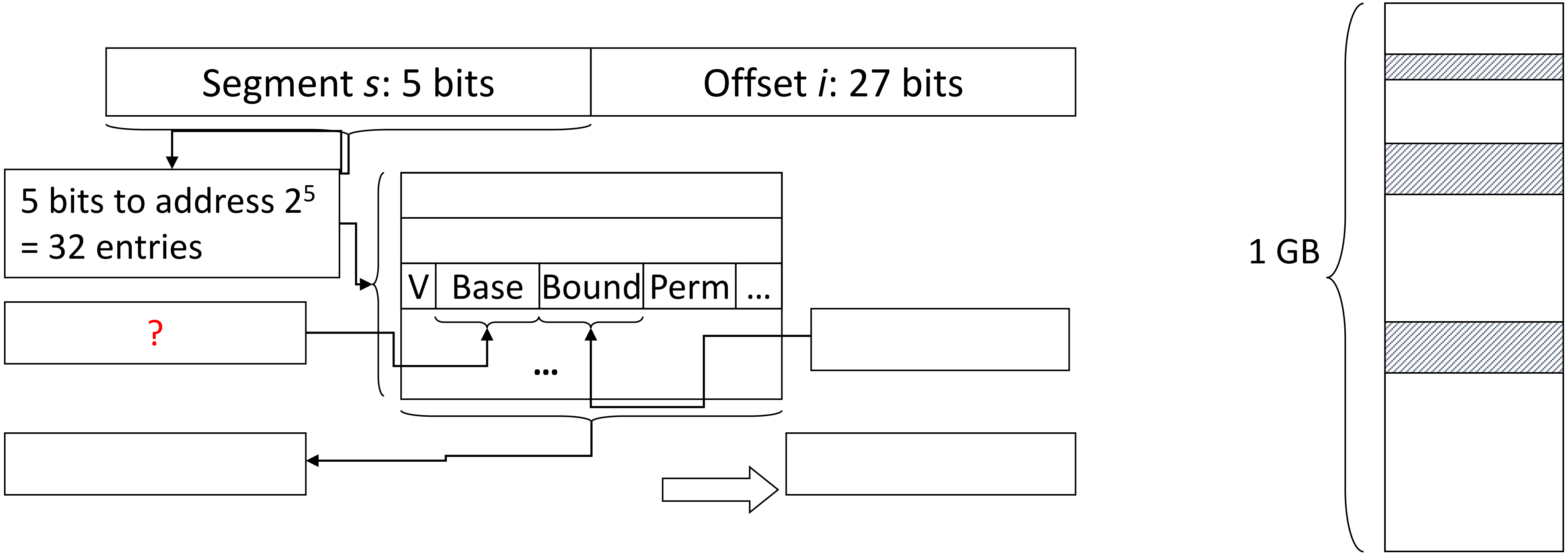


A. 30 bits, to address 1 GB of physical memory.

B. 5 bits, because we have 32 rows in the segment table.

C. 27 bits, to address any potential offset value.

How many bits do we need for the base?

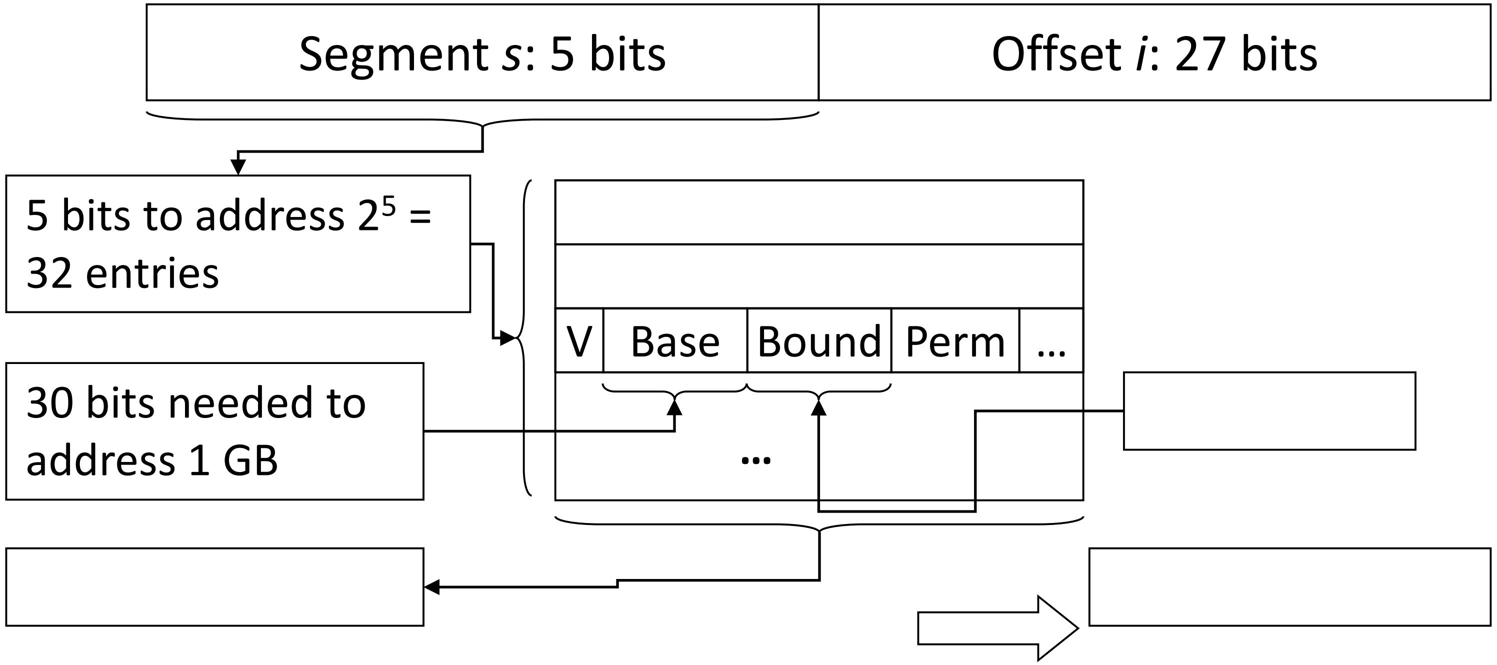


A. 30 bits, to address 1 GB of physical memory.

B. 5 bits, because we have 32 rows in the segment table.

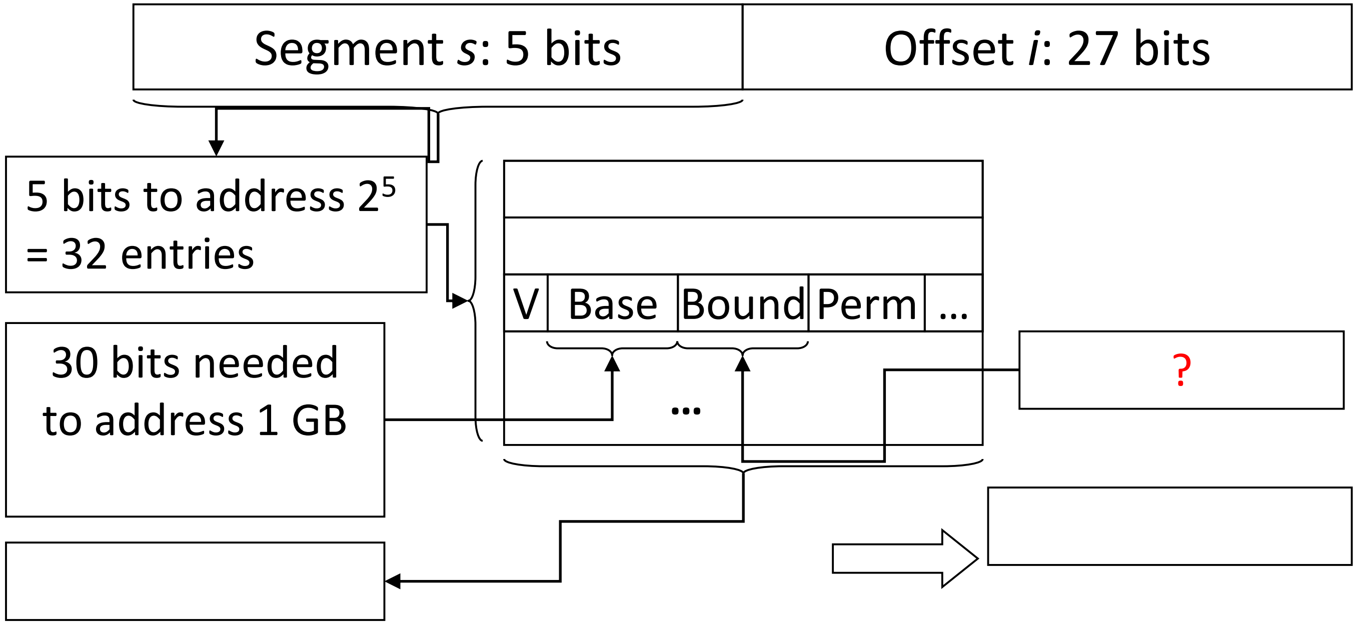
C. 27 bits, to address any potential offset value.

Example of Sizing the Segment Table


Segment *s*: 5 bits
Offset *i*: 27 bits
5 bits to address 2^5 = 32 entries
30 bits needed to address 1 GB
V
Base
Bound
Perm
...
...

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

How many bits do we need for the bound?

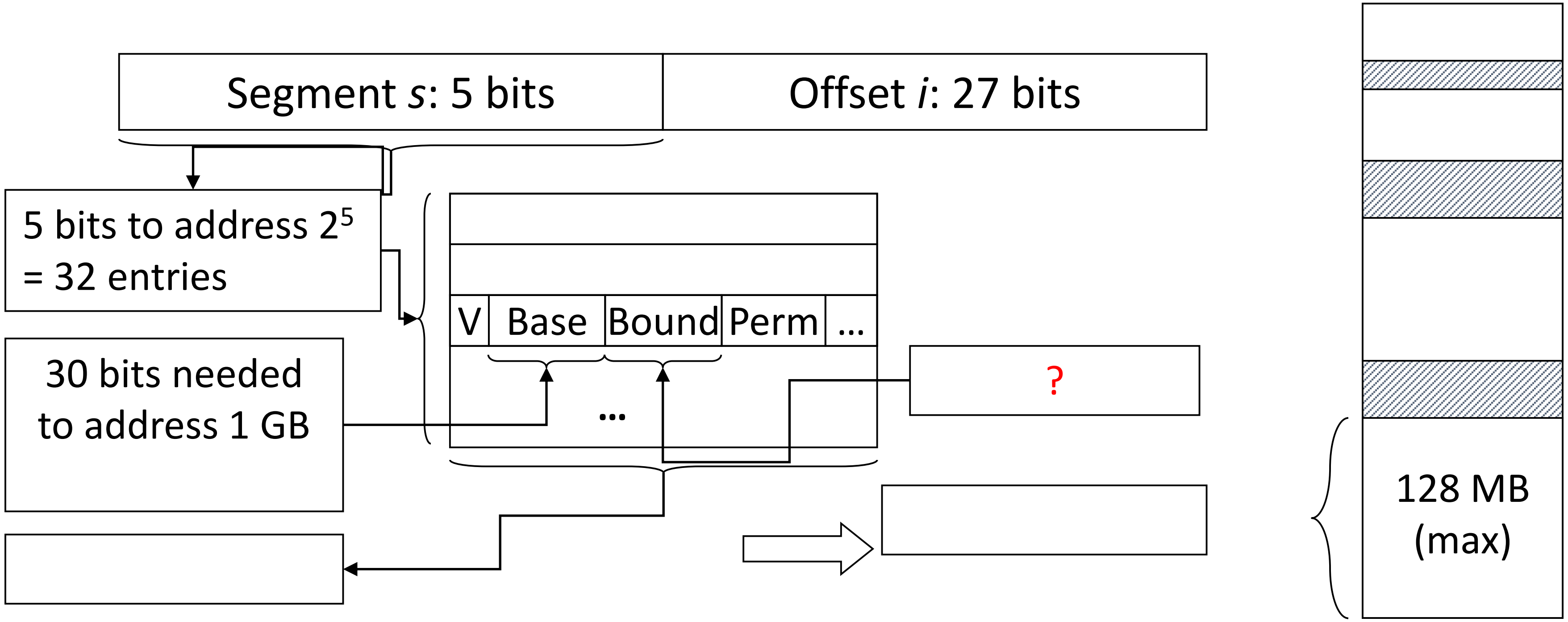

Segment *s*: 5 bits
Offset *i*: 27 bits
5 bits to address 2^5 = 32 entries
30 bits needed to address 1 GB
V
Base
Bound
Perm
...
...
?

A. 5 bits: the size of the segment portion of the virtual address.

B. 27 bits: the size of the offset portion of the virtual address.

C. 32 bits: the size of the virtual address.

How many bits do we need for the bound?

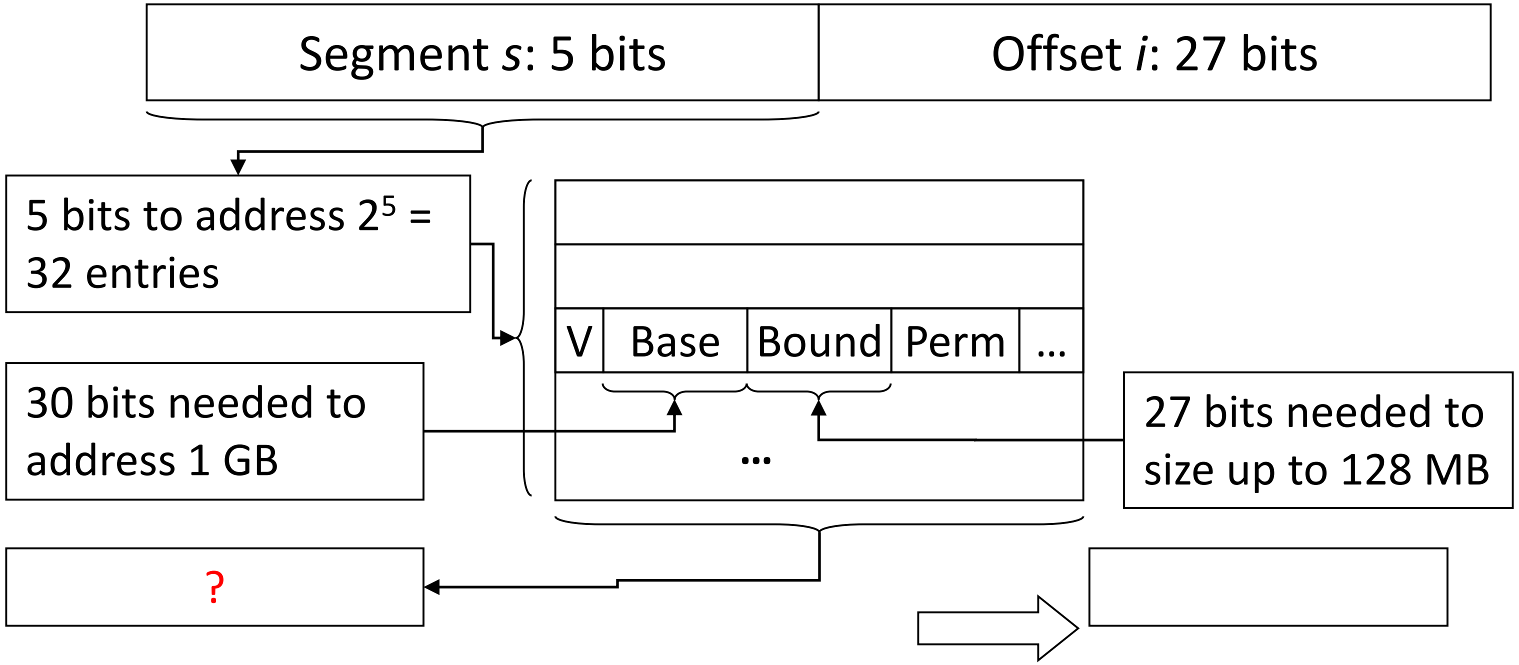


A. 5 bits: the size of the segment portion of the virtual address.

B. 27 bits: the size of the offset portion of the virtual address.

C. 32 bits: the size of the virtual address.

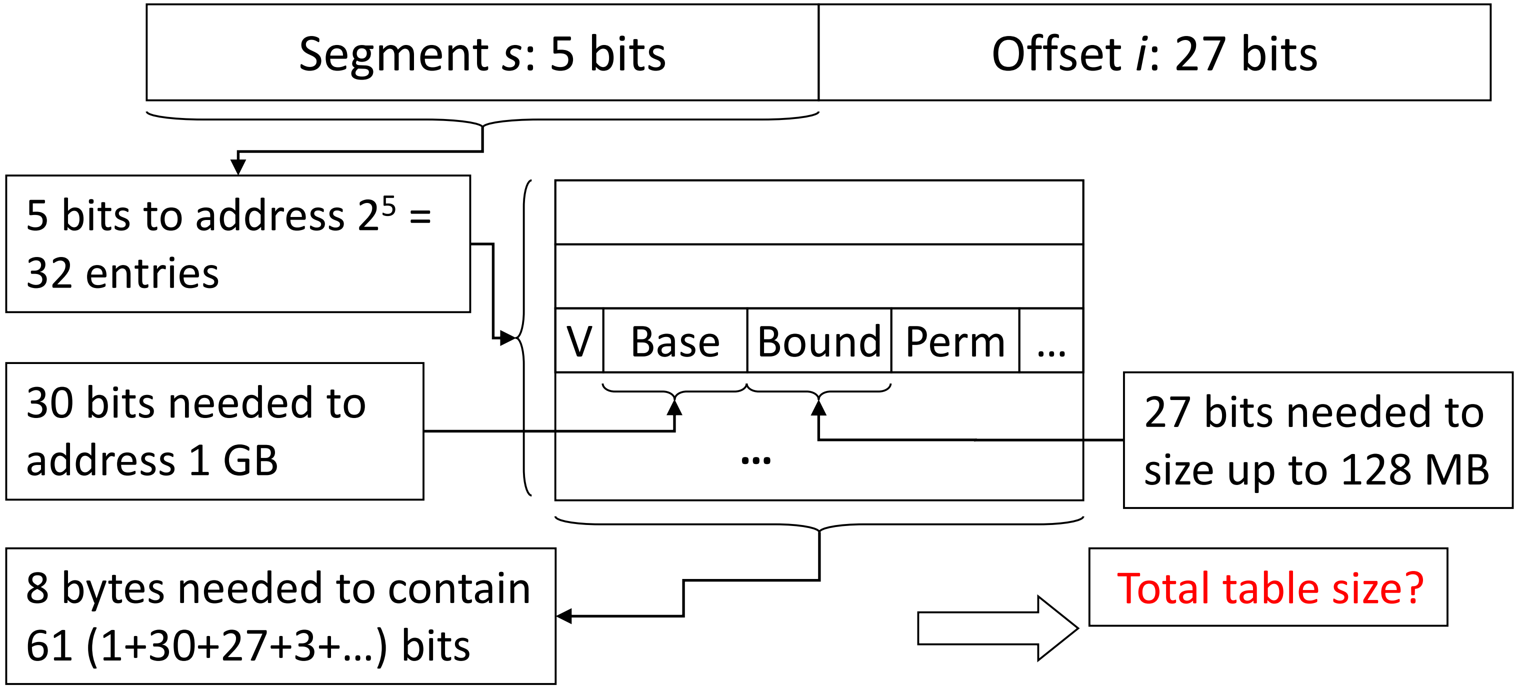
Example of Sizing the Segment Table



Segment *s*: 5 bits
Offset *i*: 27 bits
5 bits to address 2^5 = 32 entries
30 bits needed to address 1 GB
?
V
Base
Bound
Perm
...
...
27 bits needed to size up to 128 MB

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

Example of Sizing the Segment Table



Segment *s*: 5 bits

Offset *i*: 27 bits

5 bits to address 2^5 = 32 entries

30 bits needed to address 1 GB

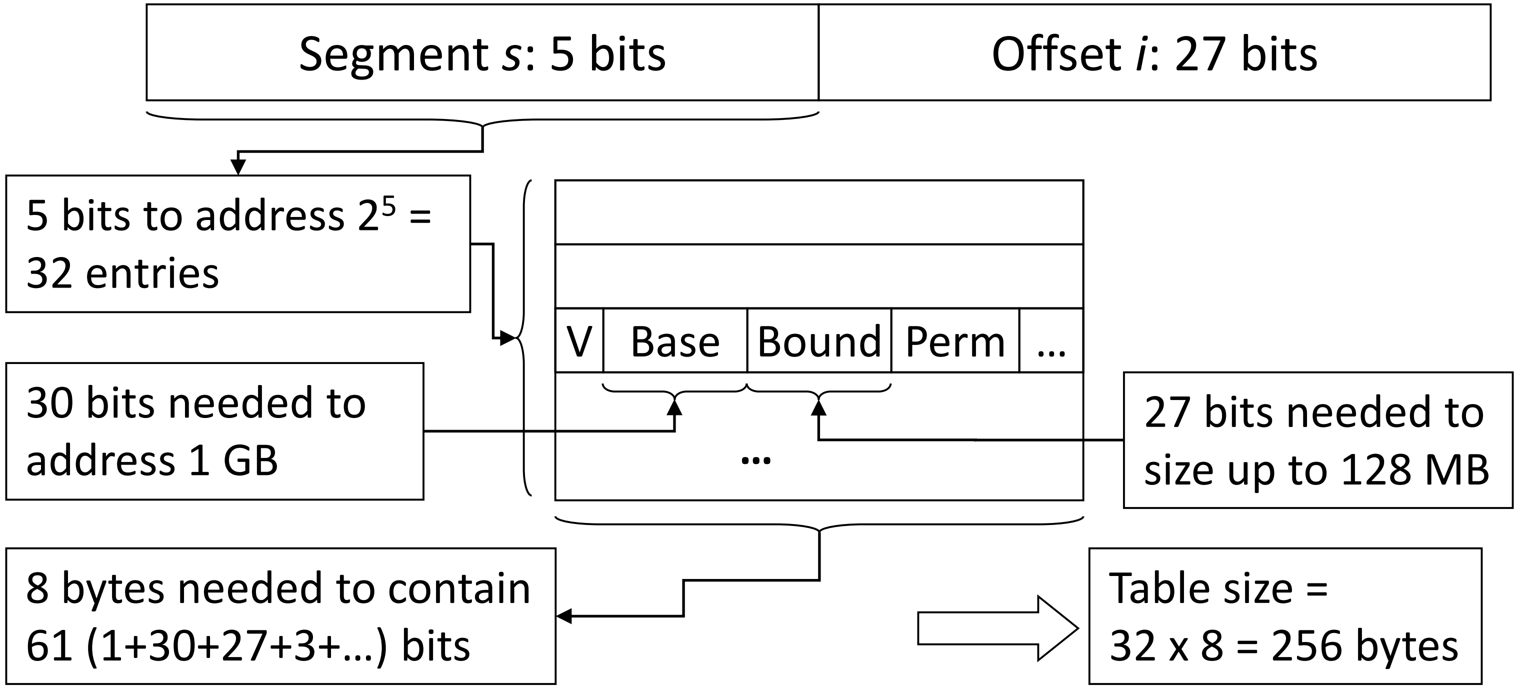
27 bits needed to size up to 128 MB

8 bytes needed to contain 61 (1+30+27+3+…) bits

Total table size?

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

Example of Sizing the Segment Table



Segment *s*: 5 bits
Offset *i*: 27 bits
5 bits to address 2^5 = 32 entries
V
Base
Bound
Perm
…
30 bits needed to address 1 GB
27 bits needed to size up to 128 MB
…
8 bytes needed to contain 61 (1+30+27+3+…) bits
Table size = 32 x 8 = 256 bytes

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

Pros and Cons of Segmentation

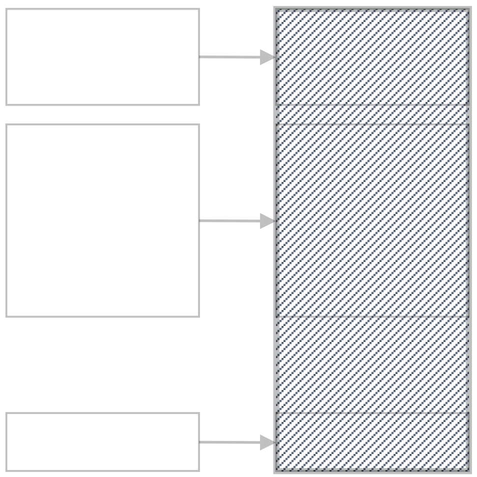
- Pro: Each segment can be
  - located independently
  - separately protected
  - grown/shrunk independently

- Pro: Small segment table size

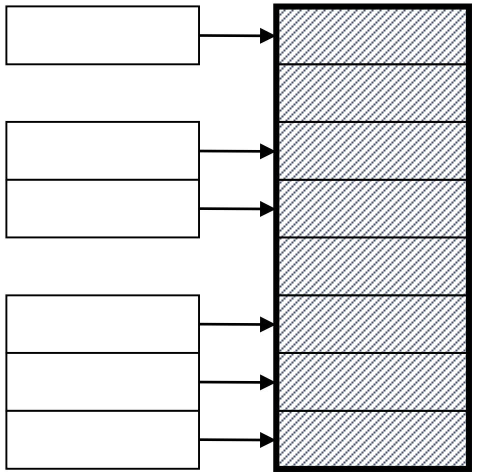
- Con: Variable-size allocation
  - Difficult to find large enough gaps (or “best” gap) in physical memory
  - External fragmentation

Defining Regions - Two Approaches

- Segmentation:
  - Partition address space and memory into segments
  - Segments have varying sizes



- Paging:
  - Partition address space and memory into pages
  - Pages are a constant, fixed size

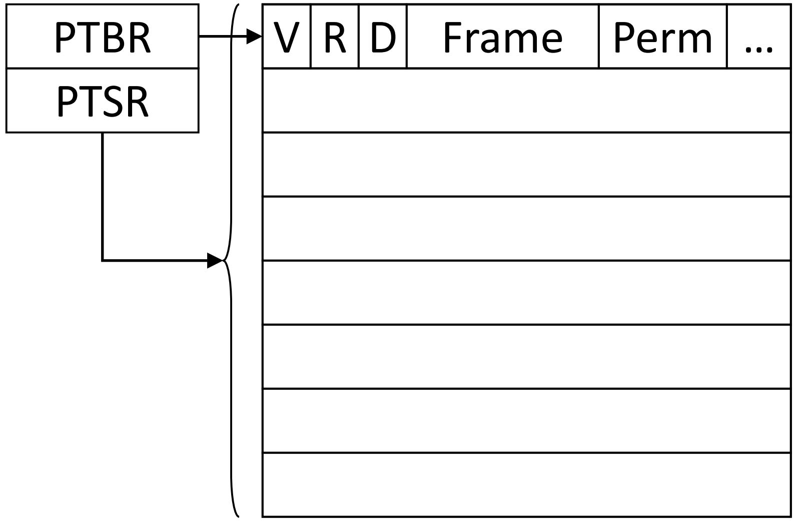


Paging Vocabulary

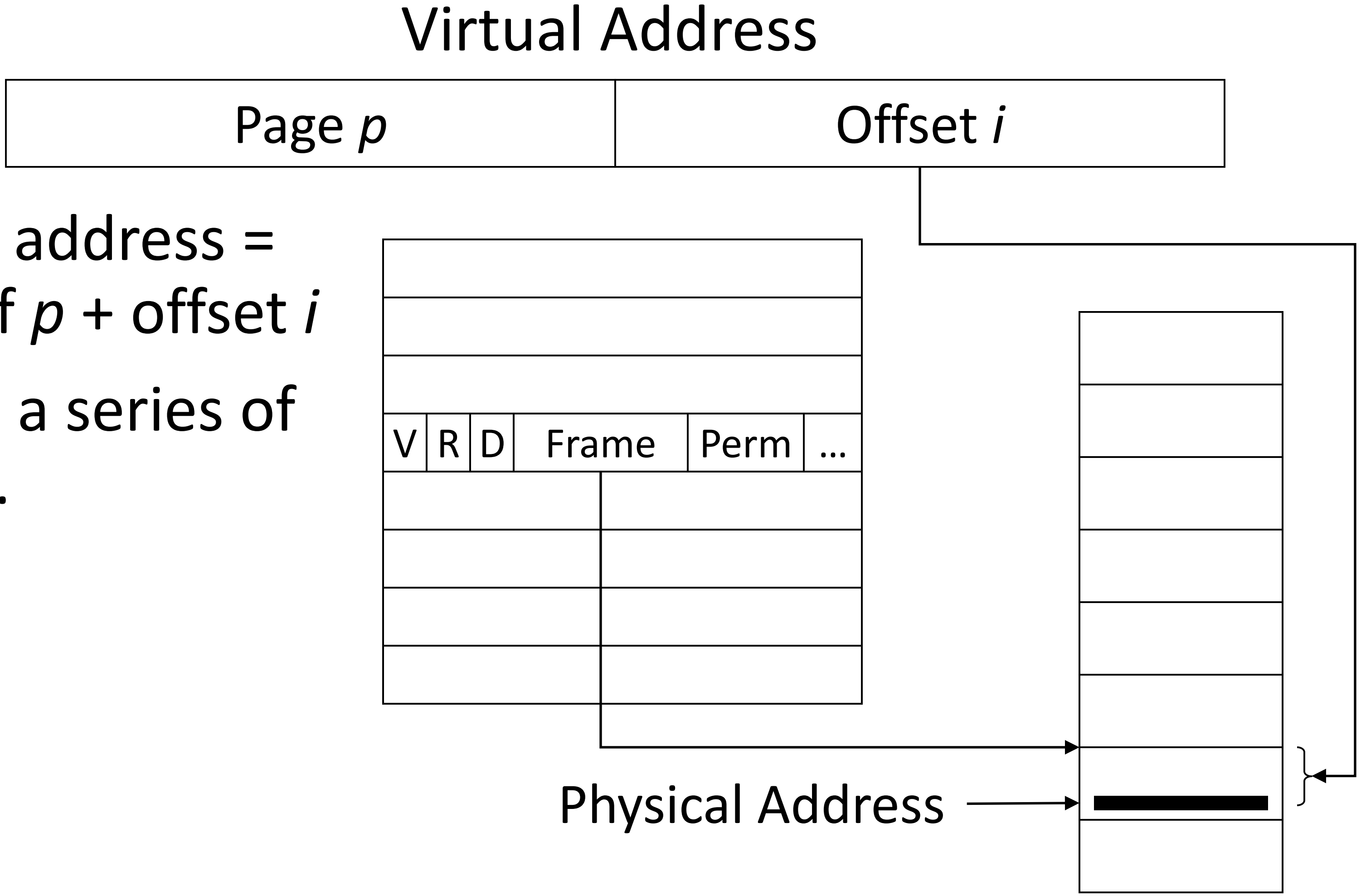
- For each process, the virtual address space is divided into fixed-size pages.
- For the system, the physical memory is divided into fixed-size frames.
- The size of a page is equal to that of a frame.
  - Often 4 KB in practice.
  - Some CPUs allow for small and large pages at the same time.

Page Table

- One table per process
- Table parameters in memory
  - Page table base register
  - Page table size register
- Table entry elements
  - V: valid bit
  - R: referenced bit
  - D: dirty bit
  - Frame: location in phy mem
  - Perm: access permissions

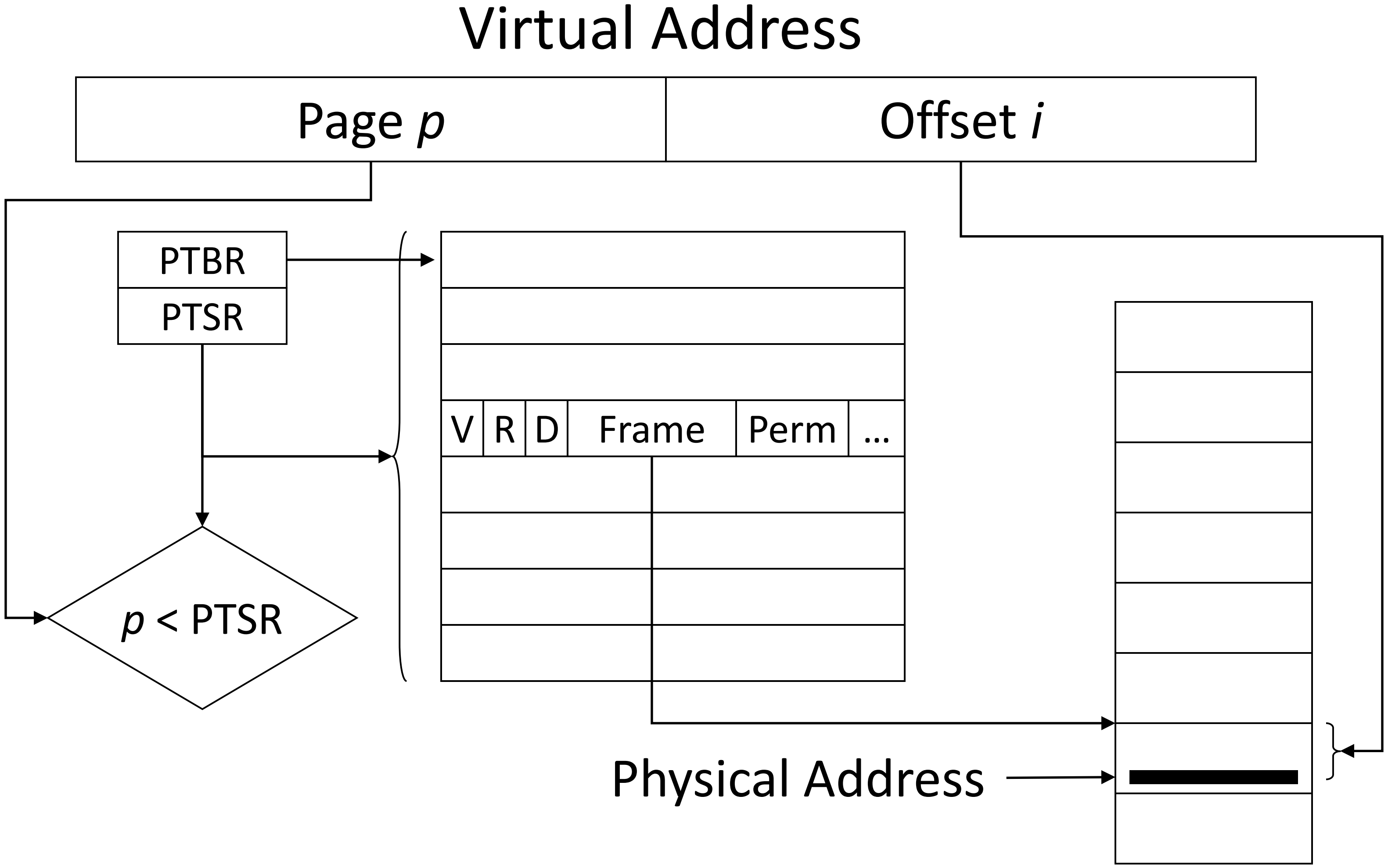


Address Translation

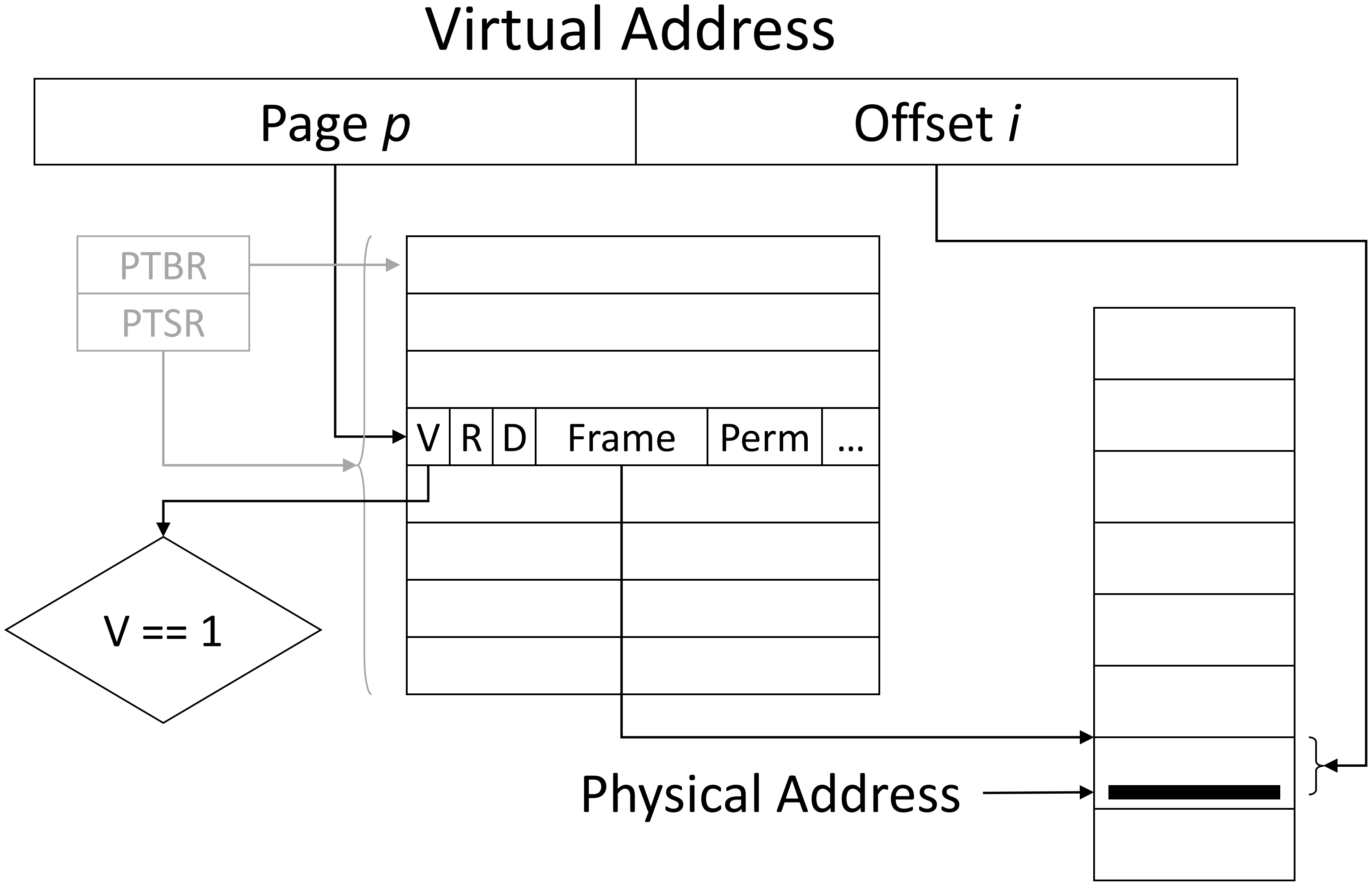


- Physical address = frame of *p* + offset *i*
- First, do a series of checks…

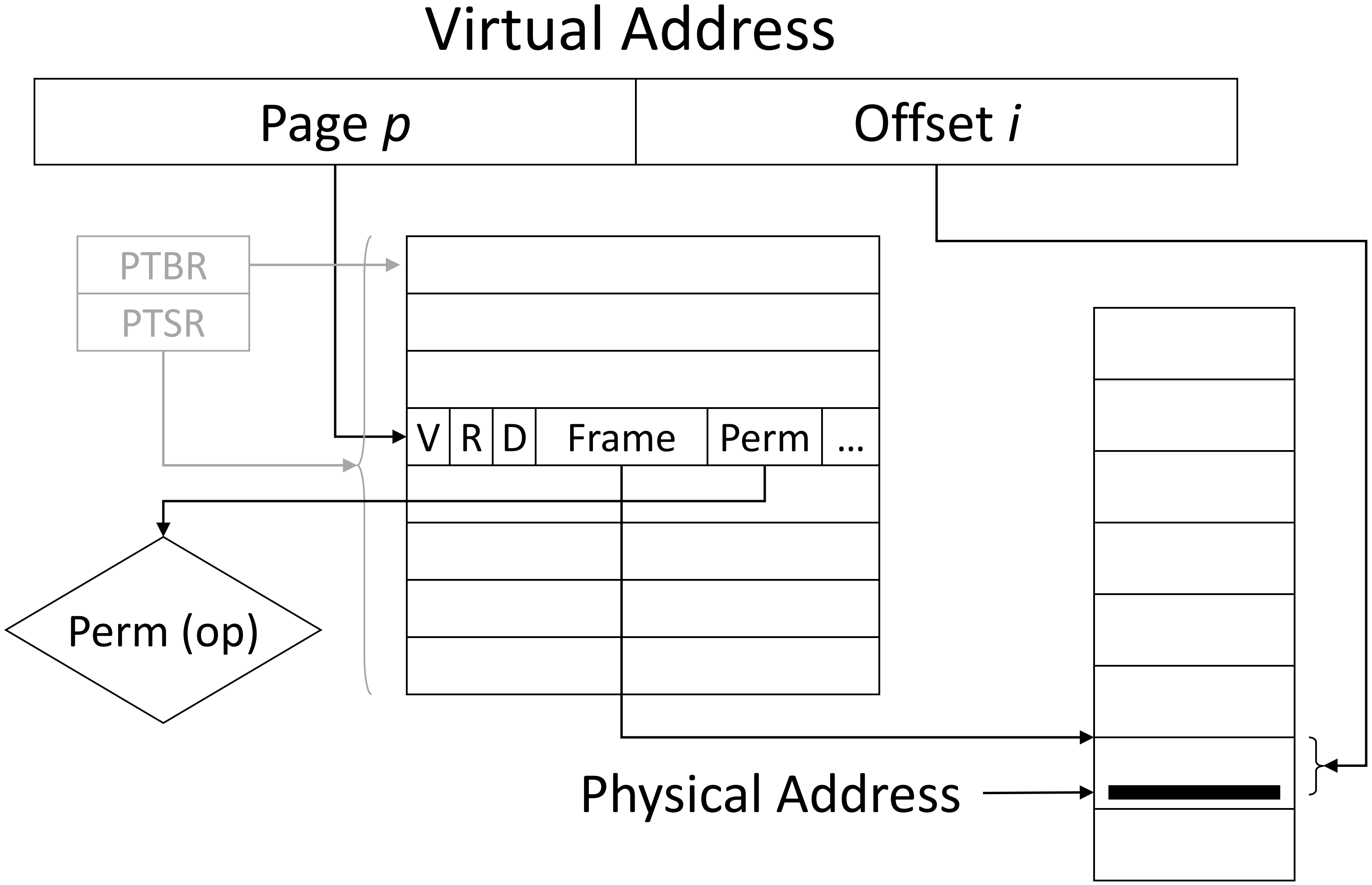
Check if Page *p* is Within Range



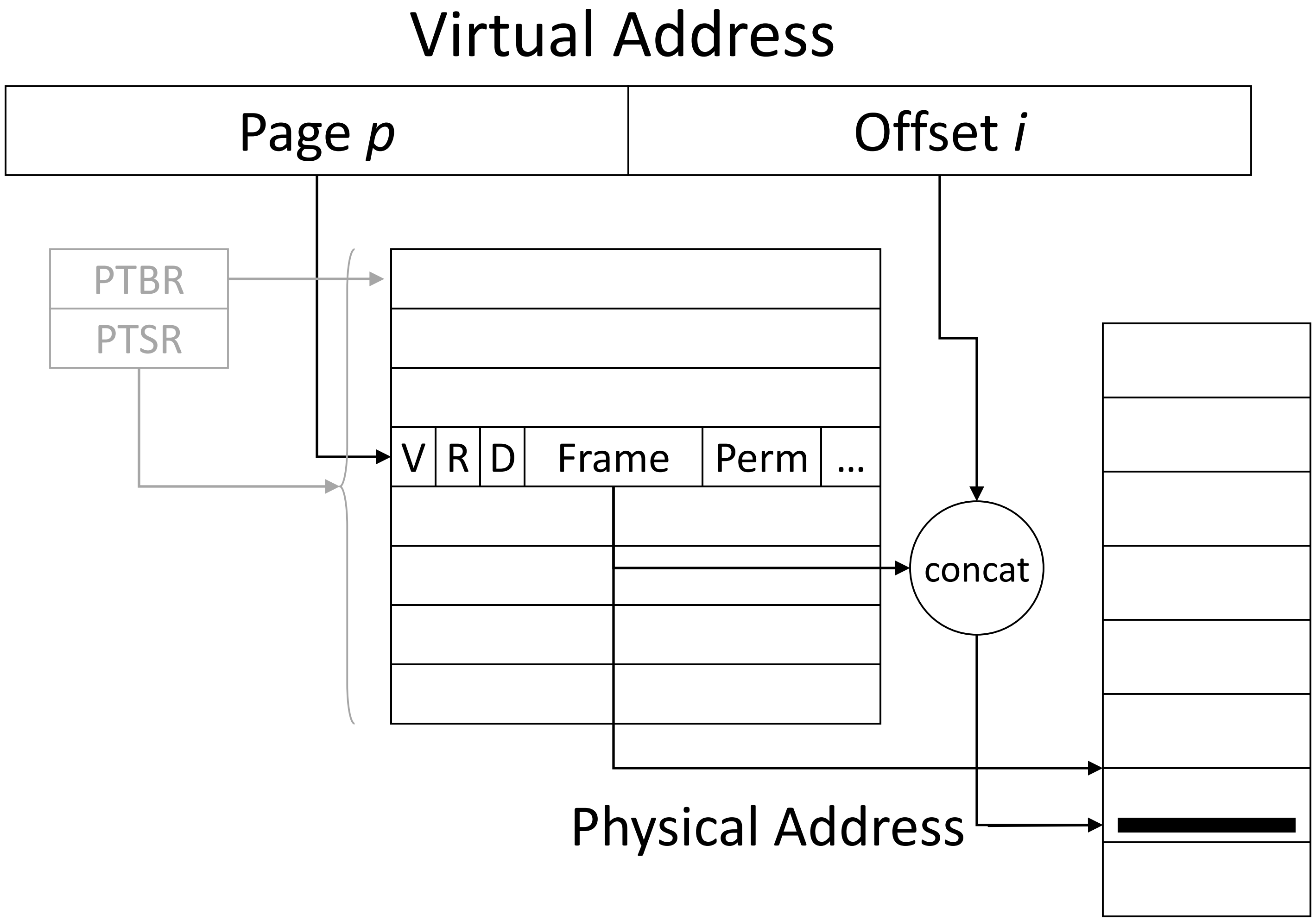
Check if Page Table Entry *p* is Valid



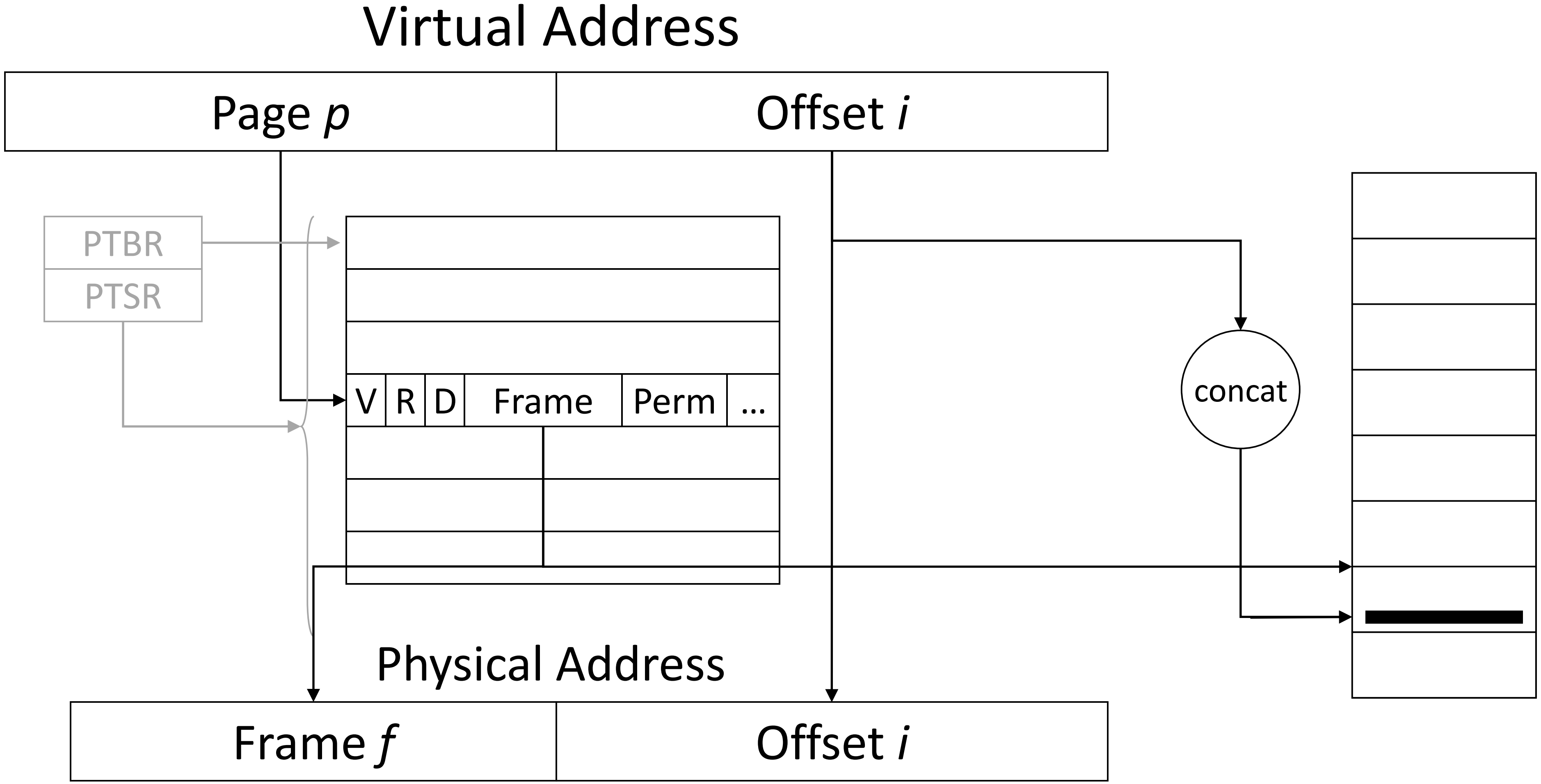
Check if Operation is Permitted



Translate Address

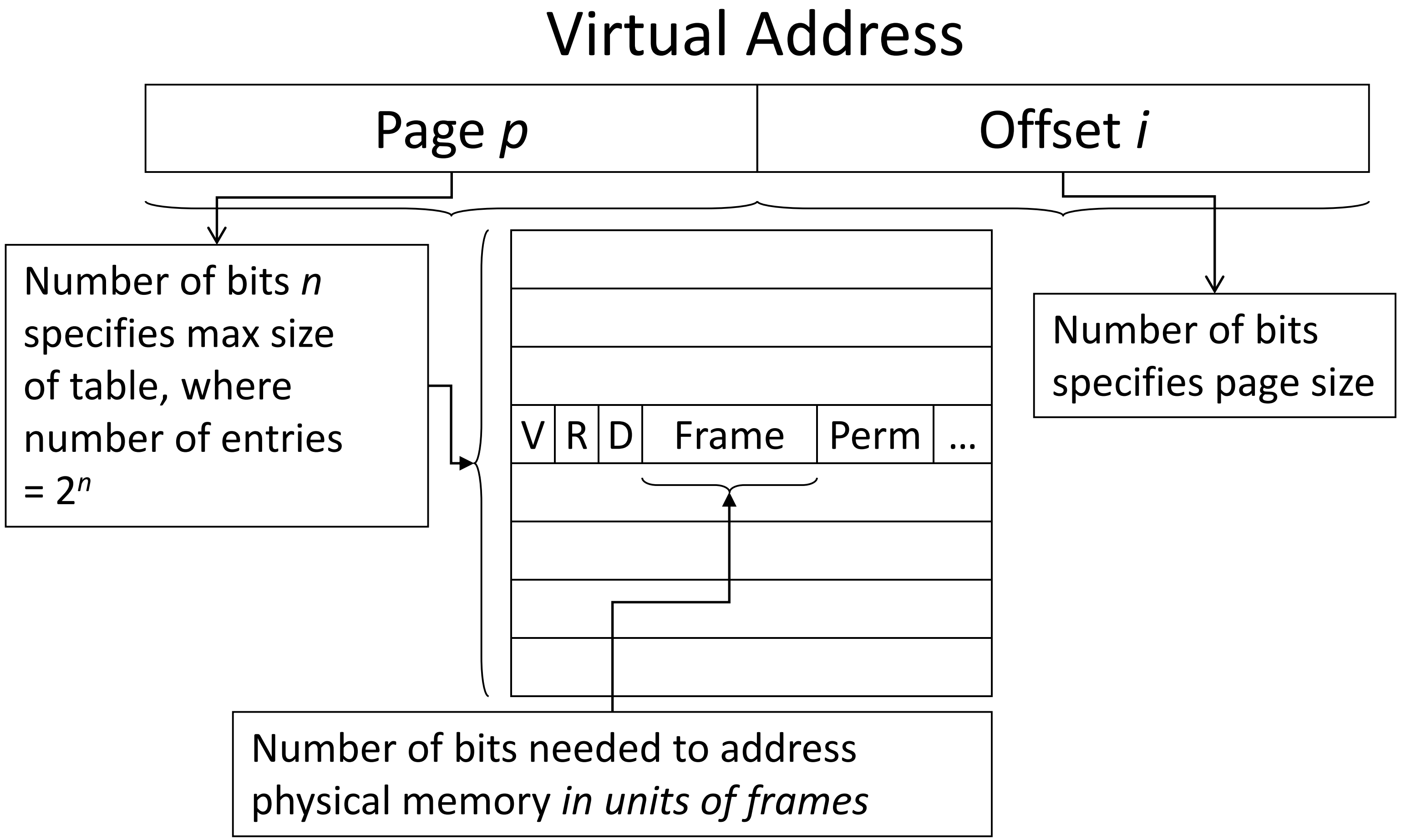


Physical Address by Concatenation



Frames are all the same size. Only need to store the *frame number* in the table, not exact address!

Sizing the Page Table



Virtual Address

Page *p*	Offset *i*

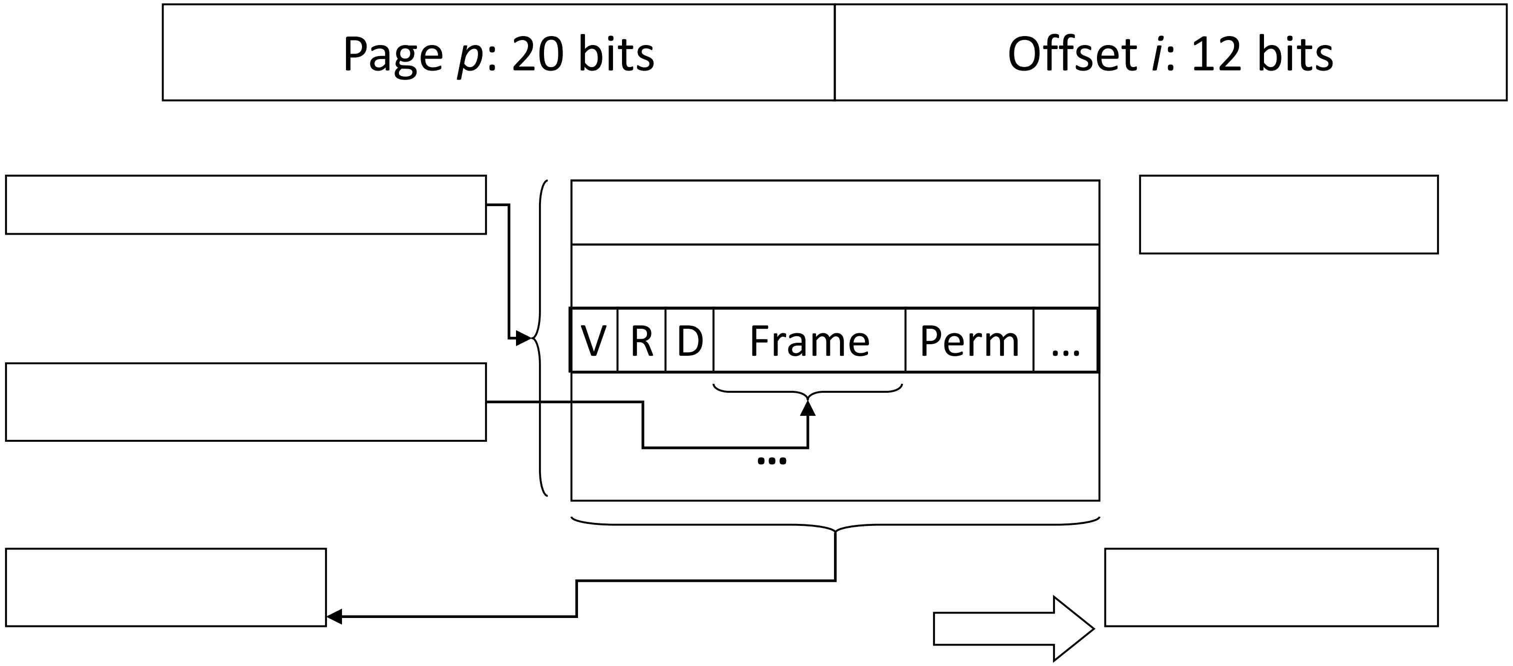
Number of bits *n* specifies max size of table, where number of entries = 2^n

V	R	D	Frame	Perm	…

Number of bits specifies page size

Number of bits needed to address physical memory *in units of frames*

Example of Sizing the Page Table



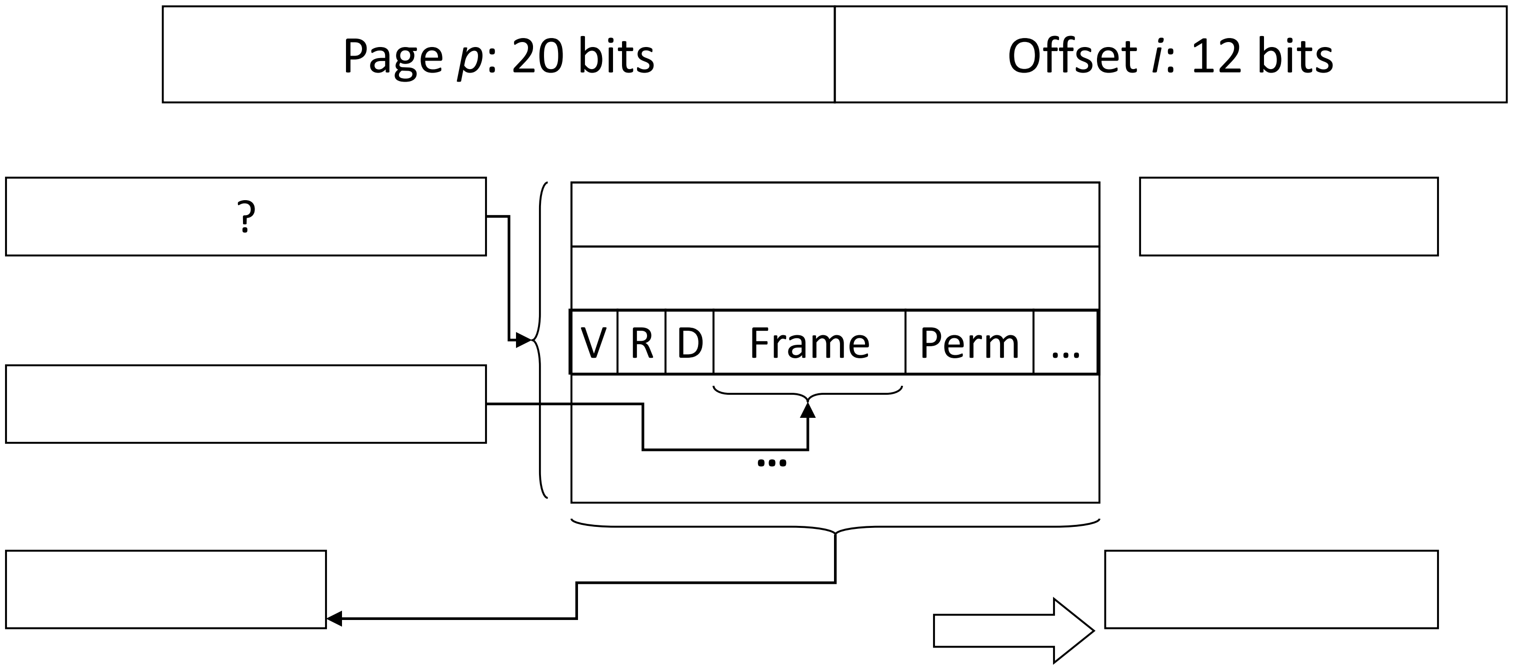
Page *p*: 20 bits | Offset *i*: 12 bits

V | R | D | Frame | Perm | …

…

- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset

Example of Sizing the Page Table



- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset

How many entries (rows) will there be in this page table?

A. 2^{12}, because that’s how many the offset field can address

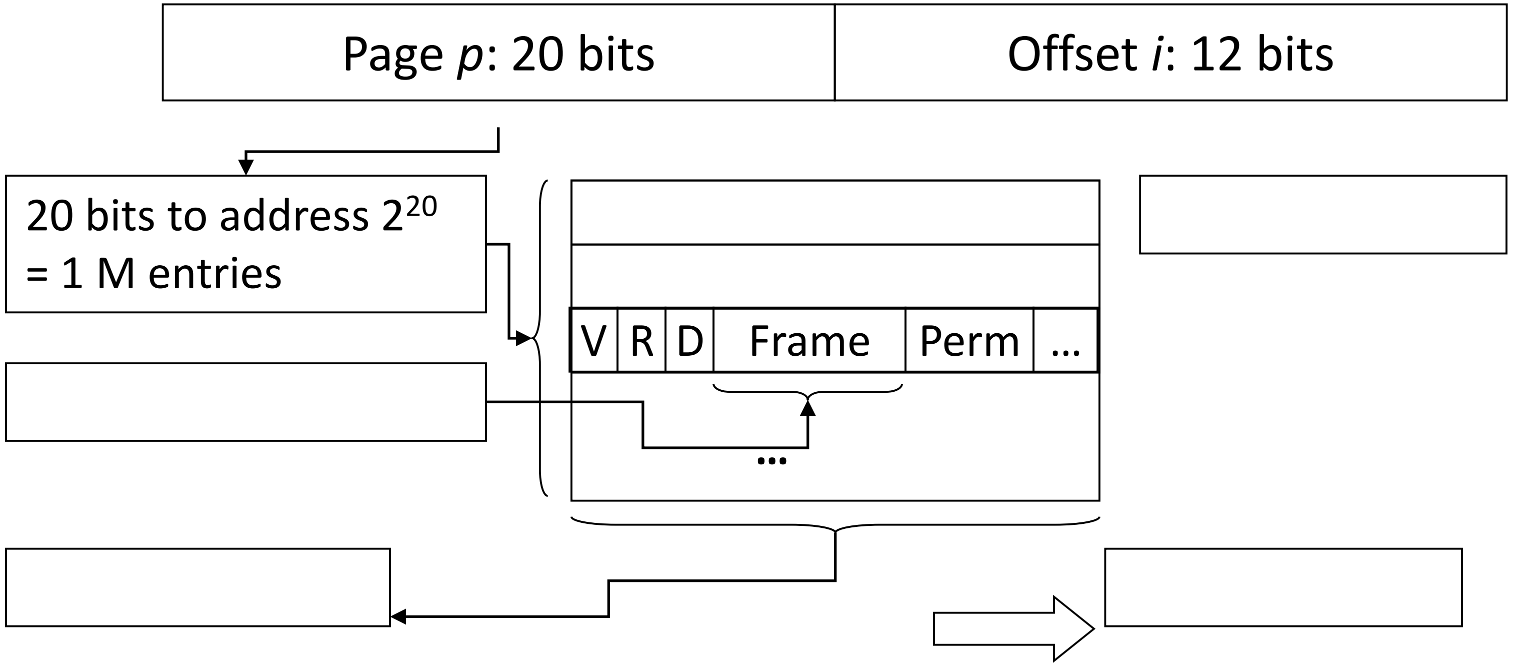
B. 2^{20}, because that’s how many the page field can address

C. 2^{30}, because that’s how many we need to address 1 GB

D. 2^{32}, because that’s the size of the entire address space

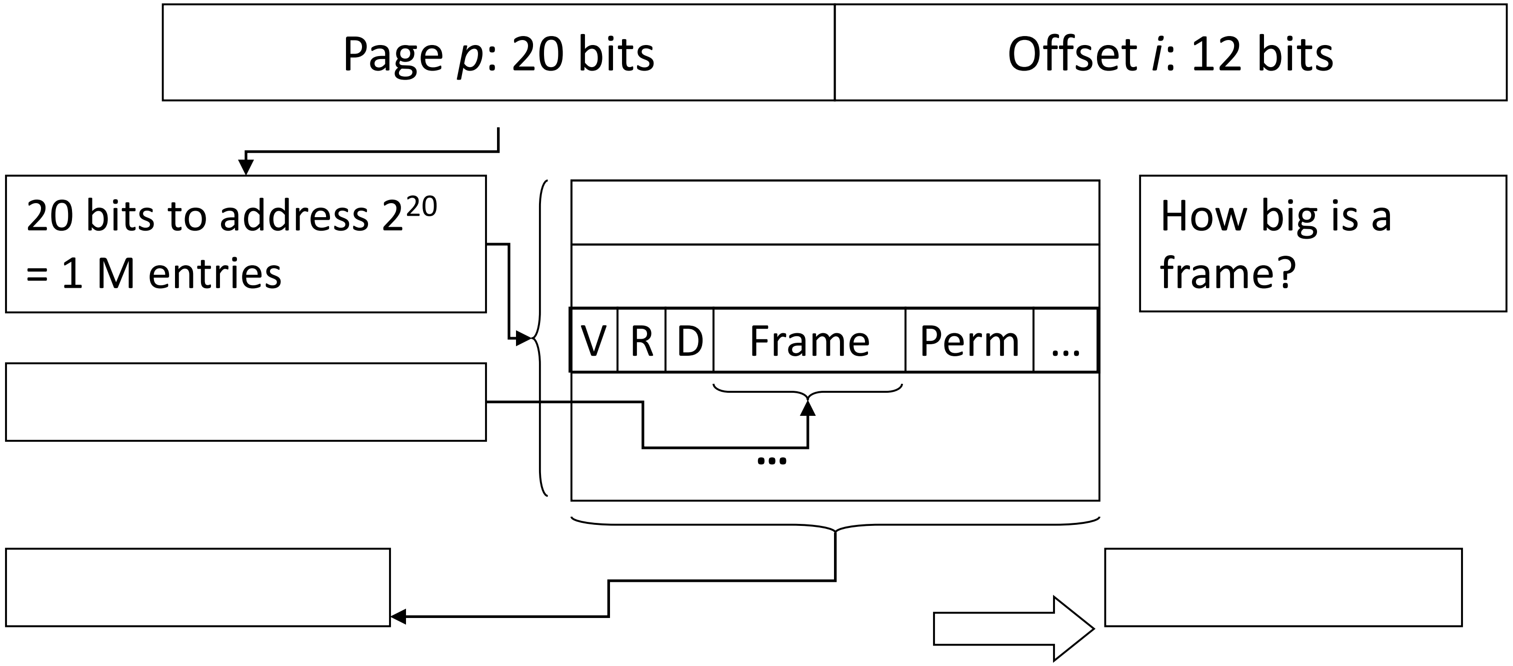
- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset

Example of Sizing the Page Table



- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset

Example of Sizing the Page Table



- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset

What will be the frame size, in bytes?

A. 2^{12}, because that’s how many bytes the offset field can address

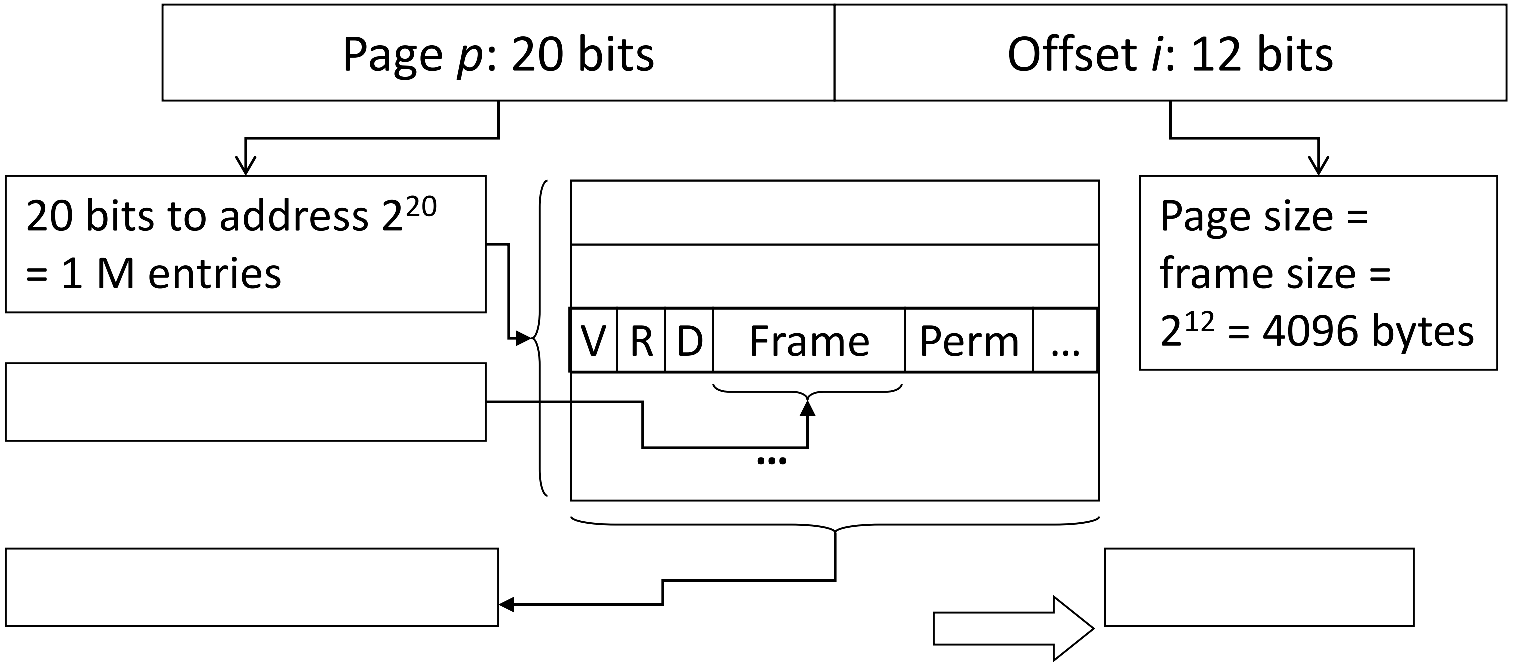
B. 2^{20}, because that’s how many bytes the page field can address

C. 2^{30}, because that’s how many bytes we need to address 1 GB

D. 2^{32}, because that’s the size of the entire address space

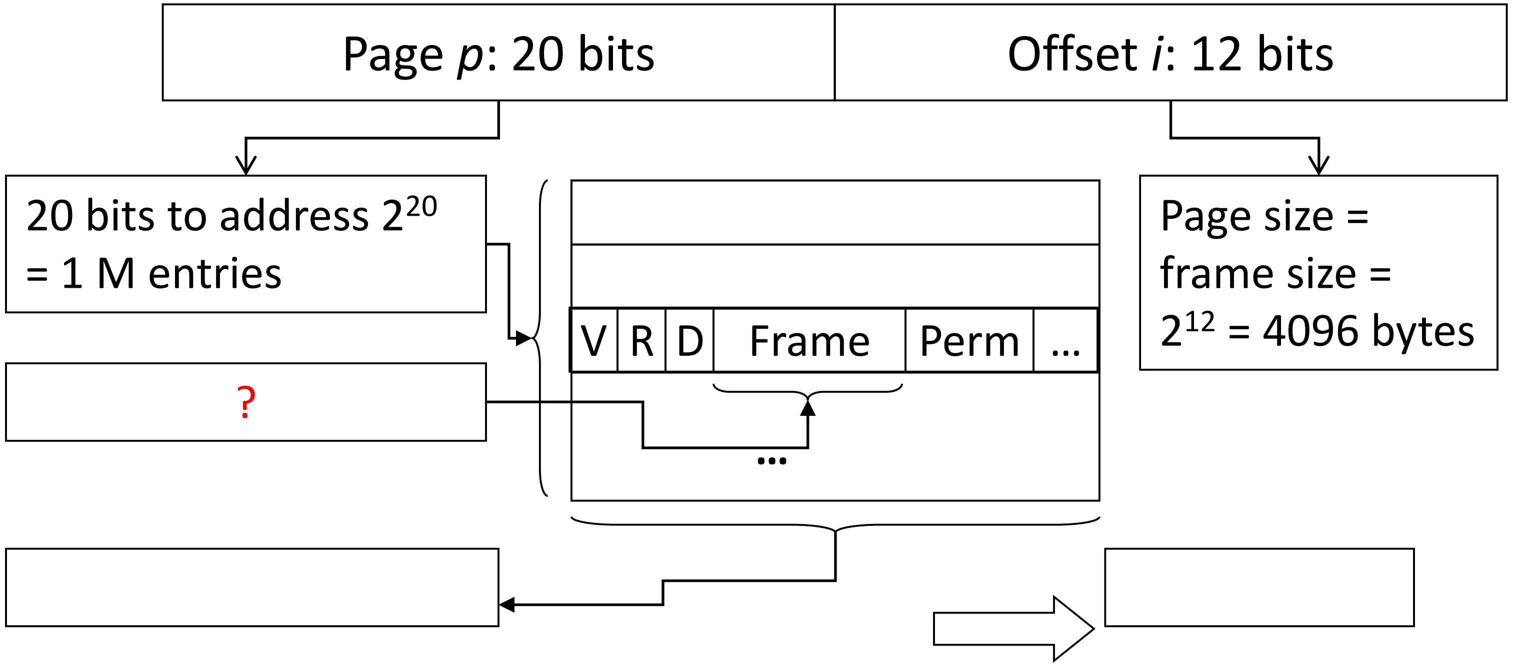
- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset

Example of Sizing the Page Table


Page *p*: 20 bits
Offset *i*: 12 bits
20 bits to address 2^{20} = 1 M entries
V
R
D
Frame
Perm
…
…
Page size = frame size = 2^{12} = 4096 bytes

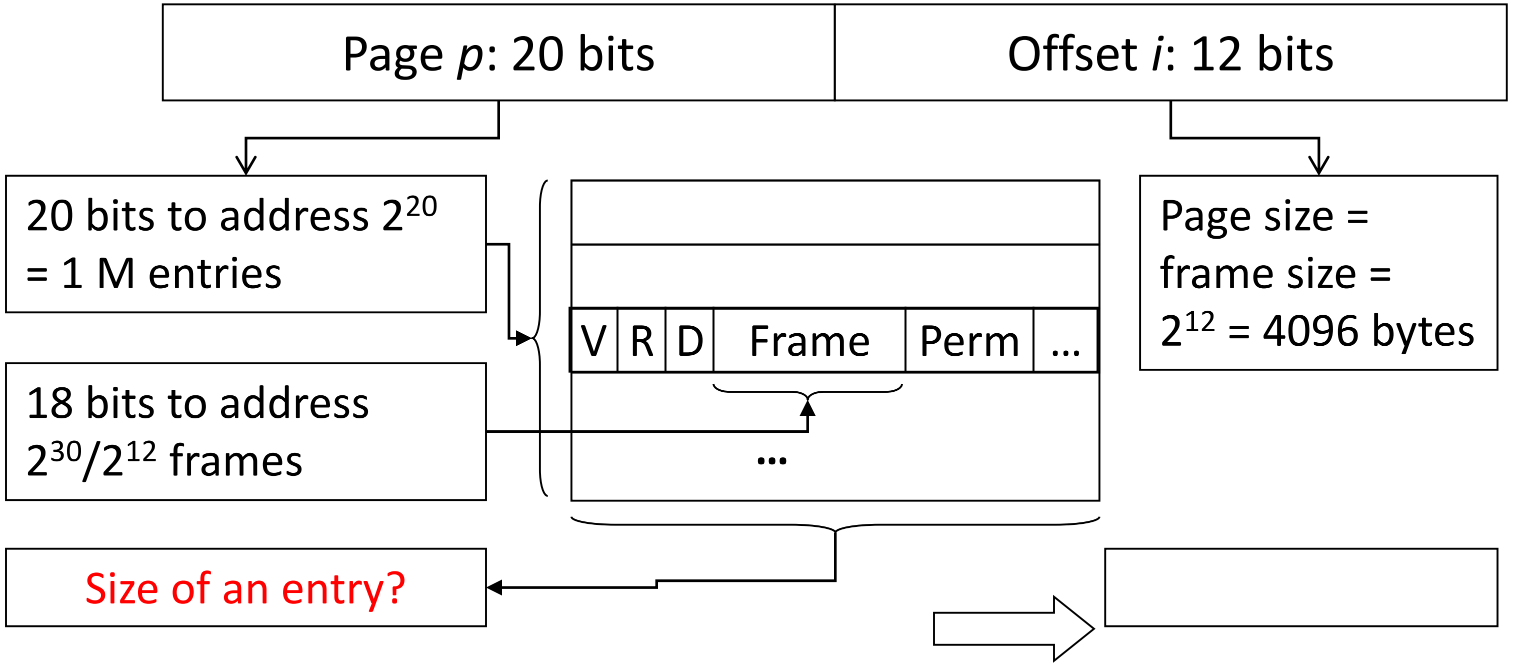
- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset

How many bits do we need to store the frame number?


Page *p*: 20 bits
Offset *i*: 12 bits
20 bits to address 2^{20} = 1 M entries
?
V
R
D
Frame
Perm
...
...
Page size = frame size = 2^{12} = 4096 bytes

- Given: 32 bit virtual addresses, 1 GB physical memory
    - Address partition: 20 bit page number, 12 bit offset
- A: 12 B: 18 C: 20 D: 30 E: 32

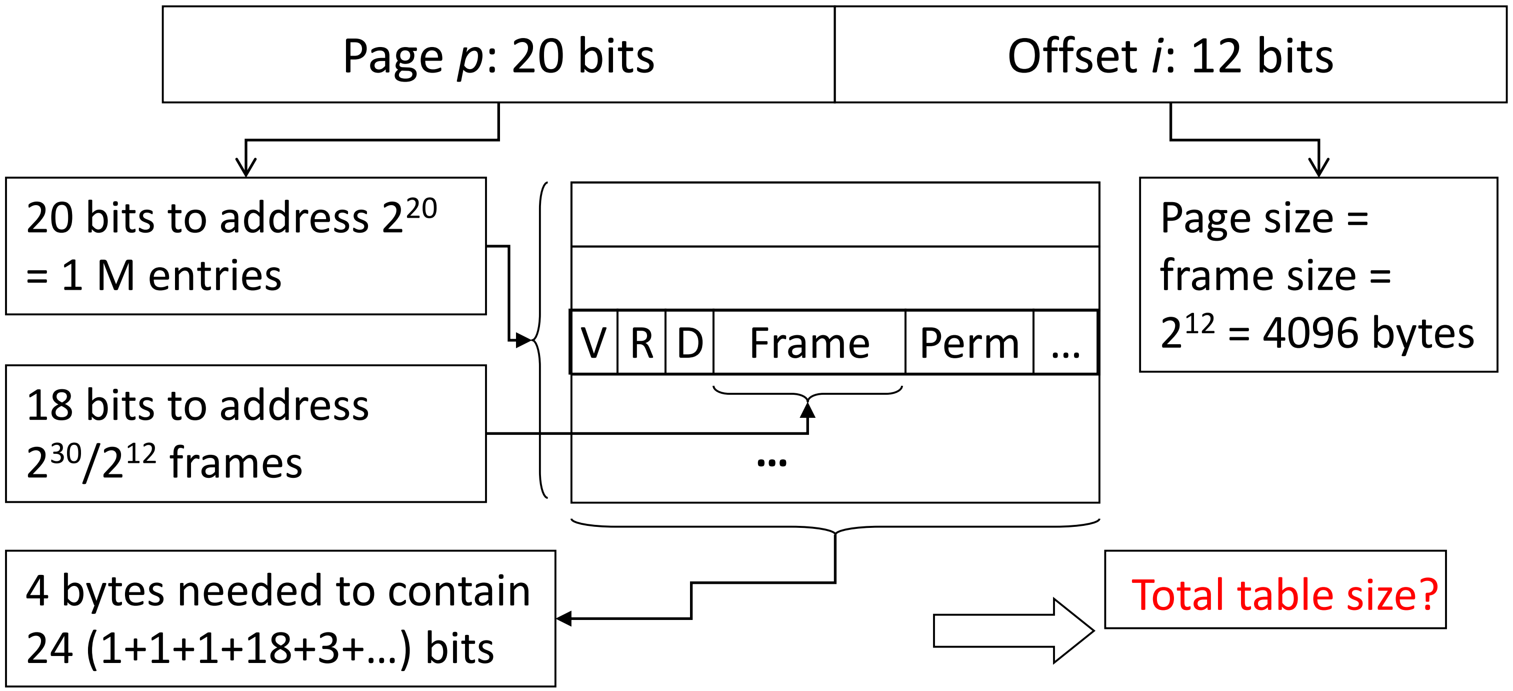
Example of Sizing the Page Table



Page *p*: 20 bits
Offset *i*: 12 bits
20 bits to address 2^{20} = 1 M entries
18 bits to address $2^{30}/2^{12}$ frames
Size of an entry?
V
R
D
Frame
Perm
...
...
Page size = frame size = 2^{12} = 4096 bytes

- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset

Example of Sizing the Page Table



Page *p*: 20 bits

Offset *i*: 12 bits

20 bits to address 2^{20} = 1 M entries

18 bits to address $2^{30}/2^{12}$ frames

4 bytes needed to contain 24 (1+1+1+18+3+…) bits

V	R	D	Frame	Perm	…

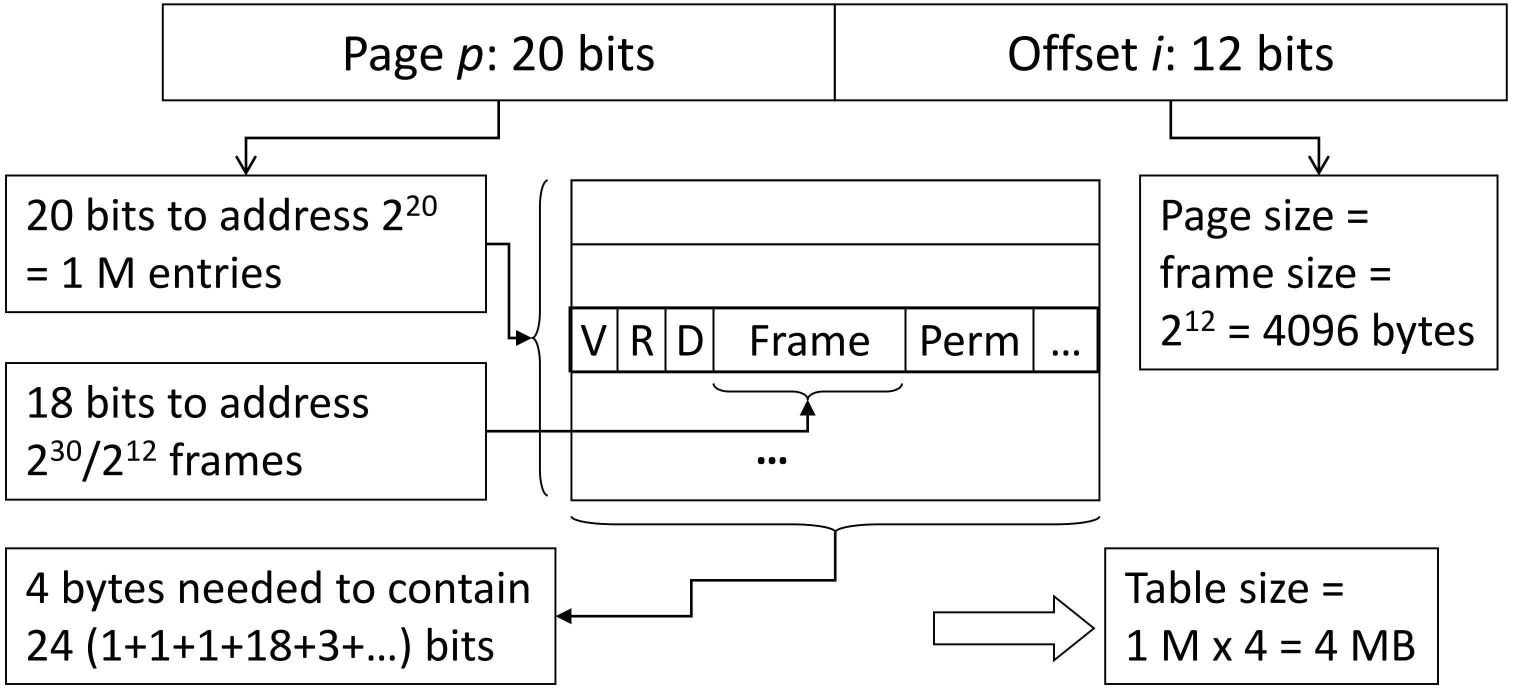
…

Page size = frame size = 2^{12} = 4096 bytes

Total table size?

- Given: 32 bit virtual addresses, 1 GB physical memory
    - Address partition: 20 bit page number, 12 bit offset

Example of Sizing the Page Table



20 bits to address 2^{20} = 1 M entries

18 bits to address $2^{30}/2^{12}$ frames

4 bytes needed to contain 24 (1+1+1+18+3+…) bits

Page size = frame size = 2^{12} = 4096 bytes

Table size = 1 M x 4 = 4 MB

- 4 MB of bookkeeping for *every process*?
  - 200 processes -> 800 MB just to store page tables…

Pros and Cons of Paging

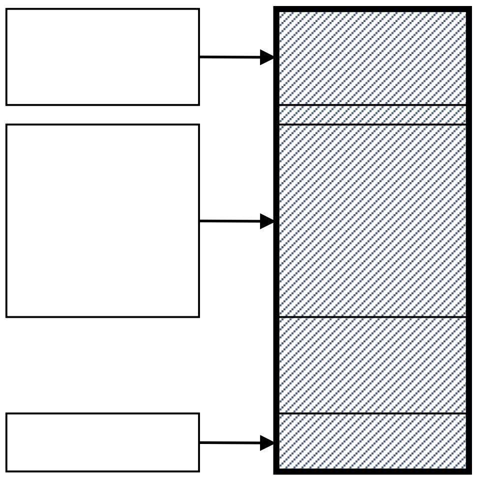
- Pro: Fixed-size pages and frames
  - No external fragmentation
  - No difficult placement decisions

- Con: large table size

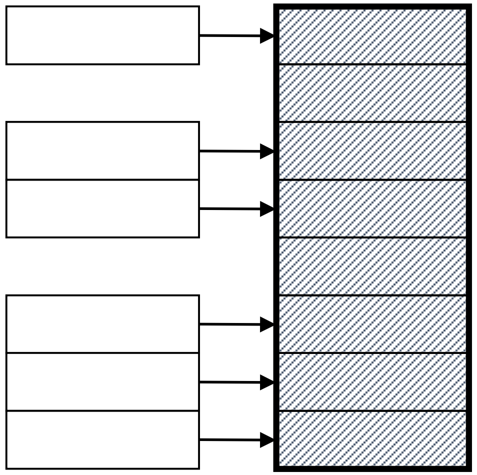
- Con: *maybe* internal fragmentation

Which would you use? Why? Pros/Cons?

A. Segmentation:
   - Partition address space and memory into segments
   - Segments have varying sizes



B. Paging:
   - Partition address space and memory into pages
   - Pages are a constant, fixed size

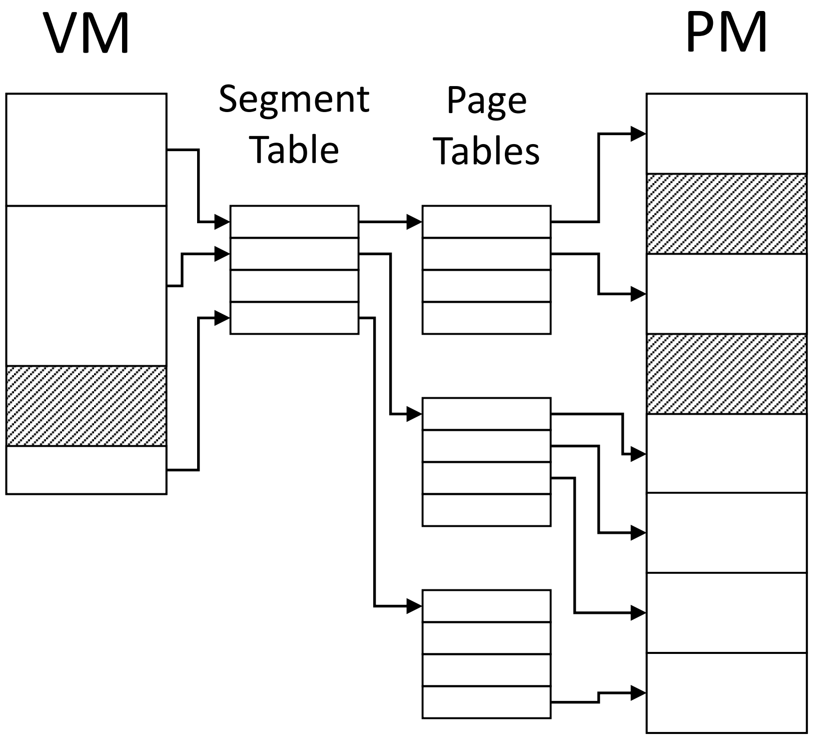


C. Something else (what?)

x86: Hybrid Approach

- Design:
  - Multiple lookups: first in segment table, which points to a page table.
  - Extra level of indirection.

- Reality:
  - All segments are max physical memory size
  - Segments effectively unused, available for "legacy" reasons.
  - (Mostly) disappeared in x86-64



Outstanding Problems

- Mostly considering paging from here on.

1. Page tables are way too big. Most processes don’t need that many pages, can’t justify a huge table.

2. Adding indirection hurts performance.

Outstanding Problems

- Mostly considering paging from here on.

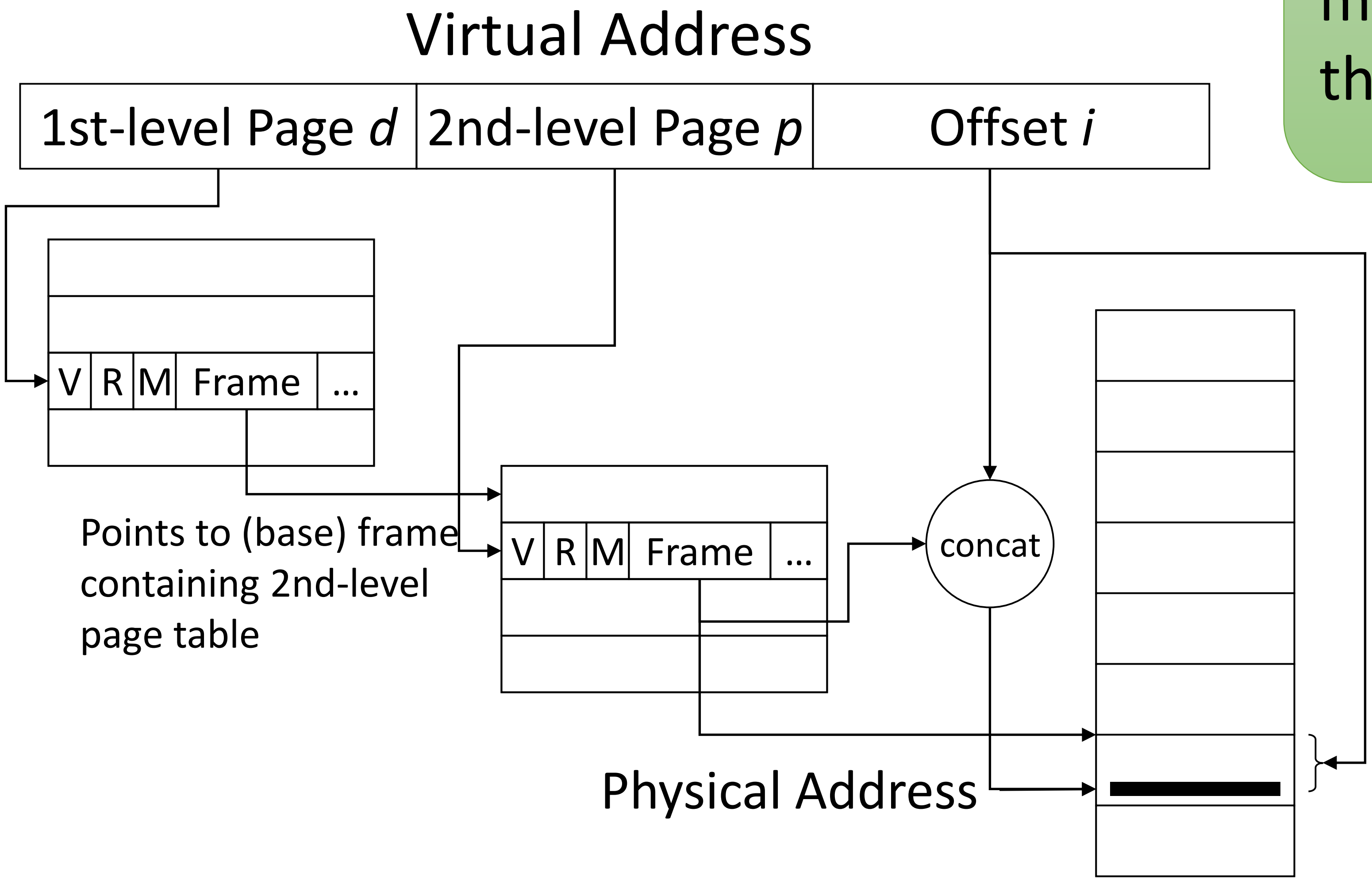
1. Page tables are way too big. Most processes don't need that many pages, can't justify a huge table.

2. Adding indirection hurts performance.

Solution:
MORE indirection!

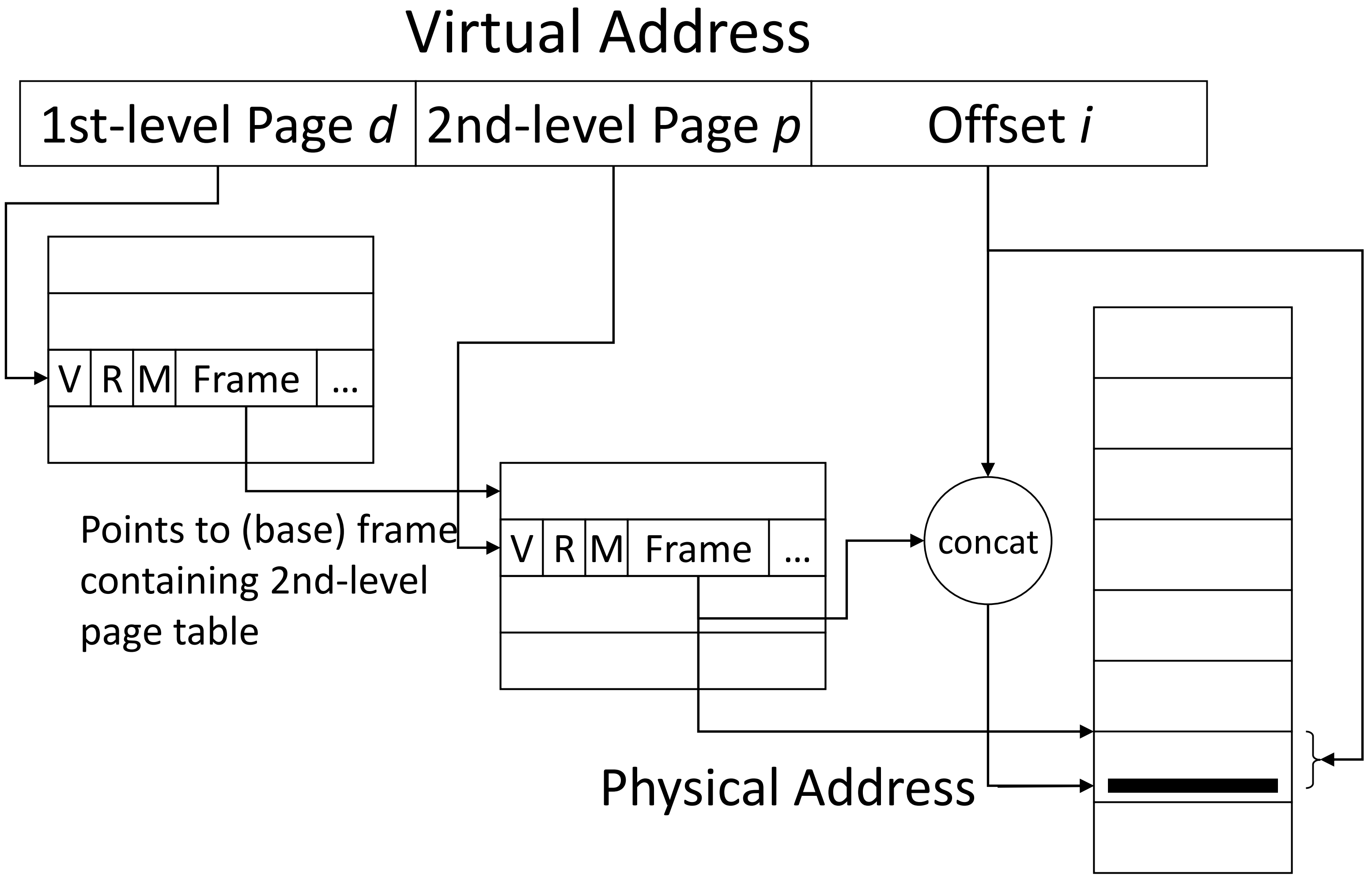
Multi-Level Page Tables

How can using two (or more) page table levels like this reduce the table size?

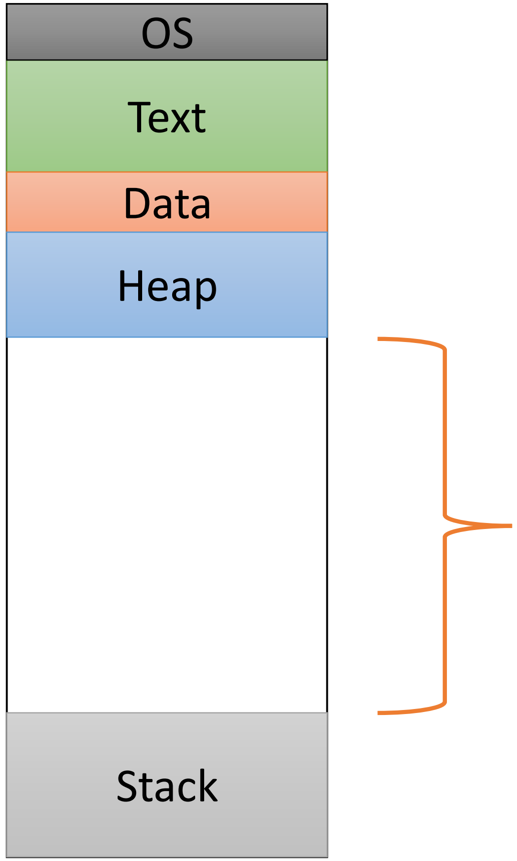


Points to (base) frame containing 2nd-level page table

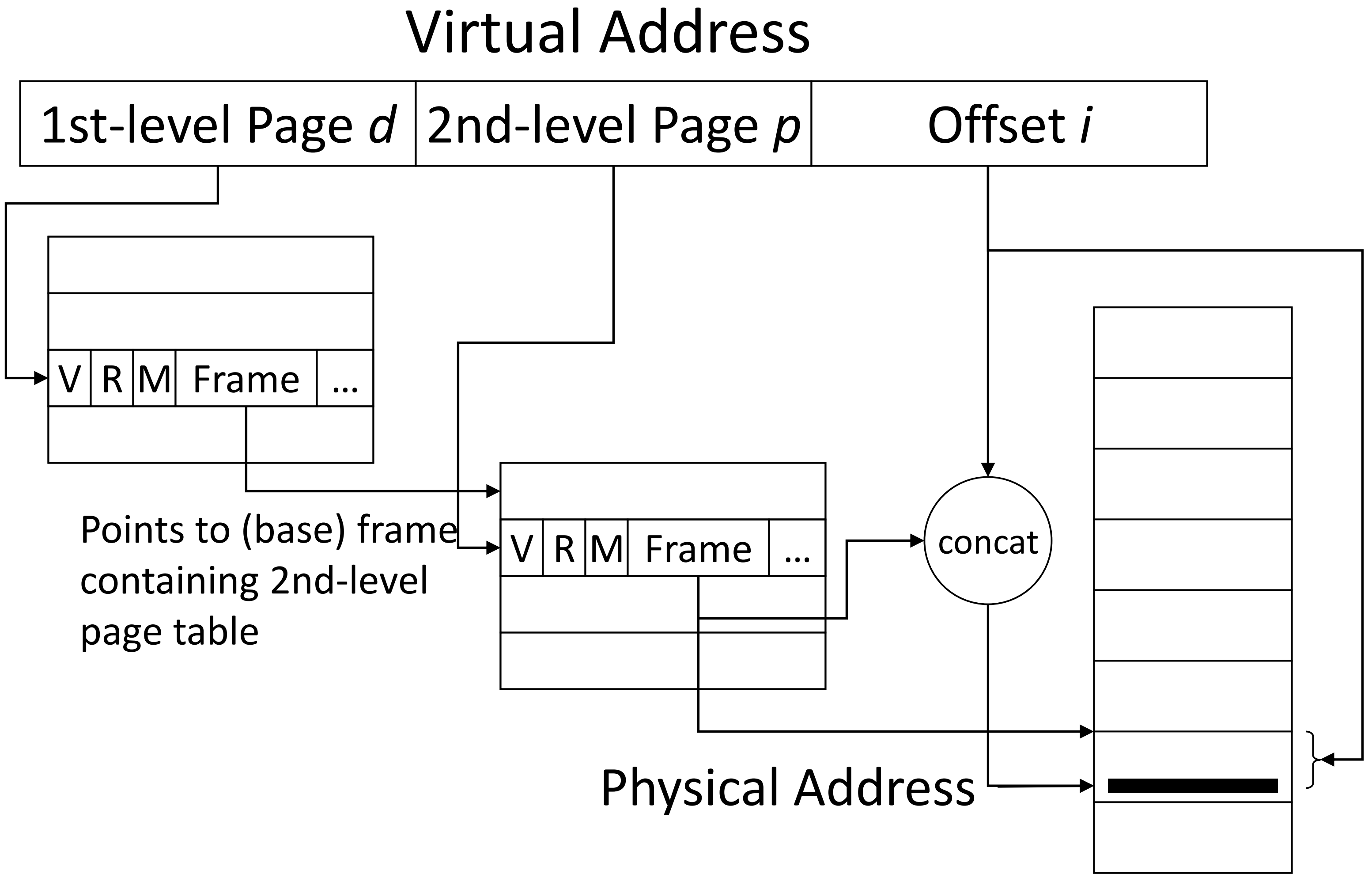
Multi-Level Page Tables



Points to (base) frame containing 2nd-level page table



Multi-Level Page Tables



Points to (base) frame containing 2nd-level page table

Insight: VAS is typically sparsely populated.

Idea: every process gets a page directory (1st-level table)

Only allocate 2nd-level tables when the process is using that VAS region!

Multi-Level Page Tables

- With only a single level, the page table must be large enough for the largest processes.

- Multi-level table => extra level of indirection:
  - WORSE performance – more memory accesses
  - Much better memory efficiency – process's page table is proportional to how much of the VAS it's using.

- Small process -> low page table storage
- Large process -> high page table storage, needed it anyway

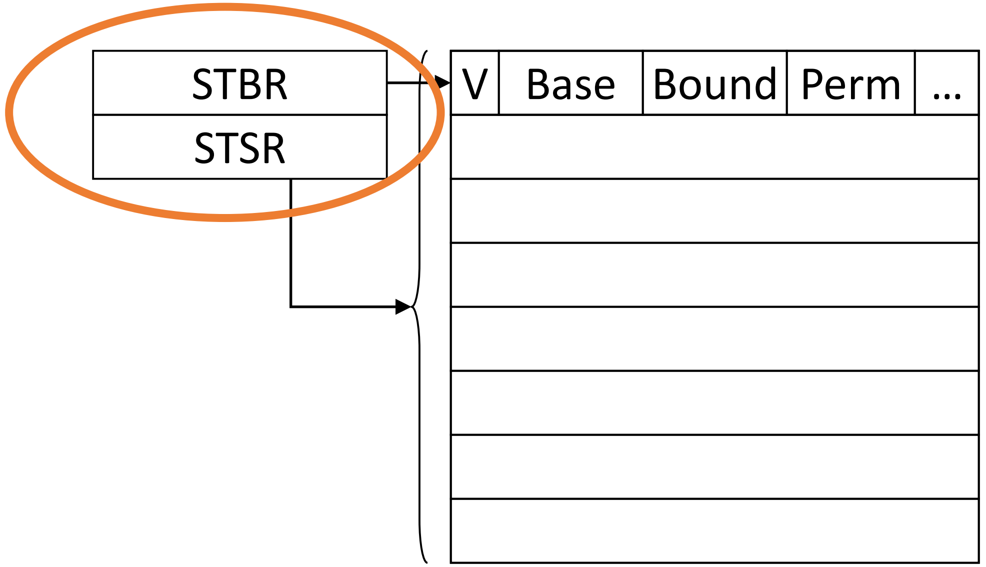
Outstanding Problems

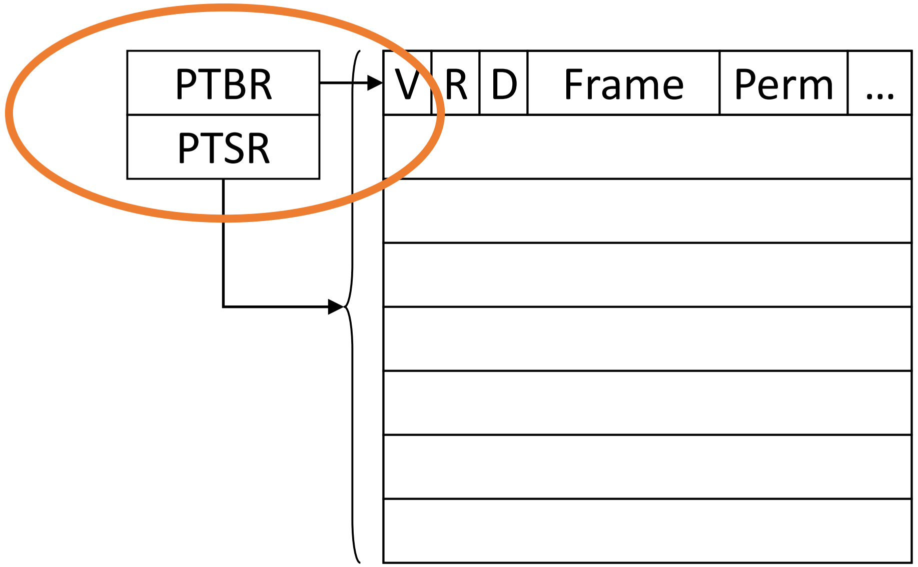
- Mostly considering paging from here on.

1. Page tables are way too big. Most processes don’t need that many pages, can’t justify a huge table.

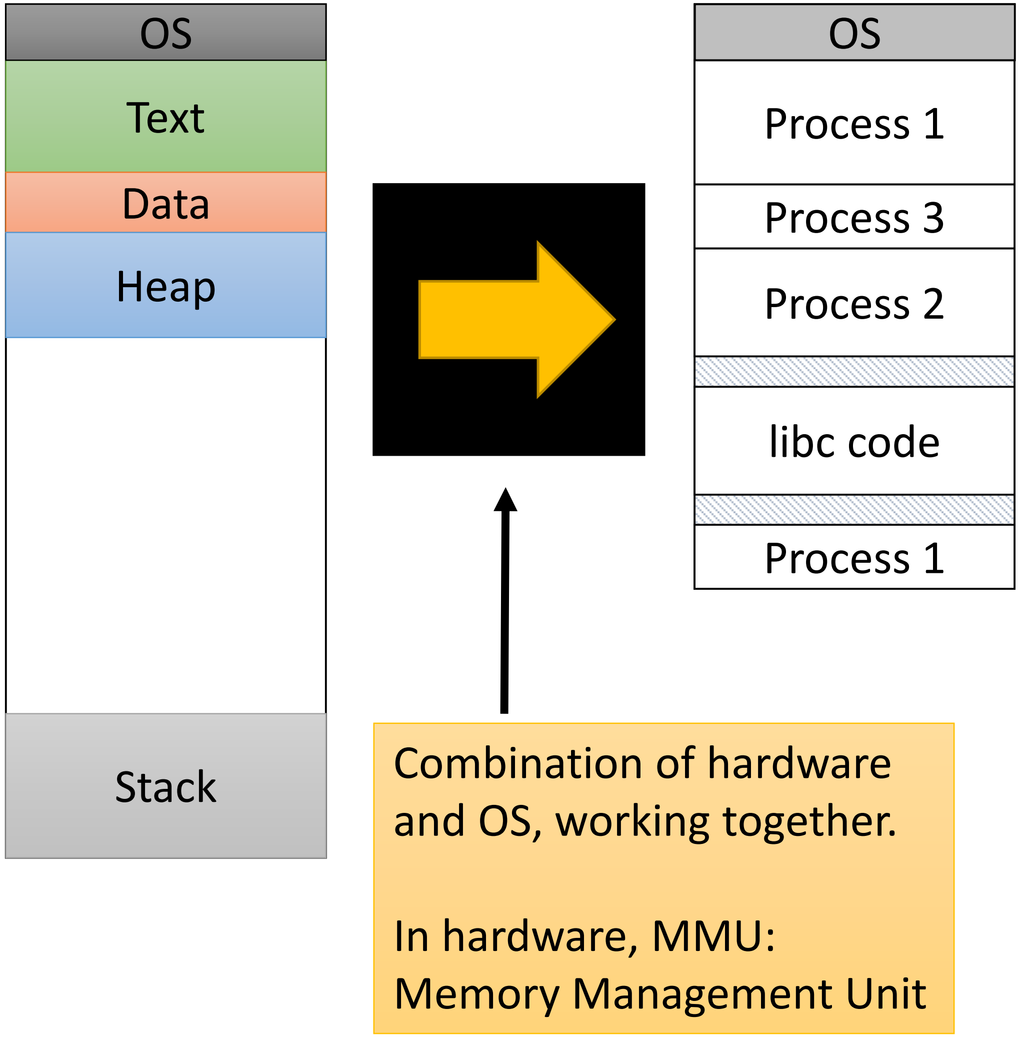
2. Adding indirection hurts performance.

How might these table registers help with performance?





Memory Management Unit



Combination of hardware and OS, working together.

In hardware, MMU: Memory Management Unit

- When a process tries to use memory, send the address to MMU.
- MMU will do as much work as it can. If it knows the answer, great!
- If it doesn’t, trigger exception (OS gets control), consult software table.

Memory Management Unit (MMU)

- By knowing where the page table is for the running process:

1. The MMU can (sometimes) translate addresses on it's own, without help from the OS! (more on this next time)

2. The MMU can cache translation info for frequently used pages

Translation Cost

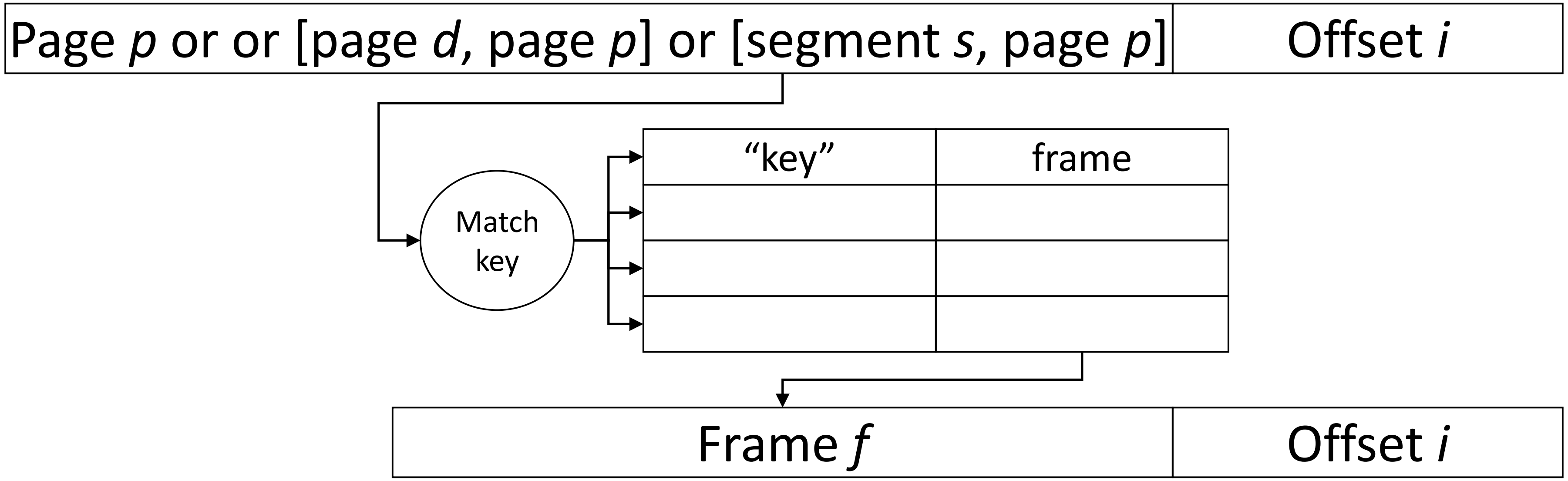
- Each application memory access now requires multiple accesses!

- Suppose memory takes 100 ns to access.
  - one-level paging: 200 ns
  - two-level paging: 300 ns

- Solution: Add hardware, take advantage of locality…
  - Most references are to a small number of pages
  - Keep translations of these in high-speed memory

Translation Look-aside Buffer (TLB)

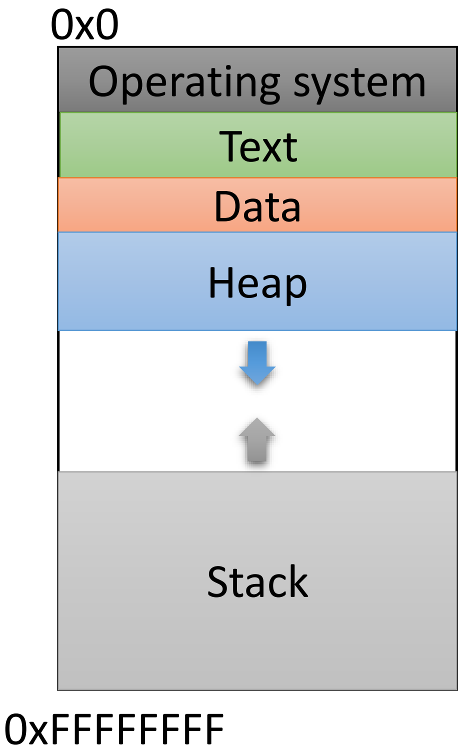
- Fast memory mapping cache inside MMU keeps most recent translations
  - If key matches, get frame number quickly
  - otherwise, wait for normal translation (in parallel)


Page *p* or or [page *d*, page *p*] or [segment *s*, page *p*]
Offset *i*
Match key
"key"
frame
Frame *f*
Offset *i*

Recall: Context Switching Performance

- Even though it’s fast, context switching is expensive:
  1. time spent is 100% overhead
  2. must invalidate other processes’ resources (caches, memory mappings)
  3. kernel must execute – it must be accessible in memory

- Also recall: Advantage of threads
  - Threads all share one process VAS



Translation Cost with TLB

- Cost is determined by
  - Speed of memory: ~ 100 nsec
  - Speed of TLB: ~ 10 nsec
  - Hit ratio: fraction of memory references satisfied by TLB, ~95%
- Speed to access memory with no address translation: 100 nsec
- Speed to access memory with address translation (2-level paging):
  - TLB miss: 300 nsec (200% slowdown)
  - TLB hit: 110 nsec (10% slowdown)
  - Average: 110 x 0.95 + 300 x 0.05 = 119.5 nsec

TLB Design Issues

- The larger the TLB…
  - the higher the hit rate
  - the slower the response
  - the greater the expense
  - the larger the space (in MMU, on chip)

- TLB has a major effect on performance!
  - Must be flushed on context switches
  - Alternative: tagging entries with PIDs

Summary

- Many options for translation mechanism: segmentation, paging, hybrid, multi-level paging. All of them: level(s) of *indirection*.

- Simplicity of paging makes it most common today.

- Multi-level page tables improve memory efficiency – page table bookkeeping scales with process VAS usage.

- TLB in hardware MMU exploits locality to improve performance