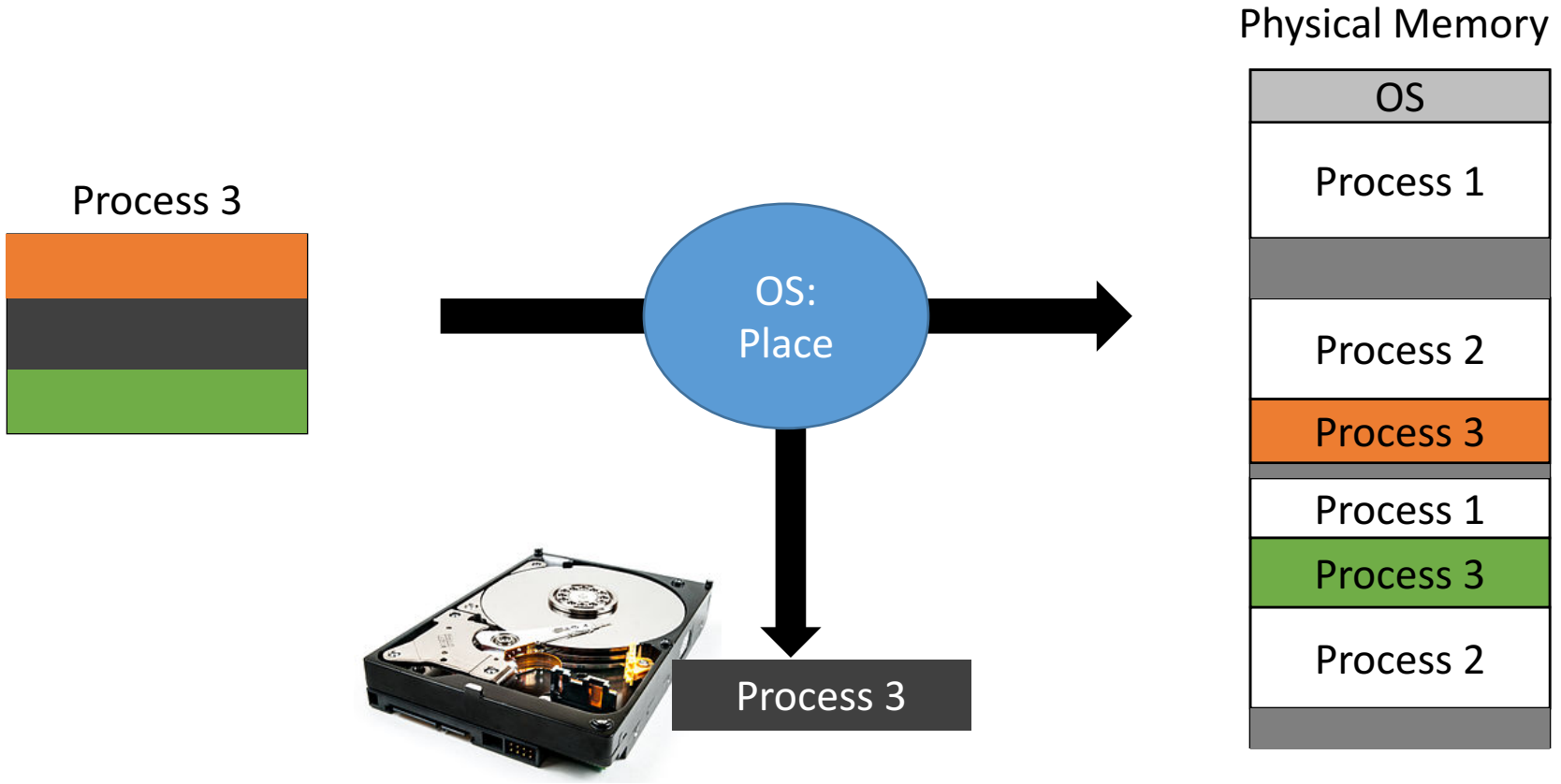
Paging

11/10/16

Recall from Tuesday

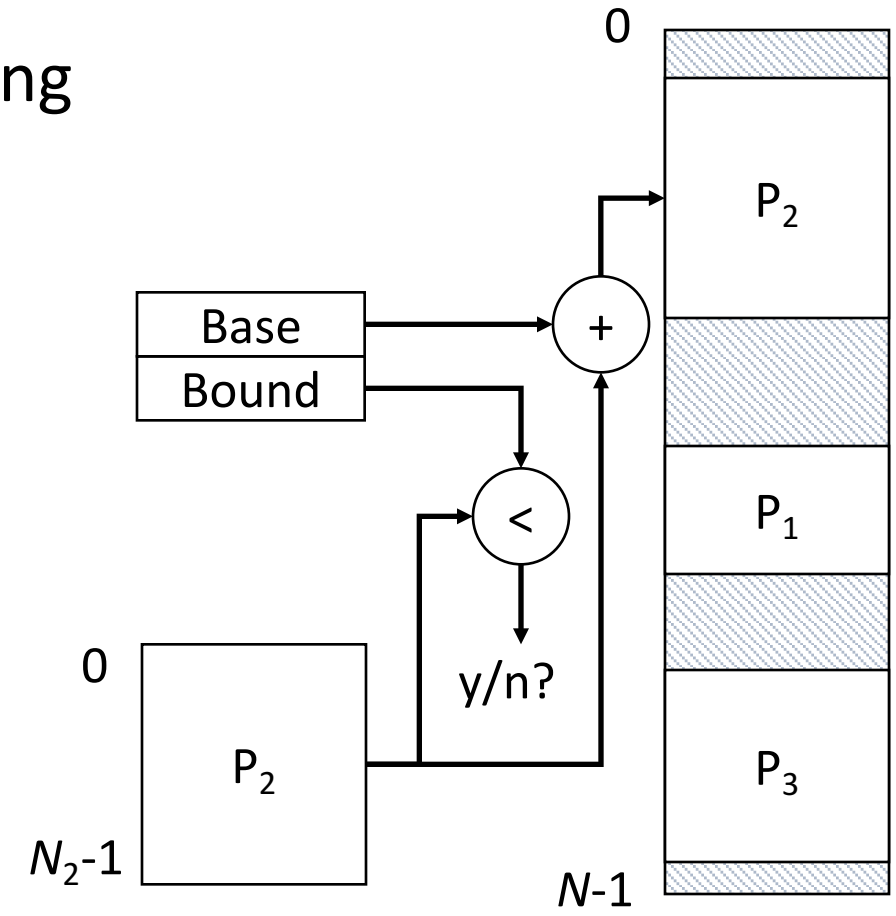
Our solution to fragmentation is to split up a process's address space into smaller chunks.



Recall from Tuesday

We support virtual addressing by translating addresses at runtime.

We can't achieve this using base and bound registers unless each process's memory is all in one block.



How can we do both?

- We want to support translation of virtual addresses to physical addresses on-the-fly.

- We want to split up each process's address space to use physical memory more efficiently.

The solution is **paging**.

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.

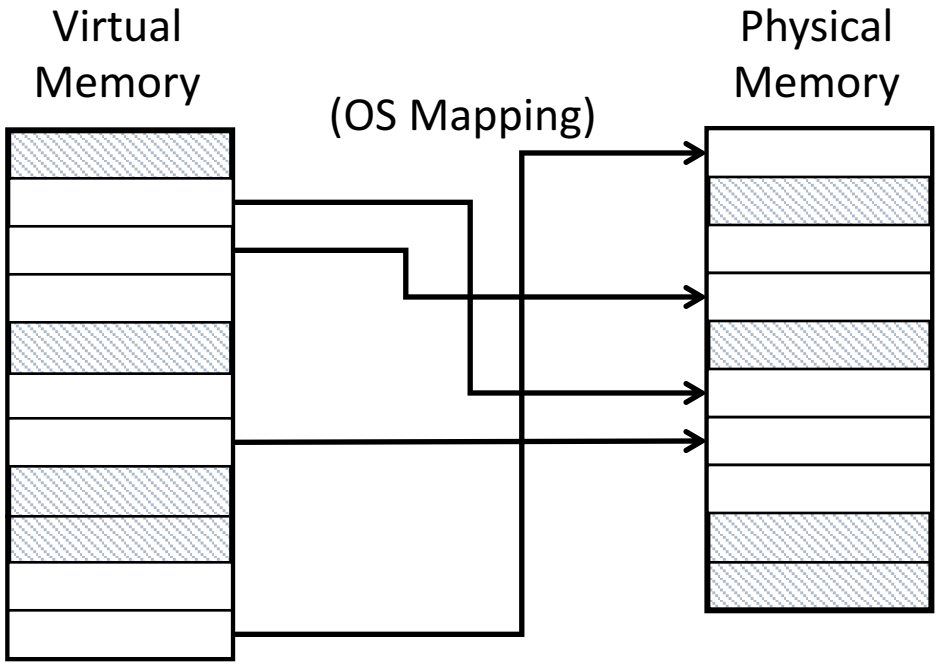
Main Idea

- ANY virtual page can be stored in any available frame.
  - Makes finding an appropriately-sized memory gap very easy – they're all the same size.

- For each process, OS keeps a table mapping each virtual page to physical frame.

Main Idea

- ANY virtual page can be stored in any available frame.
  - Makes finding an appropriately-sized memory gap very easy – they're all the same size.



Implications for fragmentation?

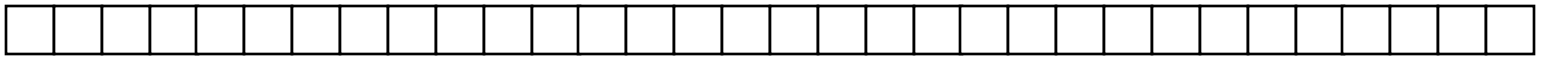
External: goes away. No more awkwardly-sized, unusable gaps.

Internal: About the same. Process can always request memory and not use it.

Addressing

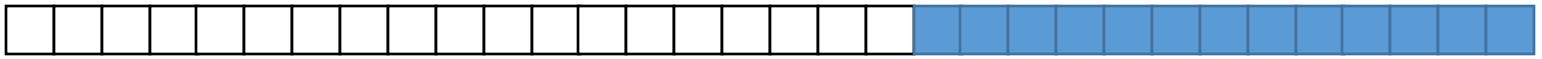
- Like we did with caching, we're going to chop up memory addresses into partitions.
- Virtual addresses:
  - High-order bits: page #
  - Low-order bits: offset within the page
- Physical addresses:
  - High-order bits: frame #
  - Low-order bits: offset within the frame

Example: 32-bit virtual addresses



- Suppose we have 8-KB (8192-byte) pages.
- We need enough bits to individually address each byte in the page.
  - How many bits do we need to address 8192 items?

Example: 32-bit virtual addresses

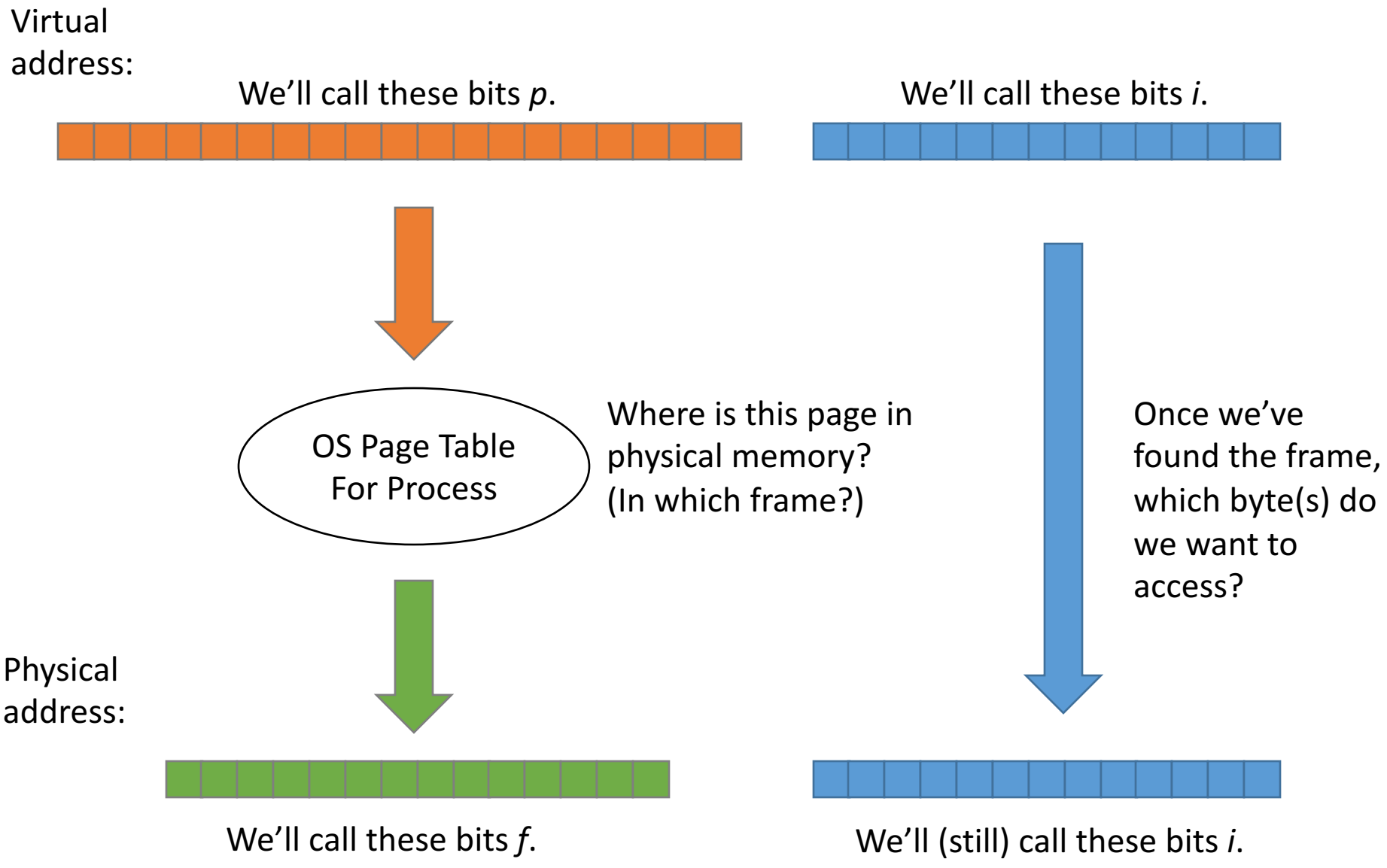


- Suppose we have 8-KB (8192-byte) pages.
- We need enough bits to individually address each byte in the page.
  - How many bits do we need to address 8192 items?
  - $2^{13} = 8192$, so we need 13 bits.
  - Lowest 13 bits: offset within page.

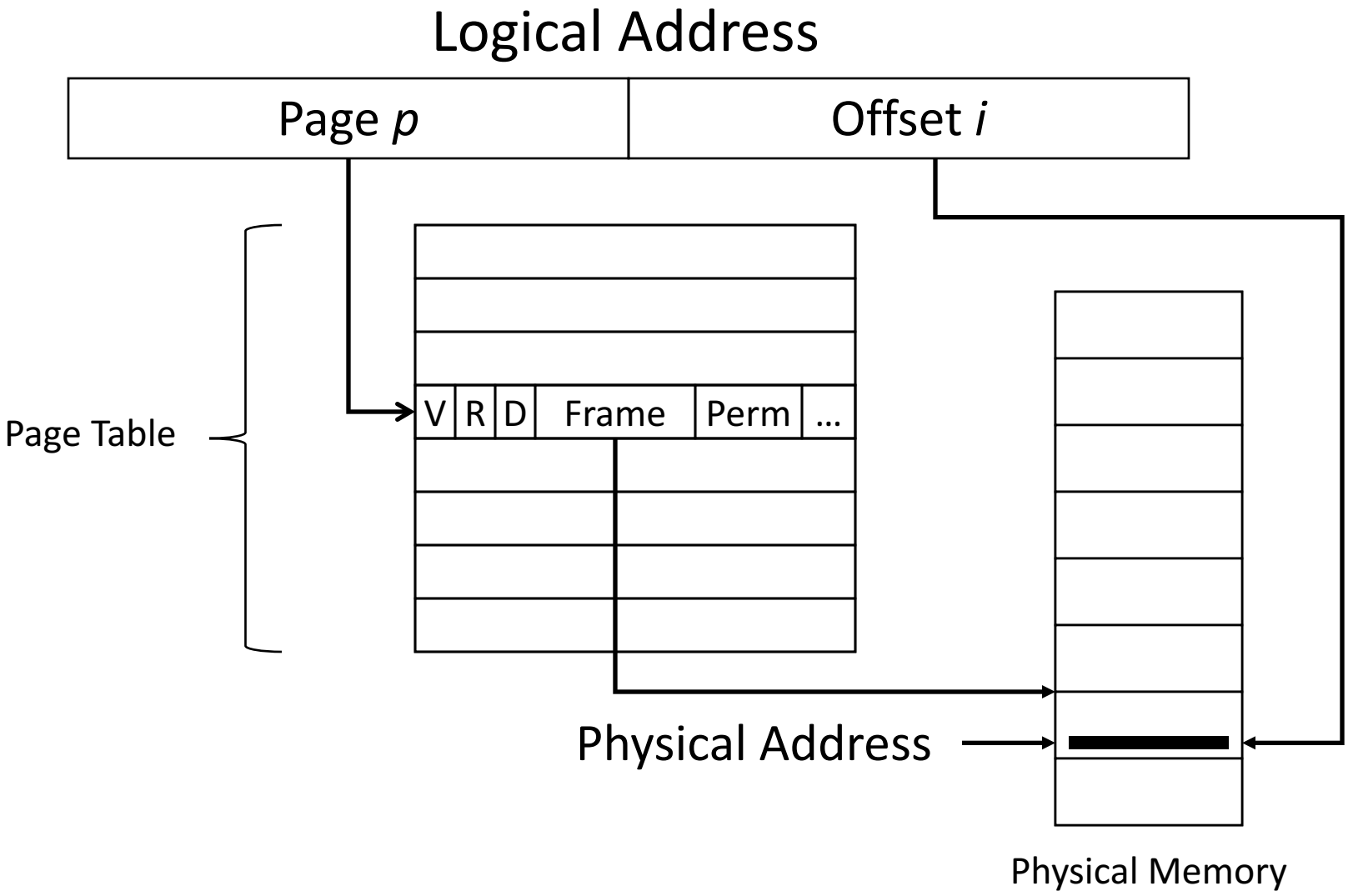
Example: 32-bit virtual addresses

- Suppose we have 8-KB (8192-byte) pages.
- We need enough bits to individually address each byte in the page.
  - How many bits do we need to address 8192 items?
  - $2^{13} = 8192$, so we need 13 bits.
  - Lowest 13 bits: offset within page.
- Remaining 19 bits: page number.

Address Partitioning

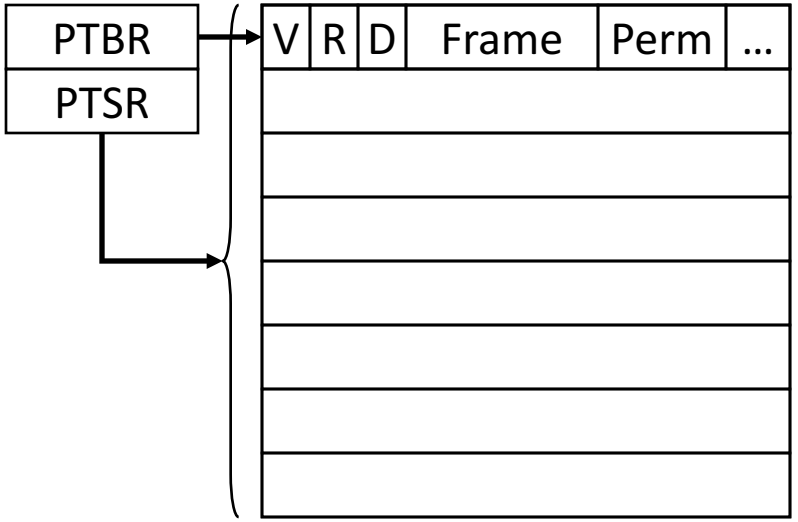


Address Translation



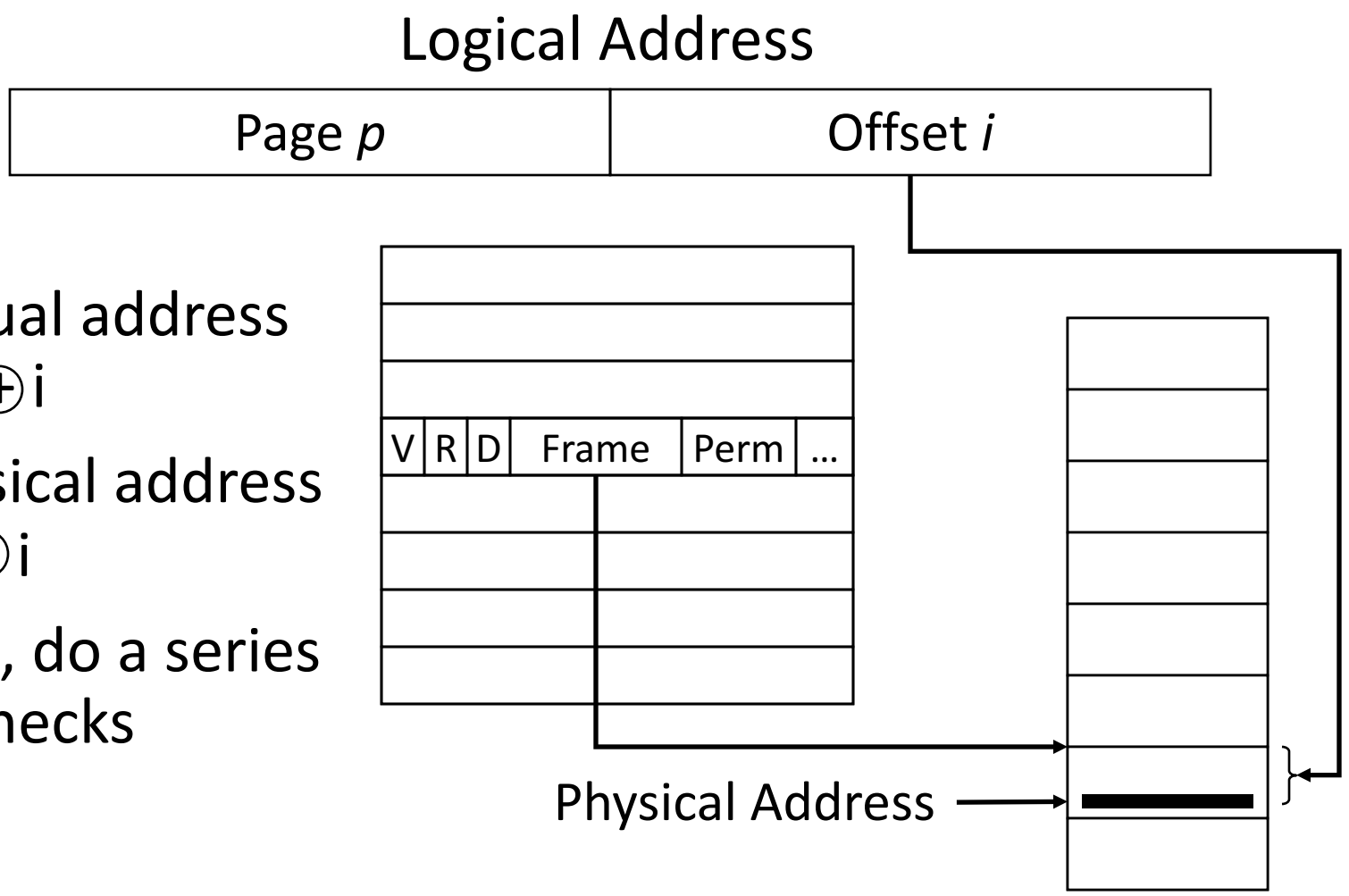
Page Table

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

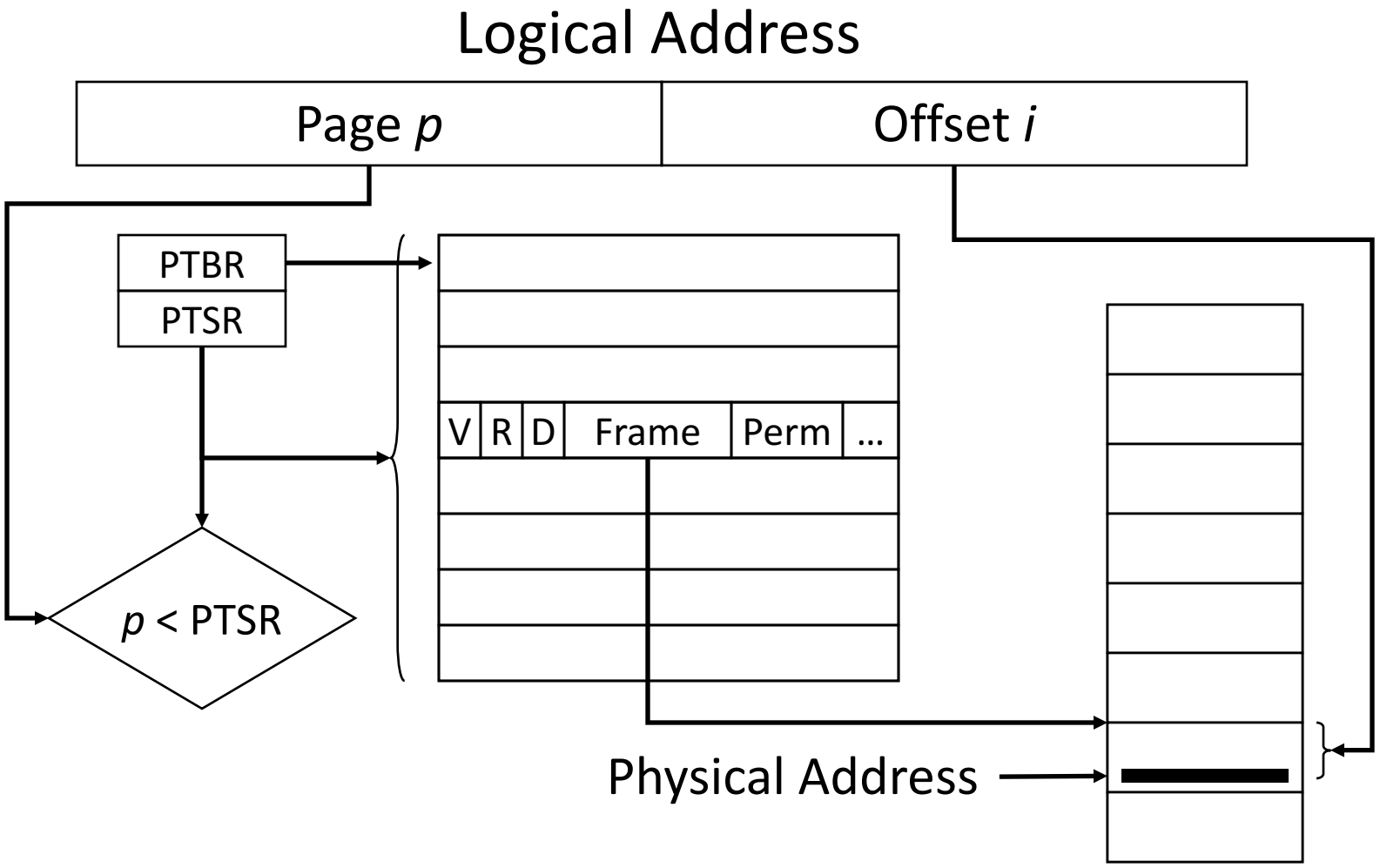


Address Translation

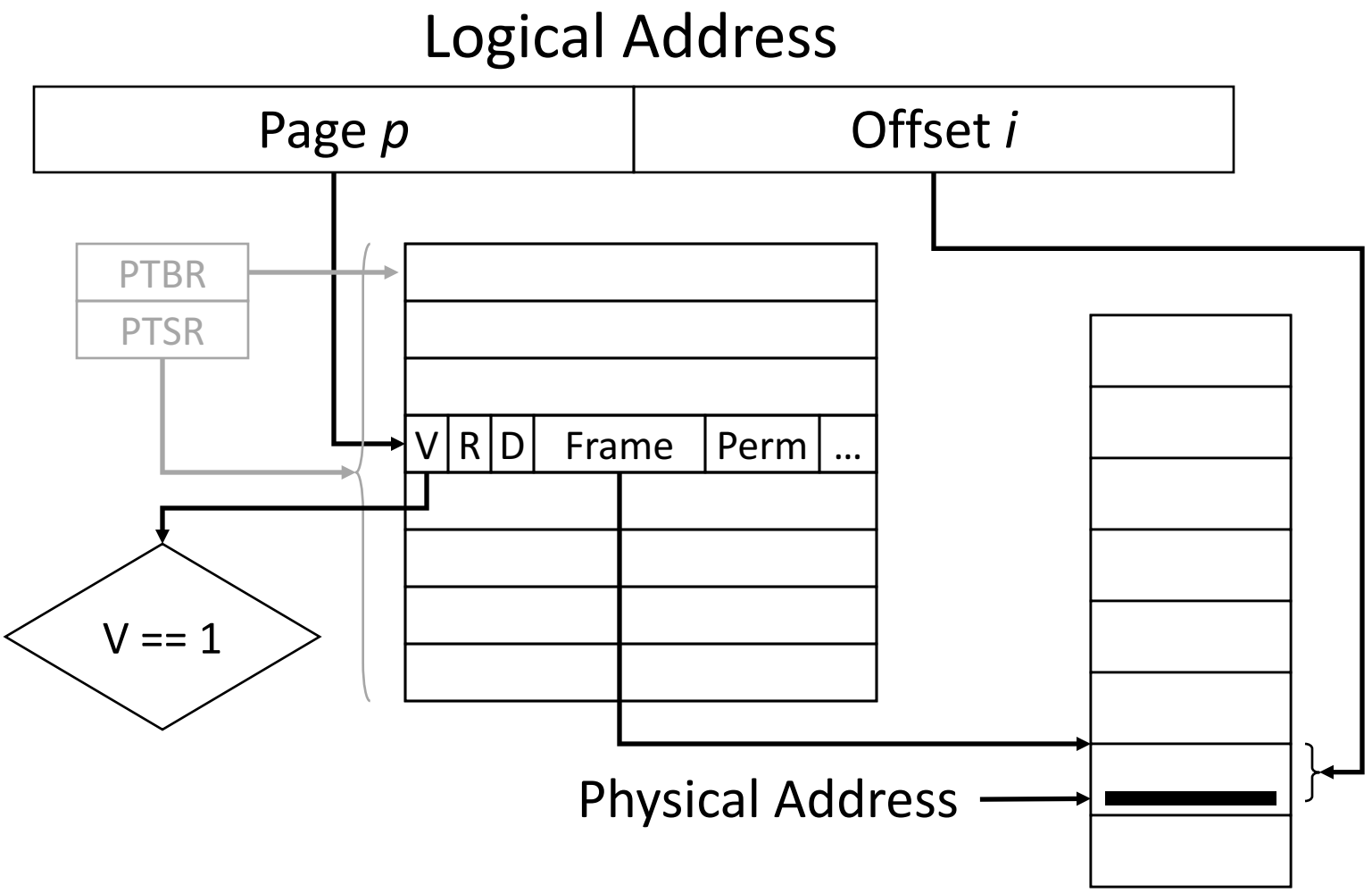
- Virtual address = p⊕i
- Physical address = f⊕i
- First, do a series of checks



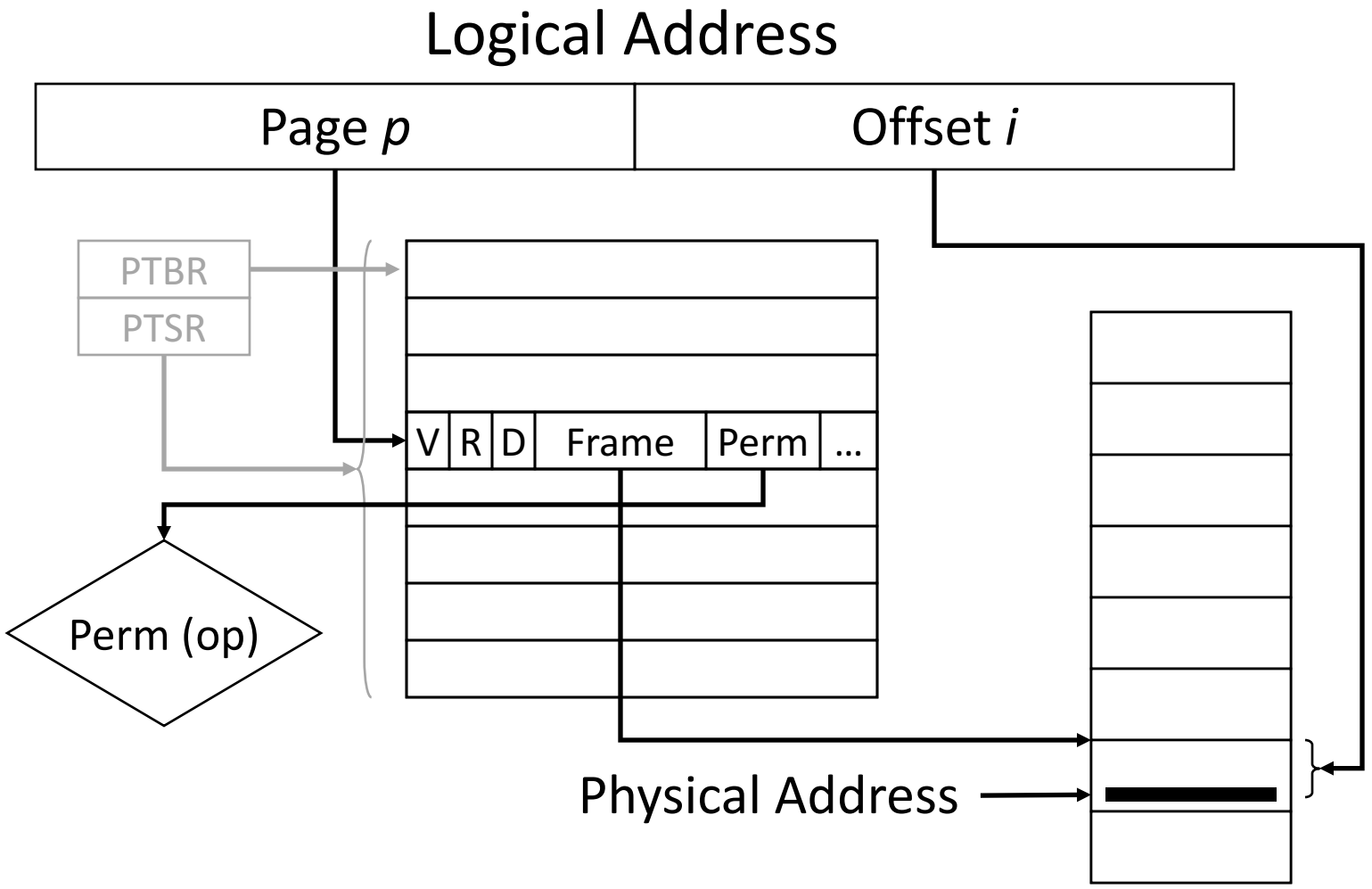
Check if Page *p* is Within Range



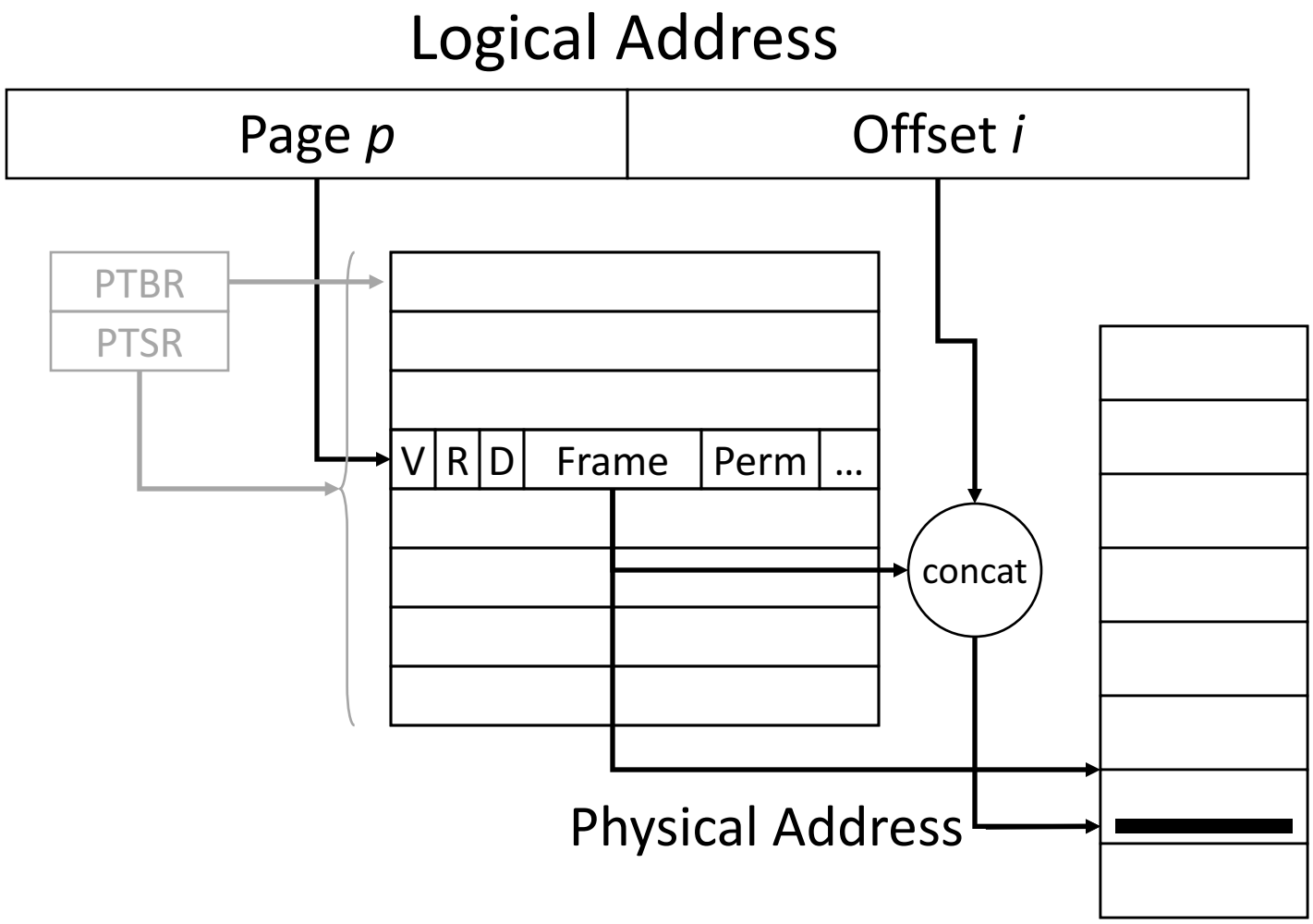
Check if Page Table Entry *p* is Valid



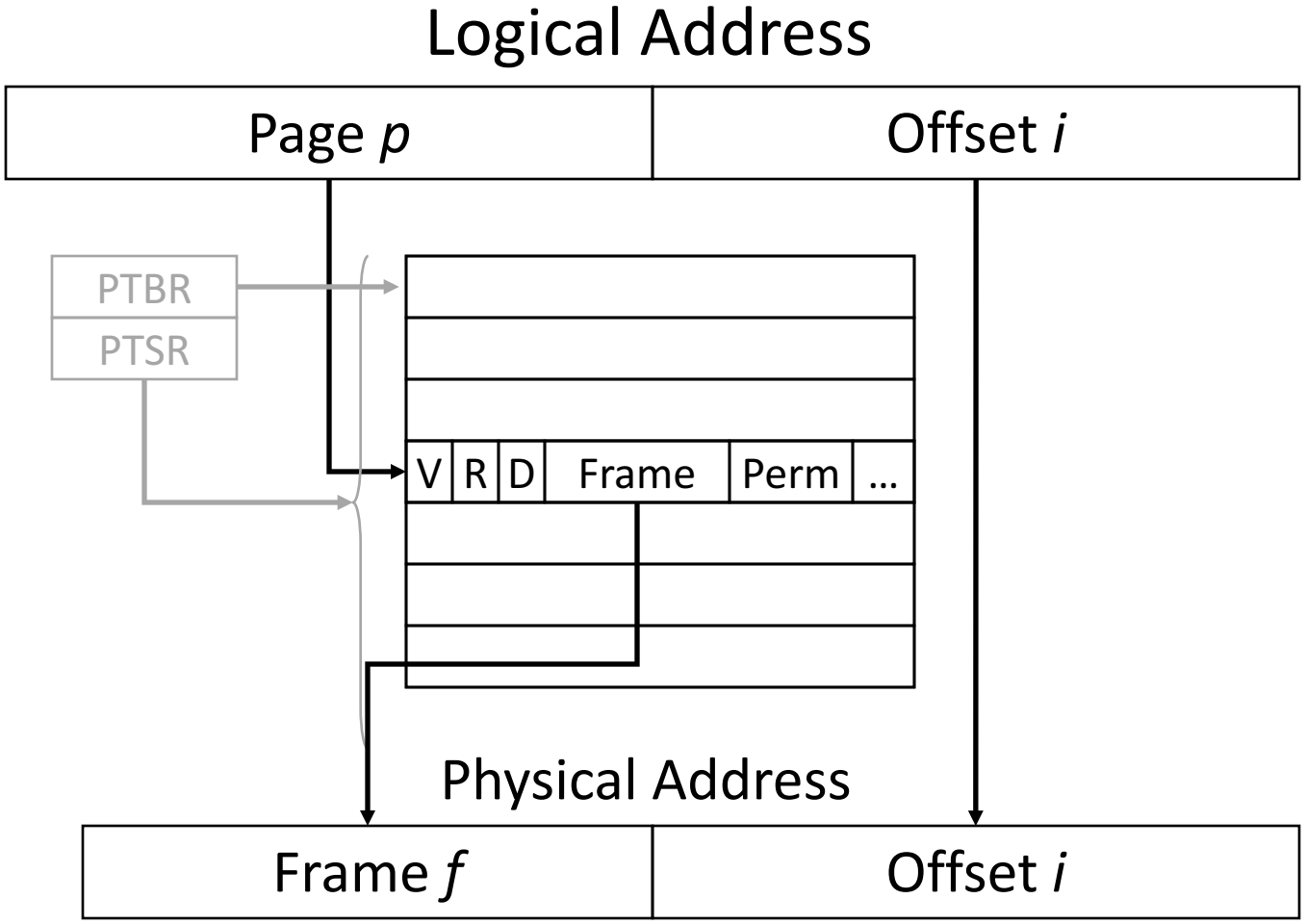
Check if Operation is Permitted



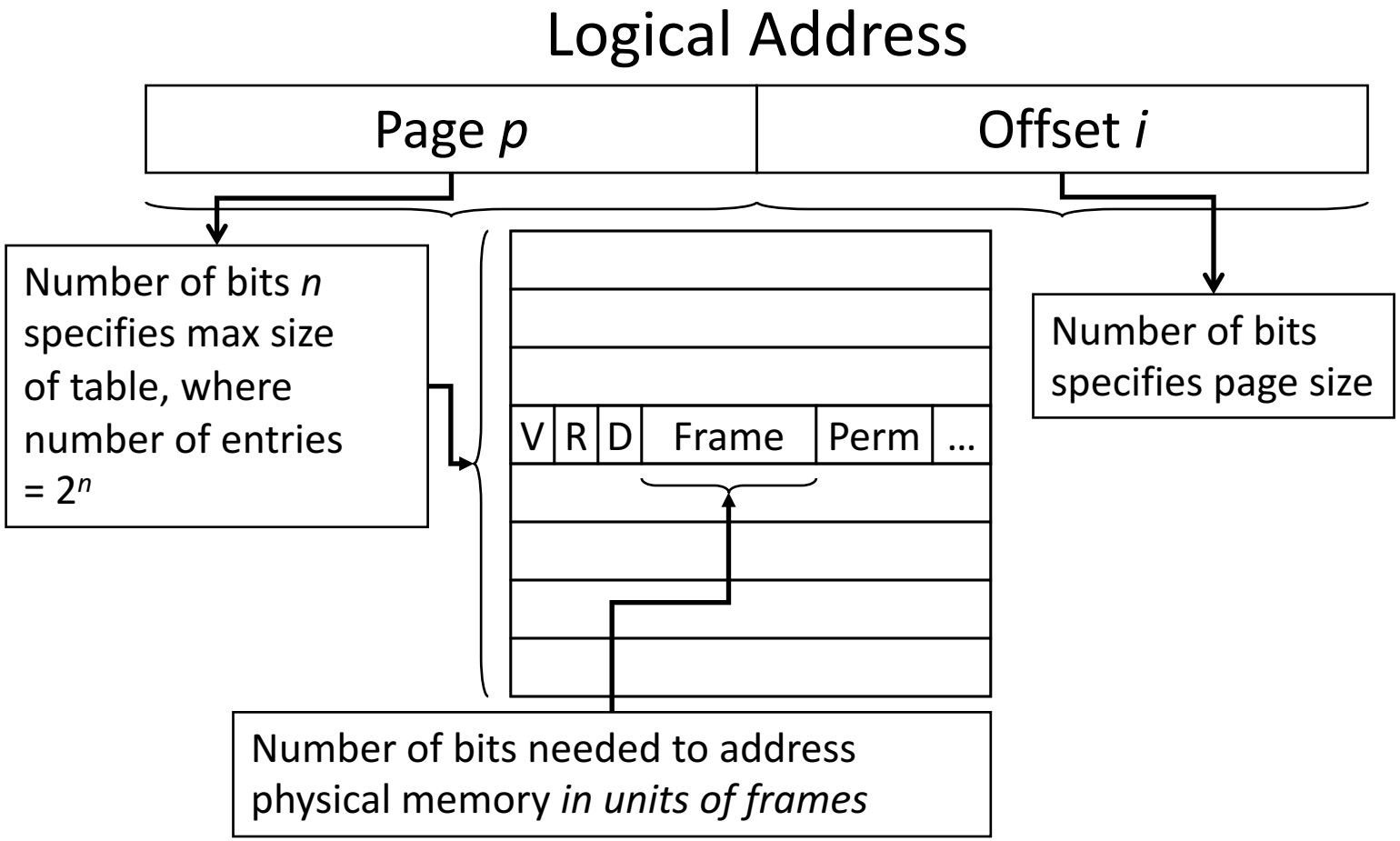
Translate Address



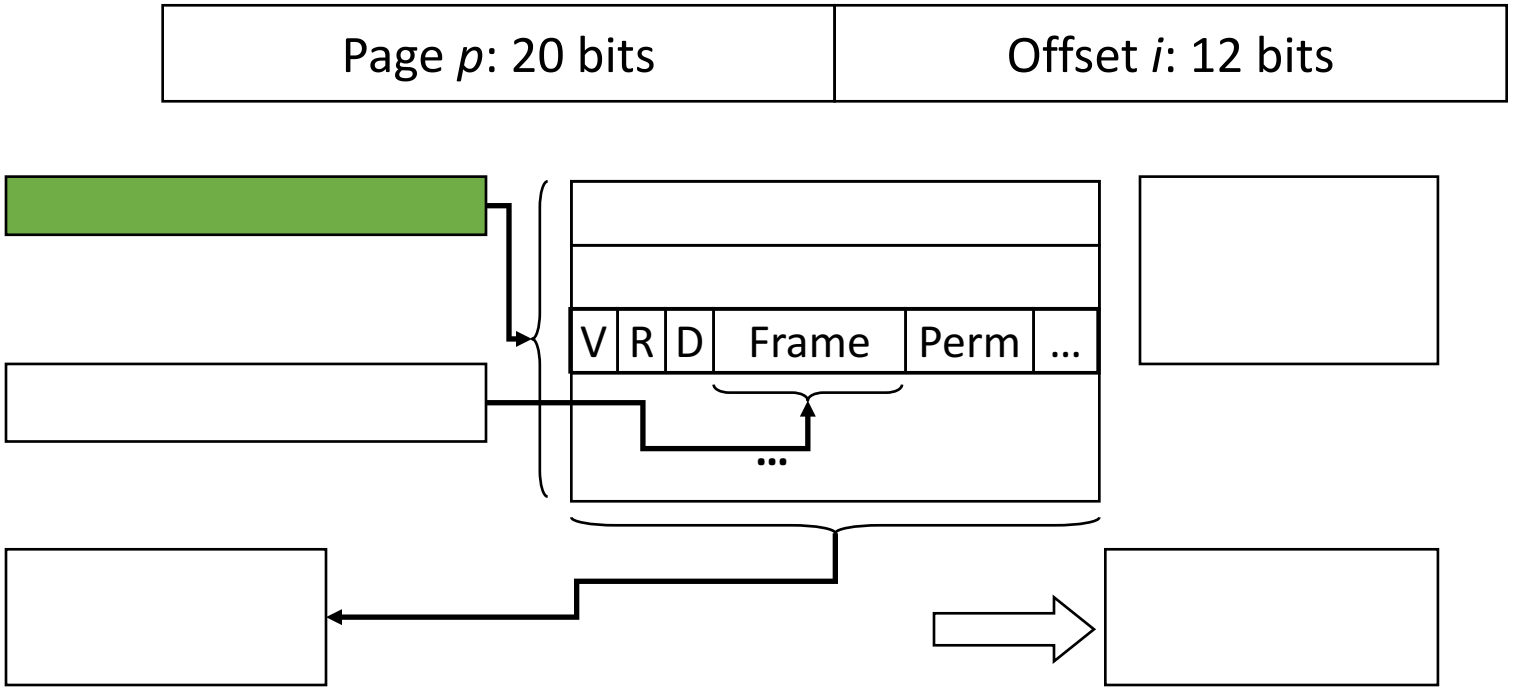
Physical Address by Concatenation



Sizing the Page Table



Example of Sizing the Page Table

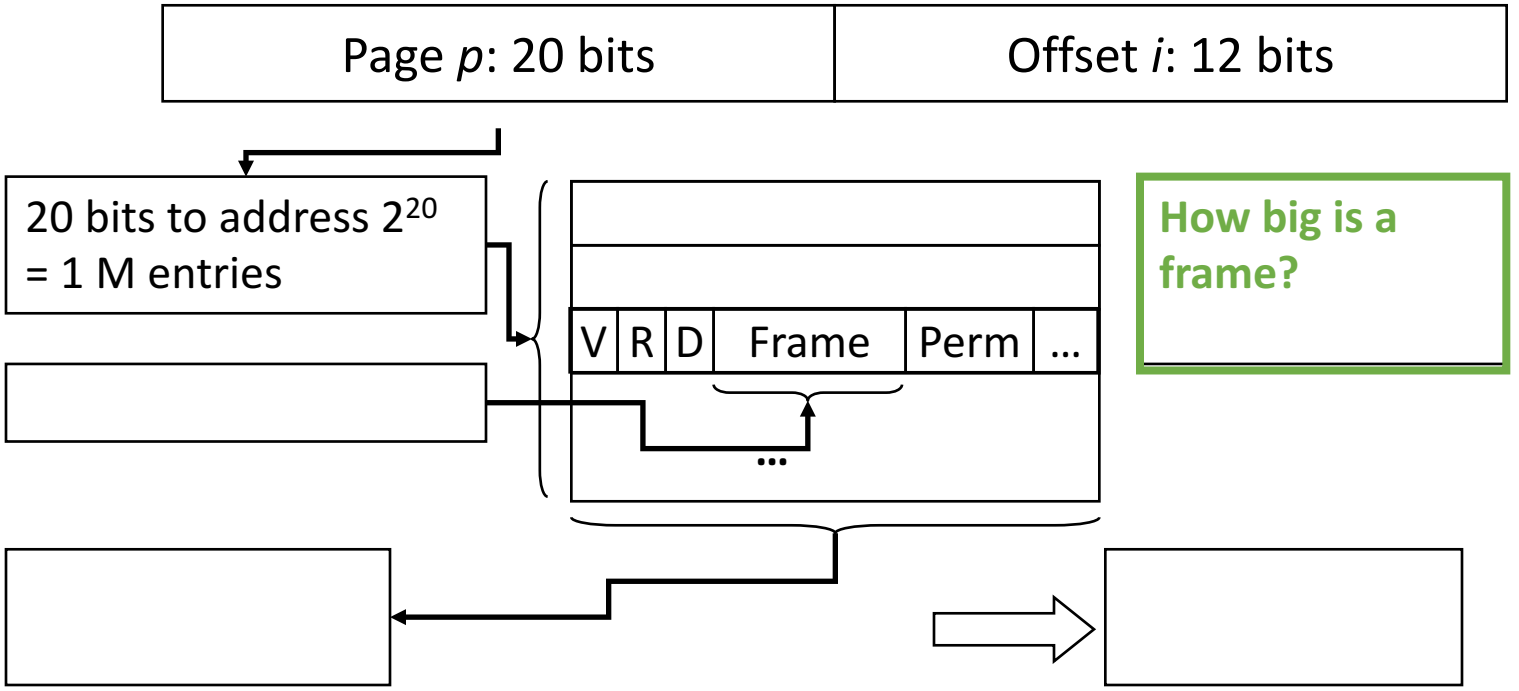


- 32 bit virtual addresses, 1 GB physical memory
- Address partition: 20 bit page number, 12 bit offset

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

Example of Sizing the Page Table

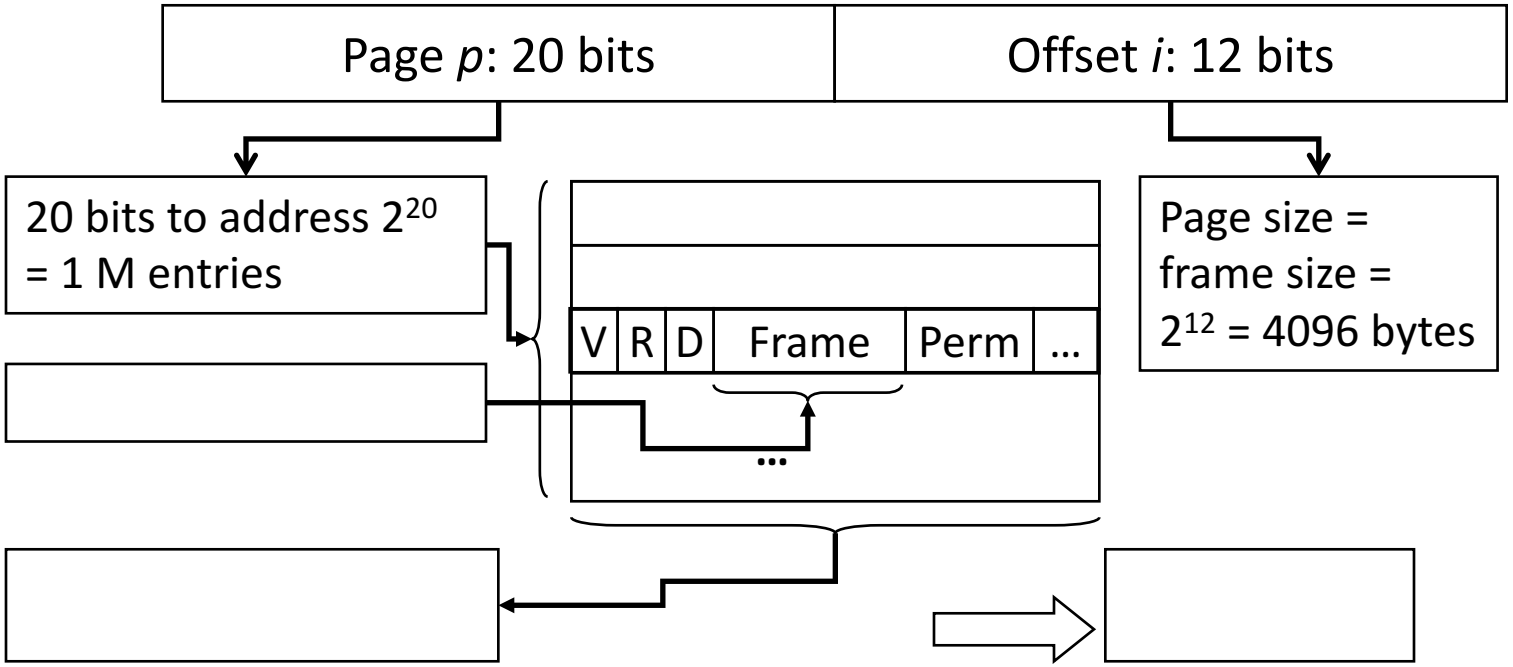


- 32 bit virtual addresses, 1 GB physical memory
- Address partition: 20 bit page number, 12 bit offset

What will be the frame size, in bytes?

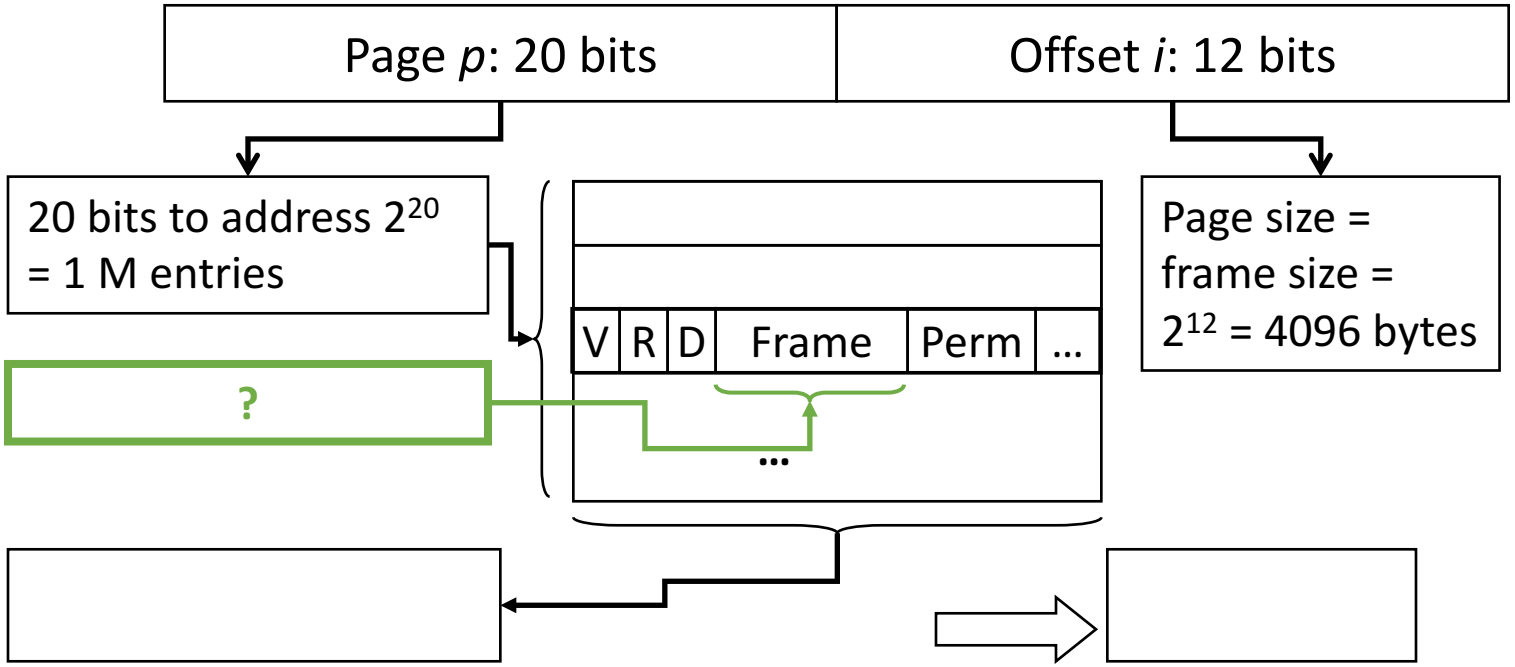
A. 2^{12}, because that's how many bytes the offset field can address
B. 2^{20}, because that's how many bytes the page field can address
C. 2^{30}, because that's how many bytes we need to address 1 GB
D. 2^{32}, because that's the size of the entire address space

Example of Sizing the Page Table



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

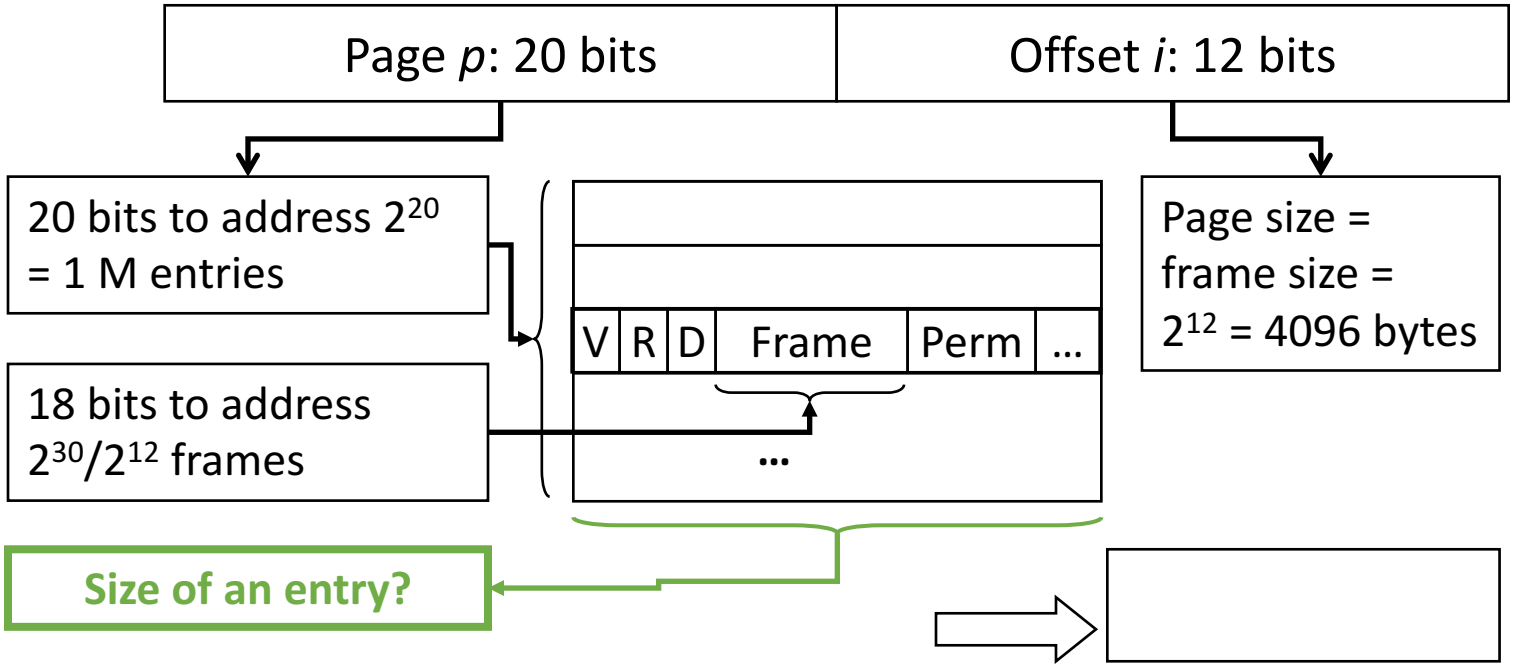
How many bits do we need to store the frame number?



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

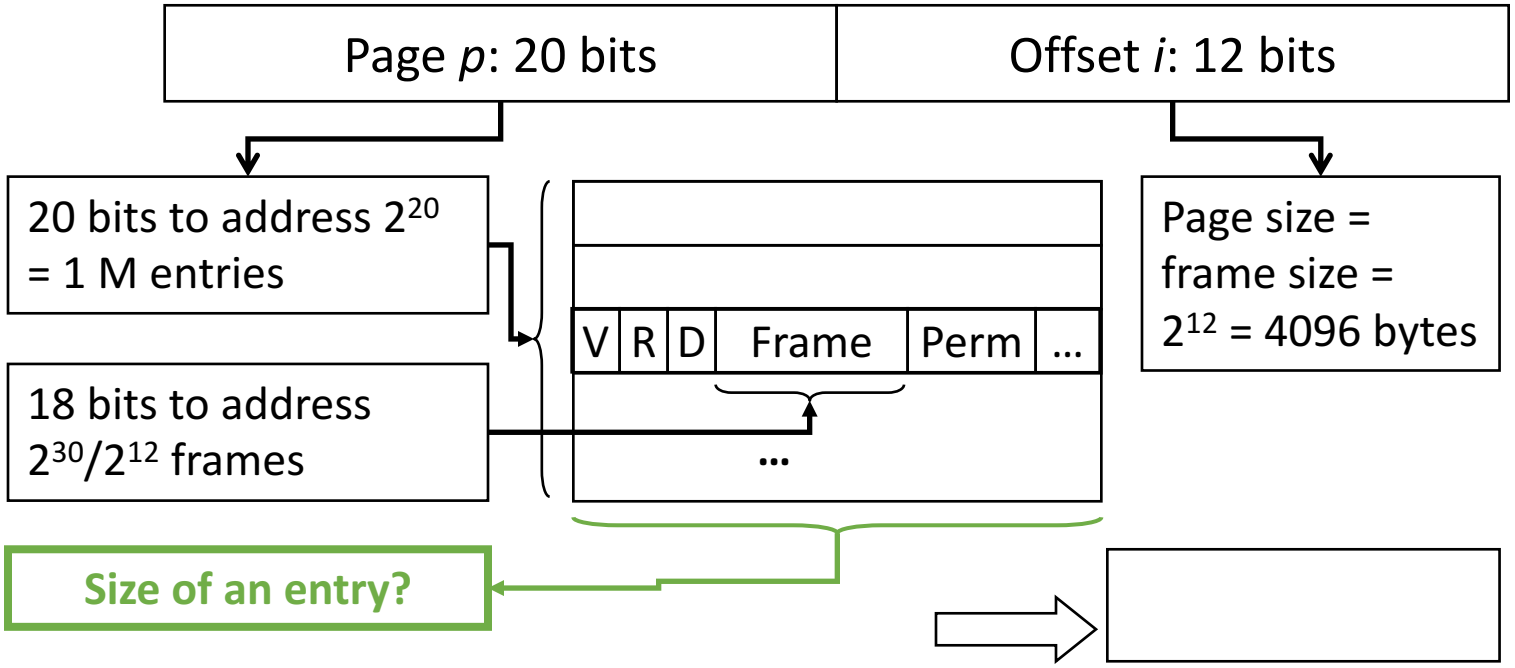
A: 12 B: 18 C: 20 D: 30 E: 32

Example of Sizing the Page Table



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

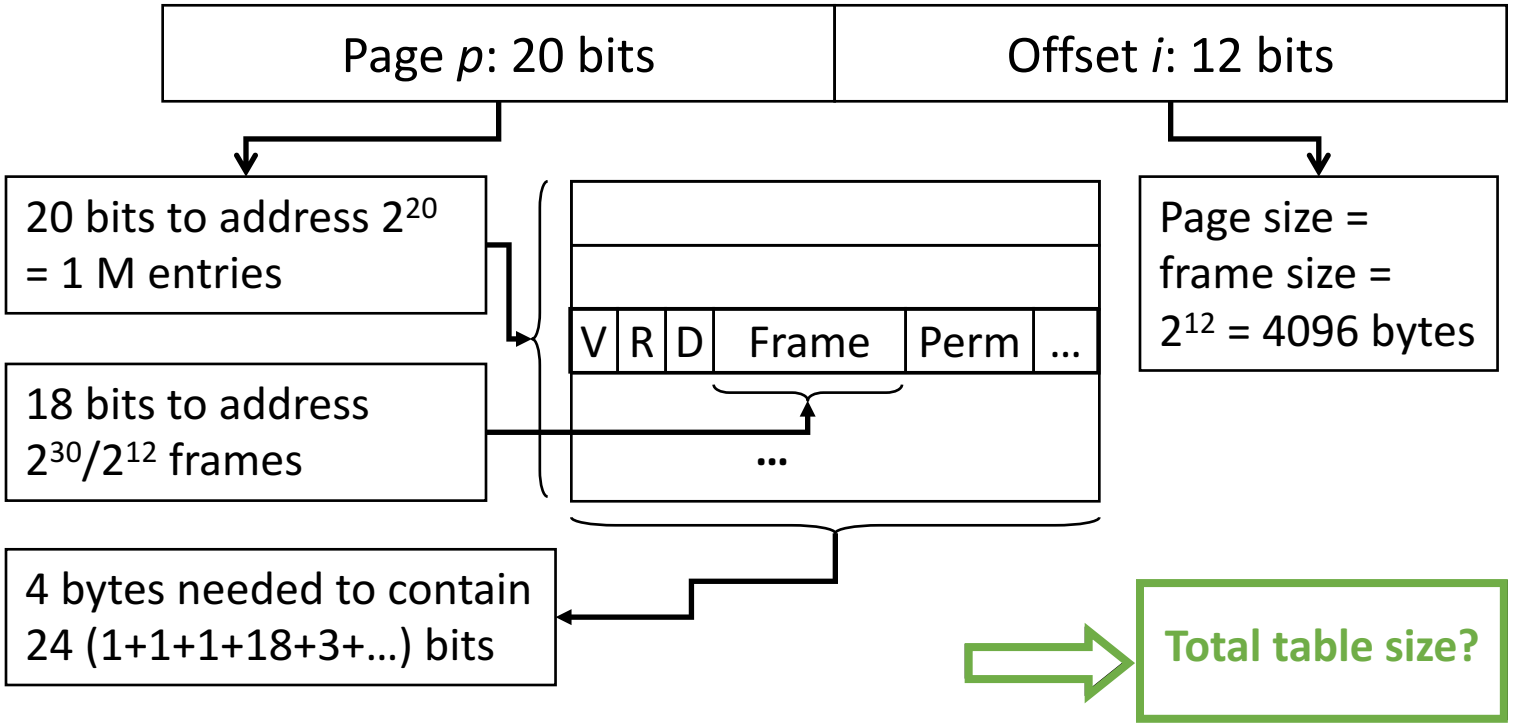
How big is an entry, in bytes? (Round to a power of two bytes.)



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

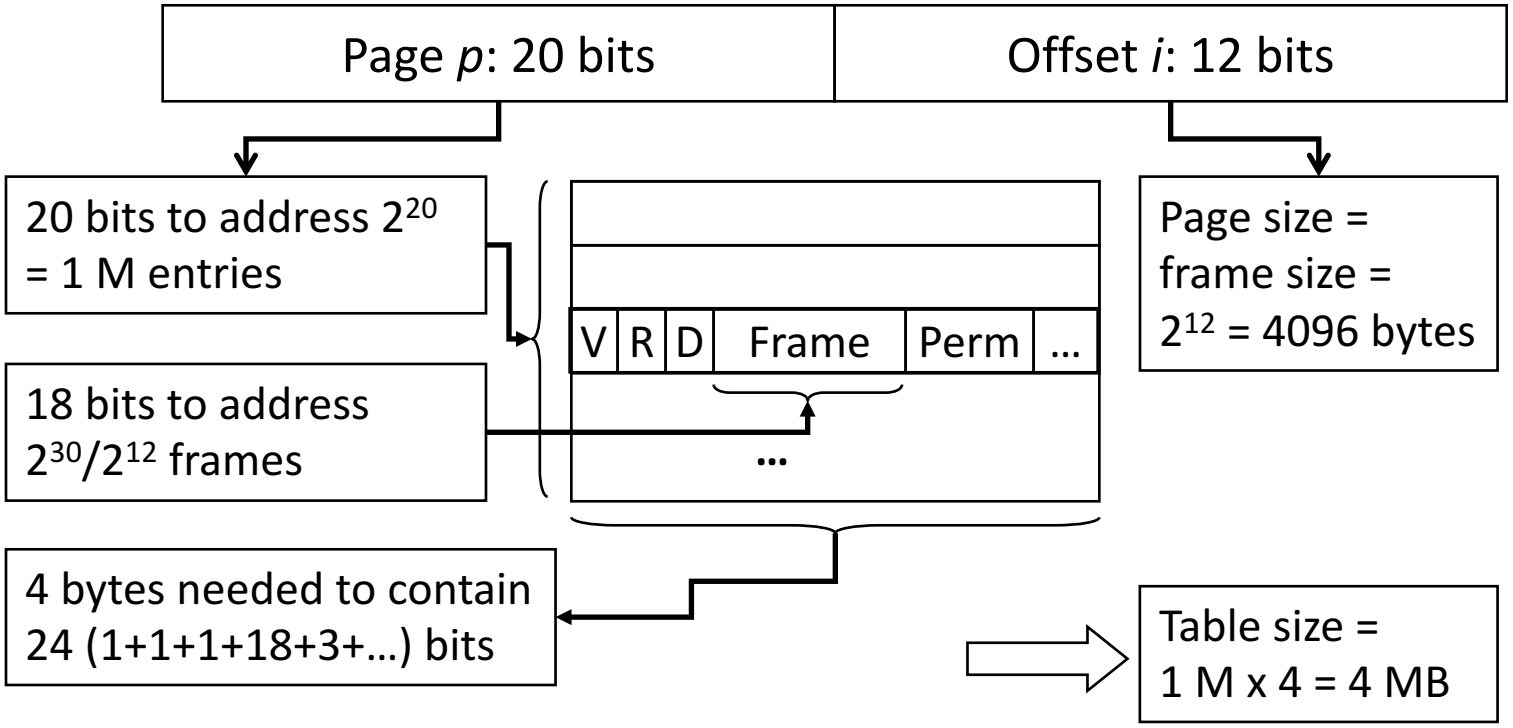
A: 1 B: 2 C: 4 D: 8 E:16

Example of Sizing the Page Table



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

Example of Sizing the Page Table



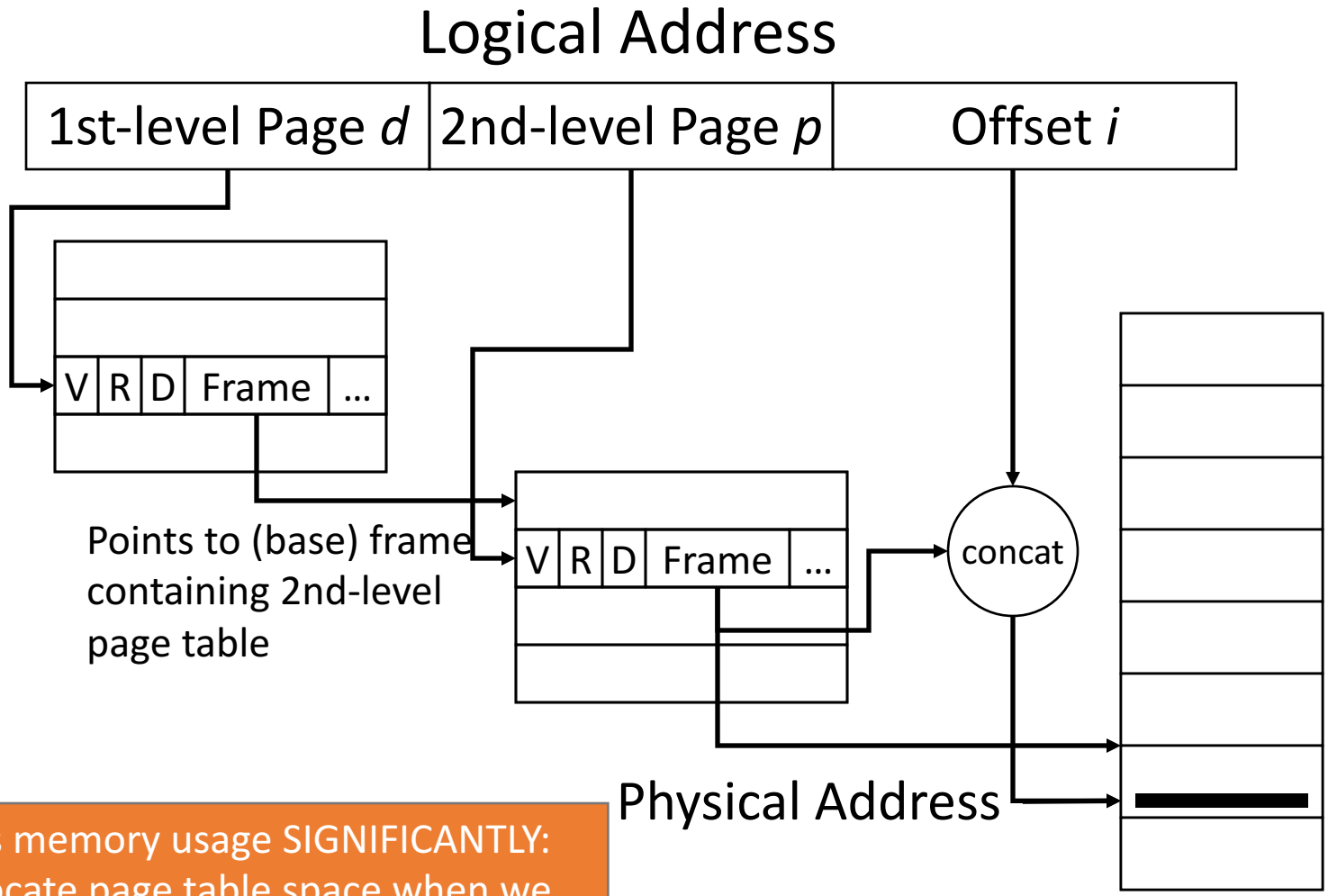
- 32 bit virtual addresses, 1 GB physical memory
- Address partition: 20 bit page number, 12 bit offset

Concerns

- 4 MB of bookkeeping for *every process*?
  - 200 processes -> 800 MB just to store page tables…
- We're going to need a ton of memory just for page tables…
- We need to do a lookup in our page table, which is in memory, every time a process accesses memory.
  - Isn't that slowing down memory by a factor of 2?

Multi-Level Page Tables

(You're not responsible for this. Take an OS class for the details.)



Reduces memory usage SIGNIFICANTLY: only allocate page table space when we need it. More memory accesses though…

Caching the page table

- 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
  - Most references are to a small number of pages
  - Keep translations of these in high-speed memory (a cache for page translation)

VM Implications

- Not all pieces need to be in memory
  - Need only piece being referenced
  - Other pieces can be on disk
  - Bring pieces in only when needed
- Illusion: there is much more memory
- What's needed to support this idea?
  - A way to identify whether a piece is in memory
  - A way to bring in pieces (from where, to where?)
  - Relocation (which we have)

Sample Contents of Page Table Entry

Valid	Ref	Dirty	Frame number	Prot: rwx

- Valid: is entry valid (page in physical memory)?
- Ref: has this page been referenced yet?
- Dirty: has this page been modified?
- Frame: what frame is this page in?
- Protection: what are the allowable operations?
  - **r**ead/**w**rite/e**x**ecute

Page faults

A page fault occurs when we try to access a virtual address that has no corresponding physical address.

mechanism for handling a page fault:

1. read in the virtual page from disk
   - Location kept in kernel data structure.
2. store it in a physical memory frame
   - May need to kick something else out.
3. Update PTE with frame num & valid bit = 1
4. Restart instruction that caused the page fault

Page Faults are Expensive

- Disk: 5-6 orders magnitude slower than RAM
  - Very expensive; but if very rare, tolerable
- Example
  - RAM access time: 100 nsec
  - Disk access time: 10 msec
  - p = page fault probability
  - Effective access time: 100 + p × 10,000,000 nsec
  - If p = 0.1%, effective access time = 10,100 nsec !

Handing faults from disk seems very expensive. How can we get away with this in practice?

A. We have lots of memory, and it isn't usually full.

B. We use special hardware to speed things up.

C. We tend to use the same pages over and over.

D. This is too common & expensive to do in practice!

Principle of Locality

- Not all pieces referenced uniformly over time
  - Make sure most referenced pieces in memory
  - If not, thrashing: constant fetching of pieces

- References cluster in time/space
  - Will be to same or neighboring areas
  - Allows prediction based on past

Page Replacement

- Goal: remove page(s) not exhibiting locality
- Page replacement is about
  - which page(s) to remove
  - when to remove them
- How to do it in the cheapest way possible
  - Least amount of additional hardware
  - Least amount of software overhead

Basic Page Replacement Algorithms

- FIFO: select page that is oldest
  - Simple: use frame ordering
  - Doesn't perform very well (oldest may be popular)
- OPT: select page to be used furthest in future
  - Optimal, but requires future knowledge
  - Establishes best case, good for comparisons
- LRU: select page that was least recently used
  - Predict future based on past; works given locality
  - Costly: time-stamp pages each access, find least
- Goal: minimize replacements (maximize locality)

Summary

- We give each process a virtual address space to simplify process execution.

- OS maintains mapping of virtual address to physical memory locations in a page table.
  - One page table for every process

- Provides the abstraction of very large memory: not all pages need be resident in memory
  - Bring pages in from disk on demand

Worksheet example

Step through the stream of Virtual Addresses from the CPU:

- Show how the bits of each address are used for each Virtual Address & its Physical Address mapping
- Translate each VA to its PA using the appropriate PTE
- Update PTEs appropriately as you go
- Show the history of the contents of RAM as these addresses are accessed
  - Which virtual page of which process does it store
- Implement a FIFO page replacement policy for RAM