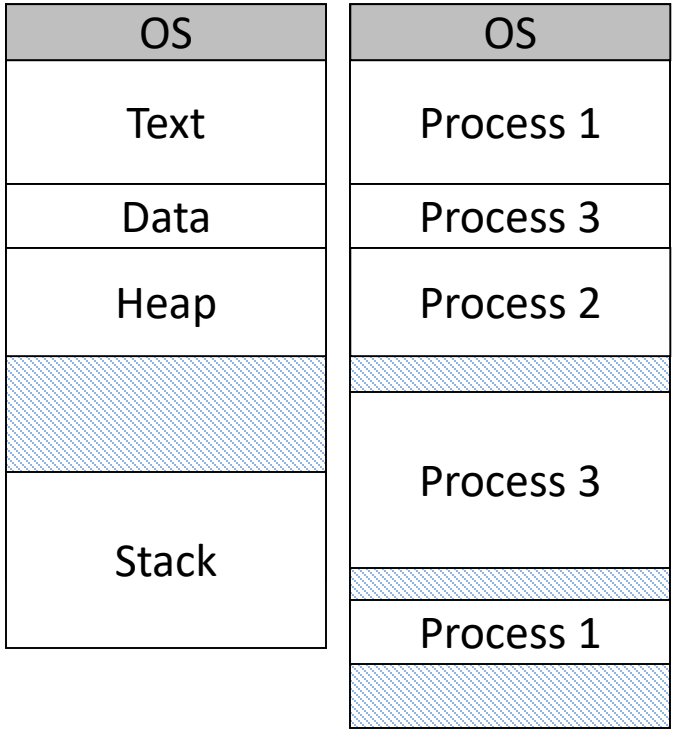
CS 31: Intro to Systems Virtual Memory

Kevin Webb
Swarthmore College
November 15, 2018

Reading Quiz

Memory

- Abstraction goal: make every process think it has the same memory layout.
  - MUCH simpler for compiler if the stack always starts at 0xFFFFFFFF, etc.
- Reality: there's only so much memory to go around, and no two processes should use the same (physical) memory addresses.

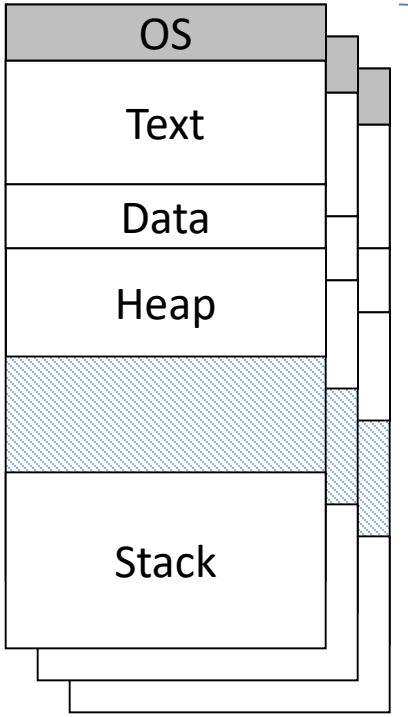


OS (with help from hardware) will keep track of who's using each memory region.

Memory Terminology

Virtual (logical) Memory: The abstract view of memory given to processes. Each process gets an independent view of the memory.

Physical Memory: The contents of the hardware (RAM) memory. Managed by OS. Only ONE of these for the entire machine!



0x0

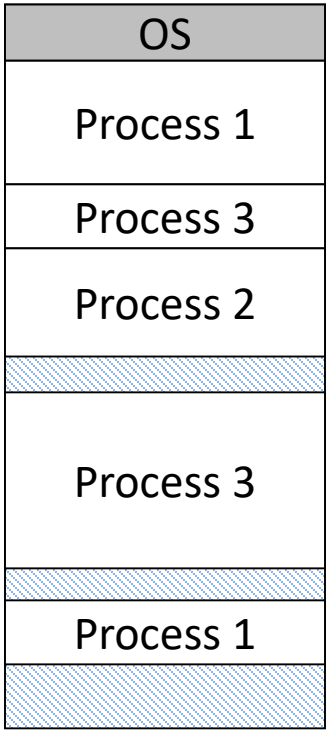
0xFFFFFFFF

Virtual address space (VAS): fixed size.

Address Space: Range of addresses for a region of memory.

The set of available storage locations.

0x0



0x…

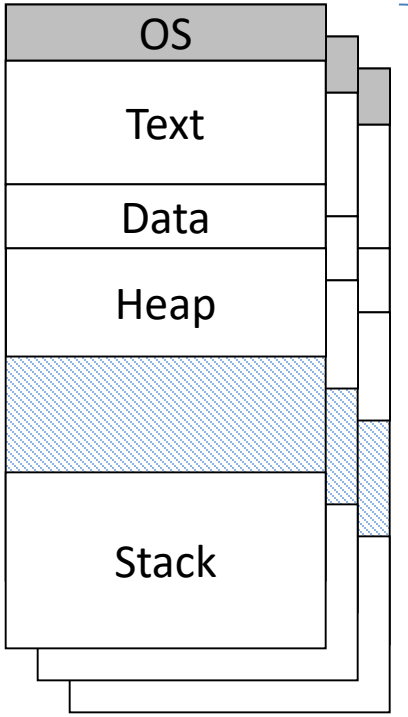
(Determined by amount of installed RAM.)

Memory Terminology

Note: It is common for VAS to appear larger than physical memory.

32-bit (IA32): Can address up to 4 GB, might have less installed

64-bit (X86-64): Our lab machines have 48-bit VAS (256 TB), 36-bit PAS (64 GB)



0x0

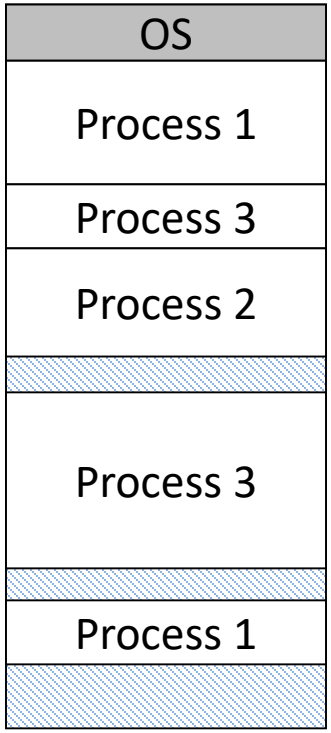
0xFFFFFFFF

Virtual address space (VAS): fixed size.

Address Space: Range of addresses for a region of memory.

The set of available storage locations.

0x0



0x…

(Determined by amount of installed RAM.)

Cohabitating Physical Memory

- If process is given CPU, must also be in memory.
- Problem
  - Context-switching time (CST): 10 µsec
  - Loading from disk: 10 MB/s
  - To load 1 MB process: 100 msec = 10,000 x CST
  - Too much overhead! Breaks illusion of simultaneity
- Solution: keep multiple processes in memory
  - Context switch only between processes in memory

Memory Issues and Topics

- Where should process memories be placed?
  - Topic: “Classic” memory management
- How does the compiler model memory?
  - Topic: Logical memory model
- How to deal with limited physical memory?
  - Topics: Virtual memory, paging

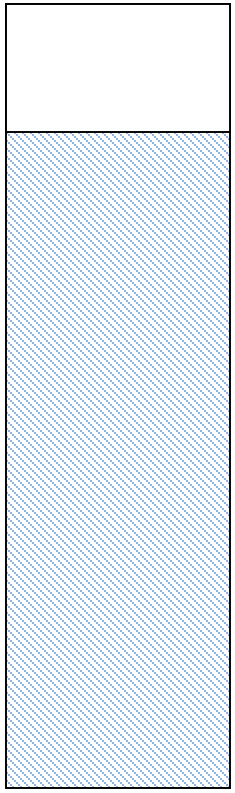
Plan: Start with the basics (out of date) to motivate why we need the complex machinery of virtual memory and paging.

Problem: Placement

- Where should process memories be placed?
  - Topic: “Classic” memory management
- How does the compiler model memory?
  - Topic: Logical memory model
- How to deal with limited physical memory?
  - Topics: Virtual memory, paging

Memory Management

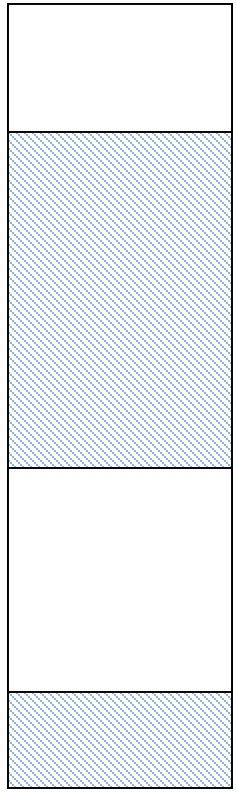
- Physical memory starts as one big empty space.



Memory Management

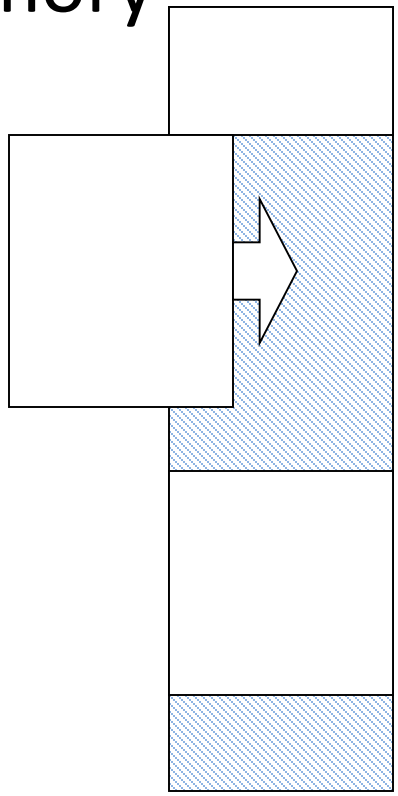
- Physical memory starts as one big empty space.

- Processes need to be in memory to execute.



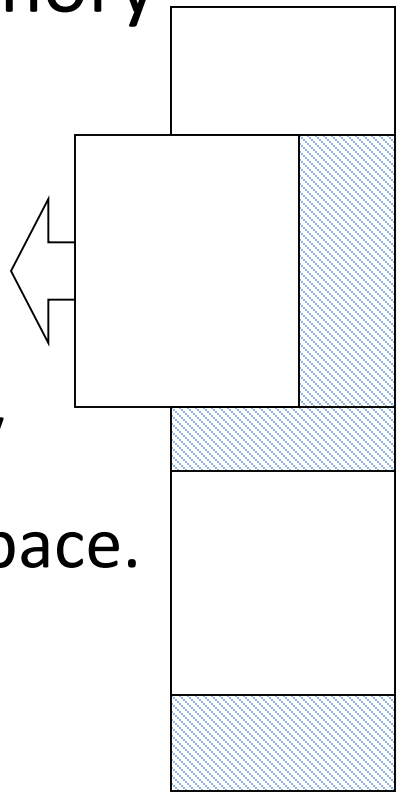
Memory Management

- Physical memory starts as one big empty space.
- When creating process, allocate memory
  - Find space that can contain process
  - Allocate region within that gap
  - Typically, leaves a (smaller) free space



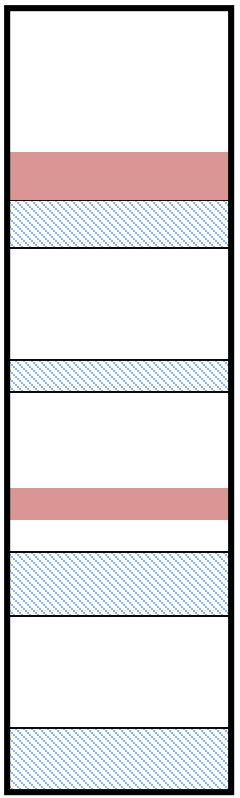
Memory Management

- Physical memory starts as one big empty space.
- When creating process, allocate memory
  - Find space that can contain process
  - Allocate region within that gap
  - Typically, leaves a (smaller) free space
- When process exits, free its memory
  - Creates a gap in the physical address space.
  - If next to another gap, coalesce.



Fragmentation

- Eventually, memory becomes fragmented
  - After repeated allocations/de-allocations
- Internal fragmentation
  - Unused space within process
  - Cannot be allocated to others
  - Can come in handy for growth
- External fragmentation
  - Unused space outside any process (gaps)
  - Can be allocated (too small/not useful?)



Which form of fragmentation is easiest for the OS to reduce/eliminate? Why?

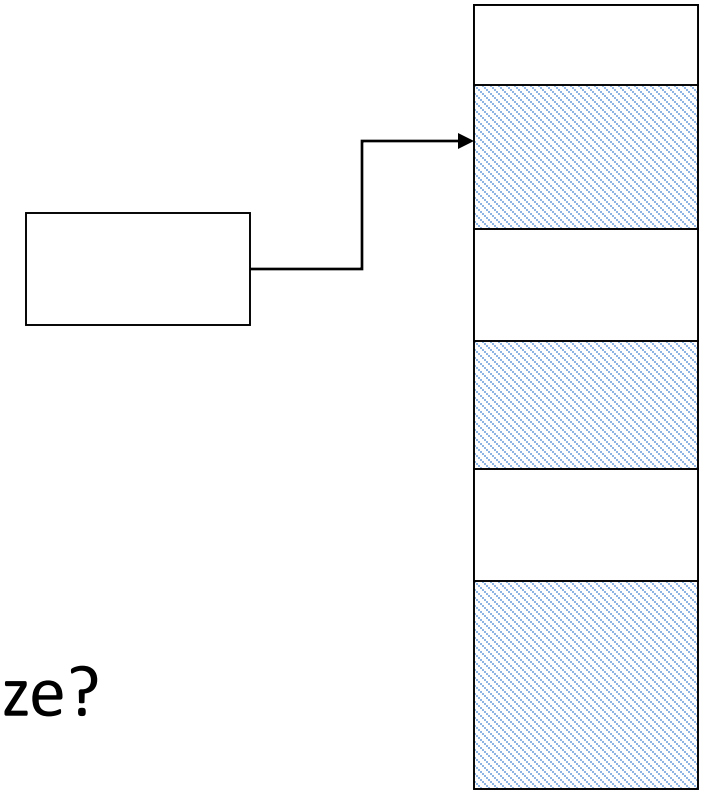
A. Internal fragmentation

B. External fragmentation

C. Neither

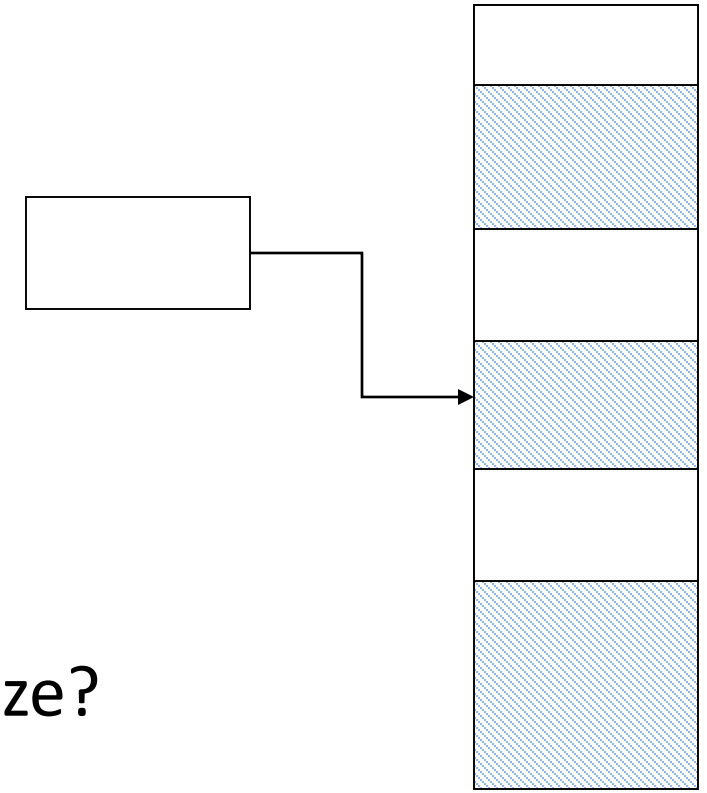
Placing Memory

- When searching for space, what if there are multiple options?
- Algorithms
  - *First (or next) fit*
  - Best fit
  - Worst fit
- Complication
  - Is region fixed or variable size?



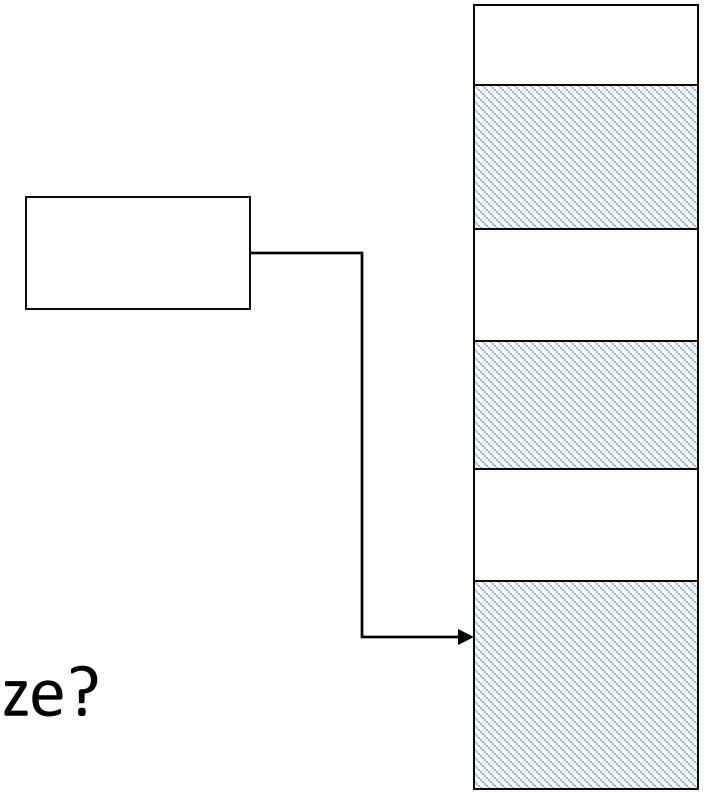
Placing Memory

- When searching for space, what if there are multiple options?
- Algorithms
  - First (or next) fit
  - *Best fit*
  - Worst fit
- Complication
  - Is region fixed or variable size?



Placing Memory

- When searching for space, what if there are multiple options?
- Algorithms
  - First (or next) fit
  - Best fit
  - *Worst fit*
- Complication
  - Is region fixed or variable size?



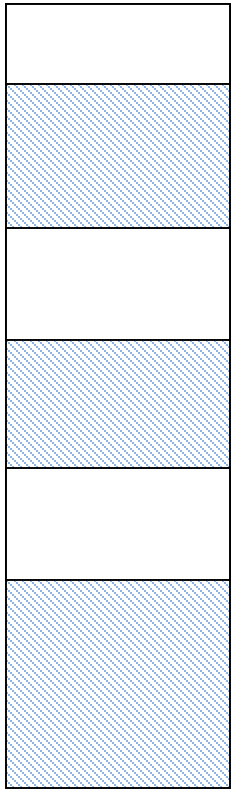
Which memory allocation algorithm leaves the smallest fragments (external)?

A. first-fit

B. worst-fit

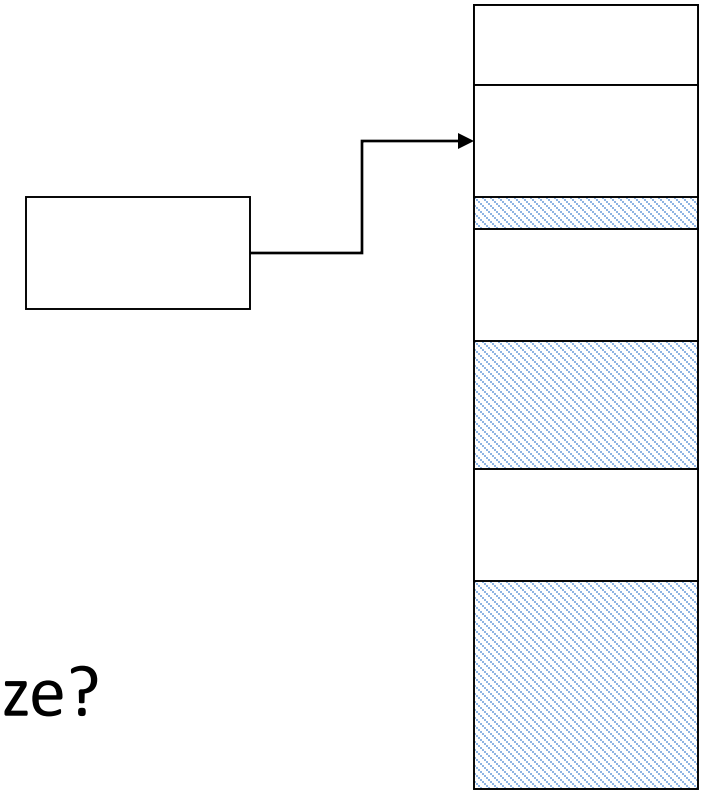
C. best-fit

Is leaving small fragments a good thing or a bad thing?



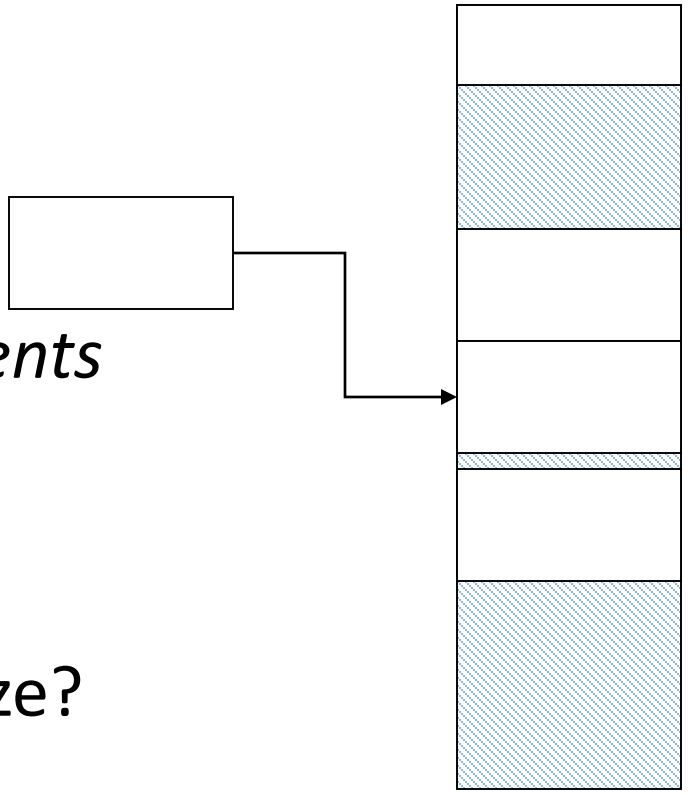
Placing Memory

- When searching for space, what if there are multiple options?
- Algorithms
  - *First (or next) fit: fast*
  - Best fit
  - Worst fit
- Complication
  - Is region fixed or variable size?



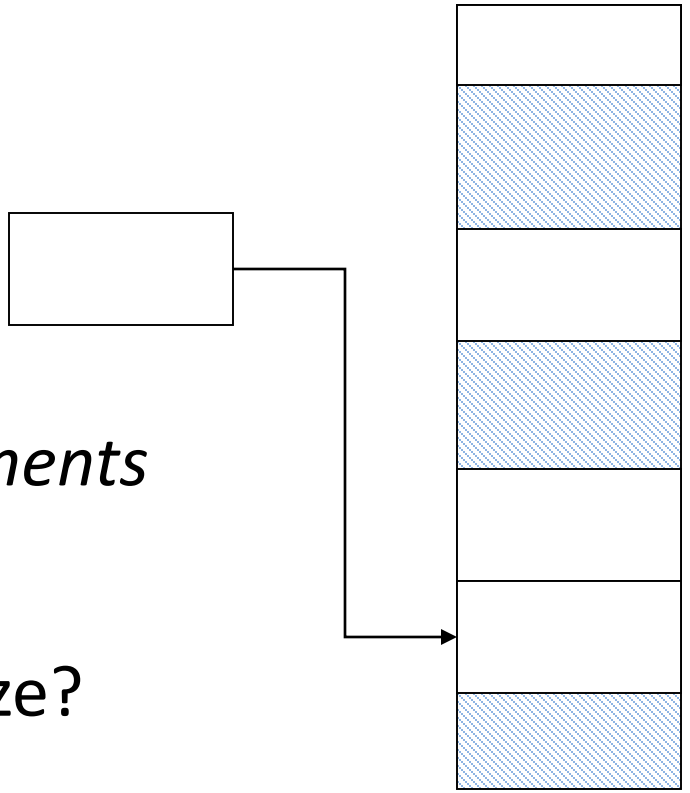
Placing Memory

- When searching for space, what if there are multiple options?
- Algorithms
  - First (or next) fit
  - *Best fit: leaves small fragments*
  - Worst fit
- Complication
  - Is region fixed or variable size?



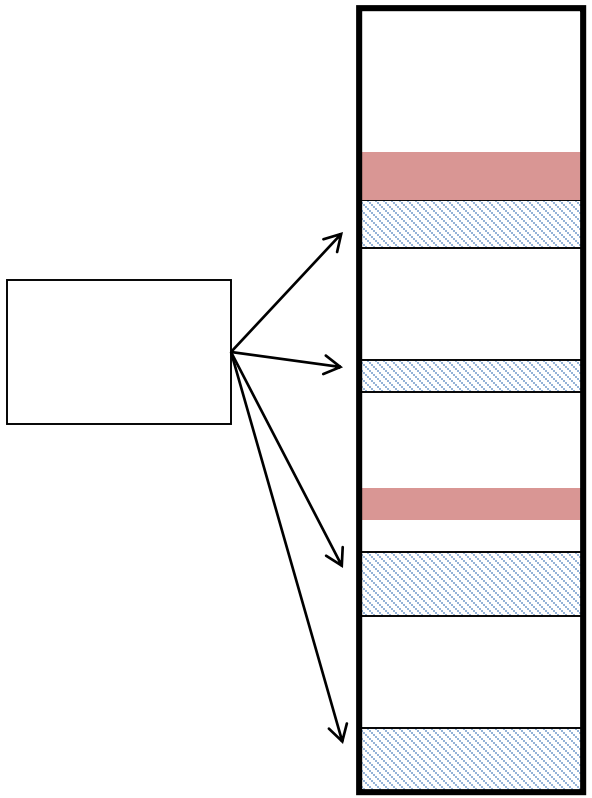
Placing Memory

- When searching for space, what if there are multiple options?
- Algorithms
  - First (or next) fit
  - Best fit
  - *Worst fit: leaves large fragments*
- Complication
  - Is region fixed or variable size?



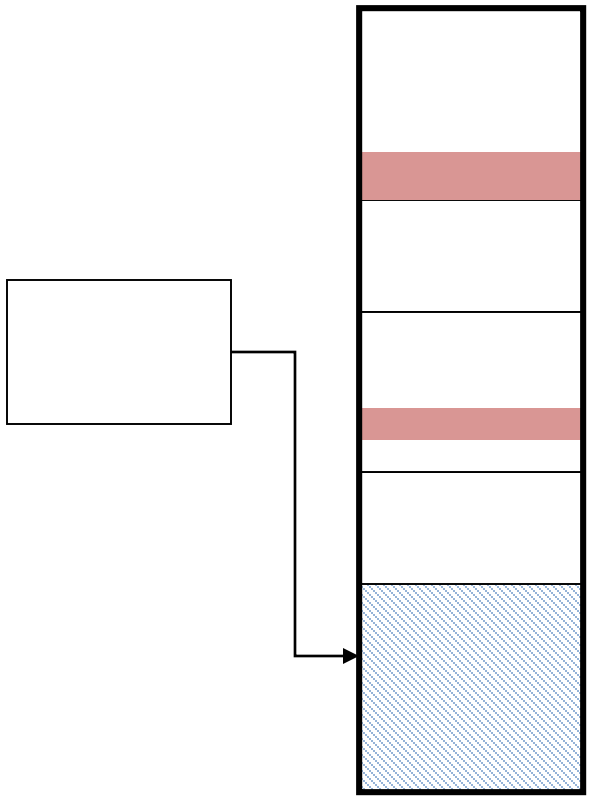
What if it doesn't fit?

- There may still be significant unused space
  - External fragments
  - Internal fragments
- Approaches



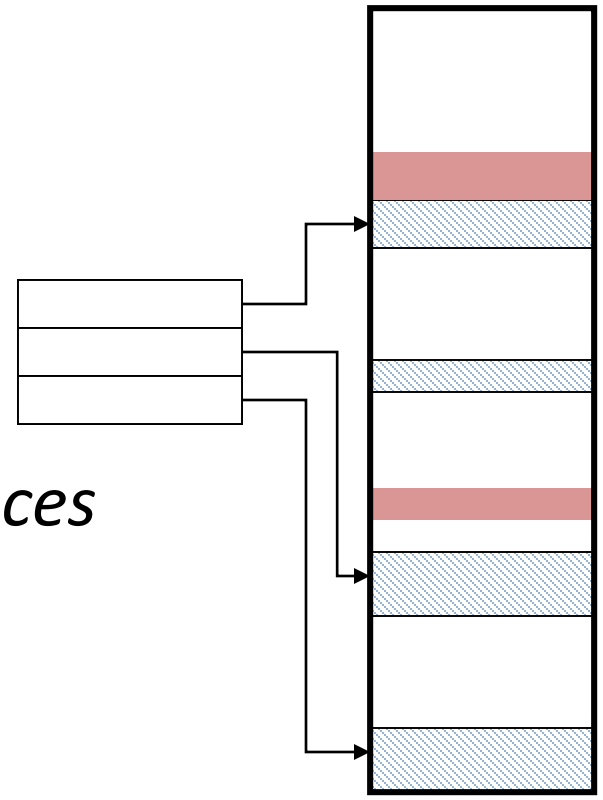
What if it doesn't fit?

- There may still be significant unused space
  - External fragments
  - Internal fragments
- Approaches
  - *Compaction*



What if it doesn't fit?

- There may still be significant unused space
  - External fragments
  - Internal fragments
- Approaches
  - Compaction
  - *Break process memory into pieces*
    - Easier to fit.
    - More state to keep track of.



Problem Summary: Placement

- When placing process, it may be hard to find a large enough free region in physical memory.
- Fragmentation makes this harder over time (free pieces get smaller, spread out)

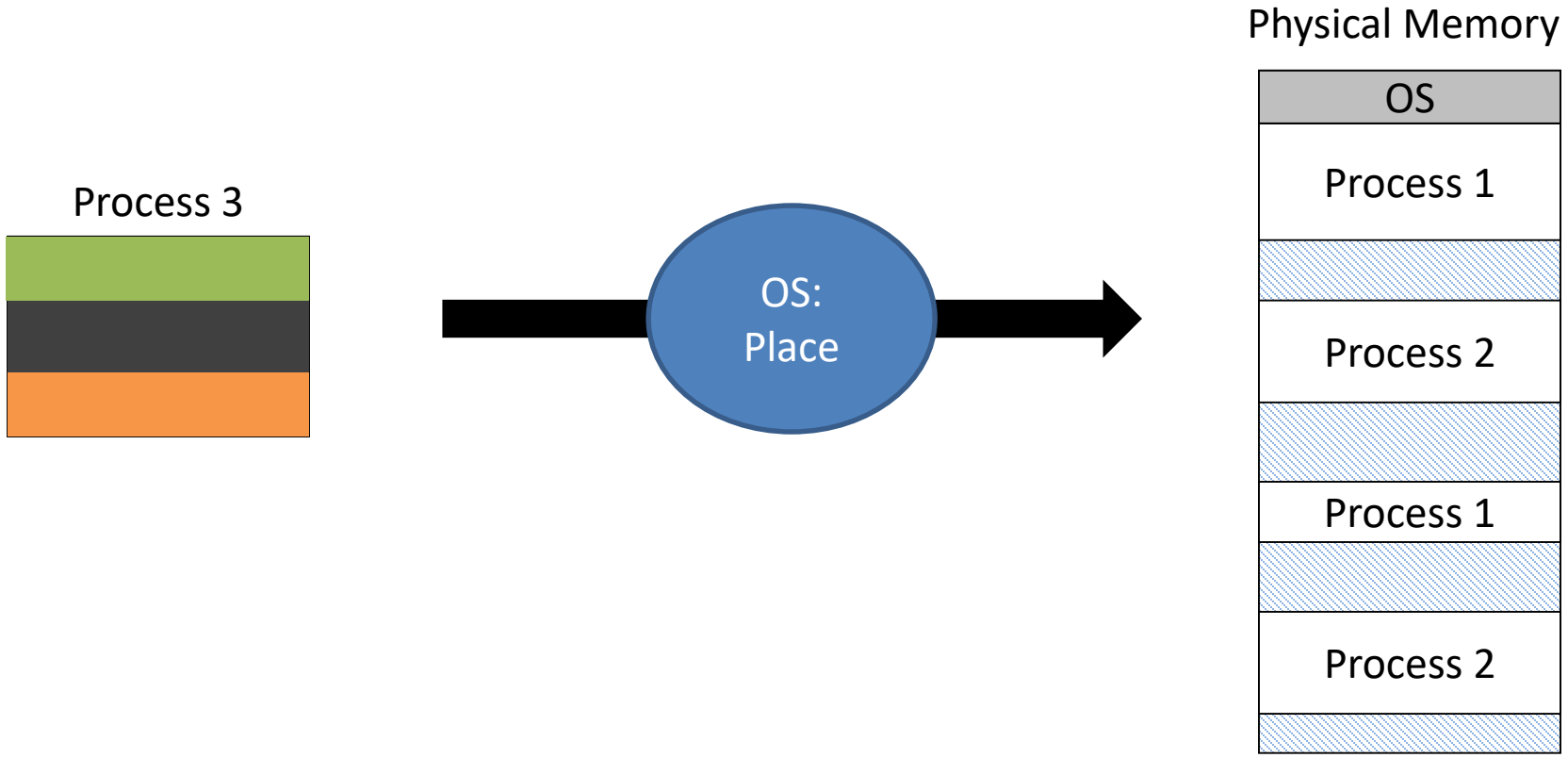
- General solution: don’t require all of a process’s memory to be in one piece!

Problem Summary: Placement

- General solution: don't require all of a process's memory to be in one piece!

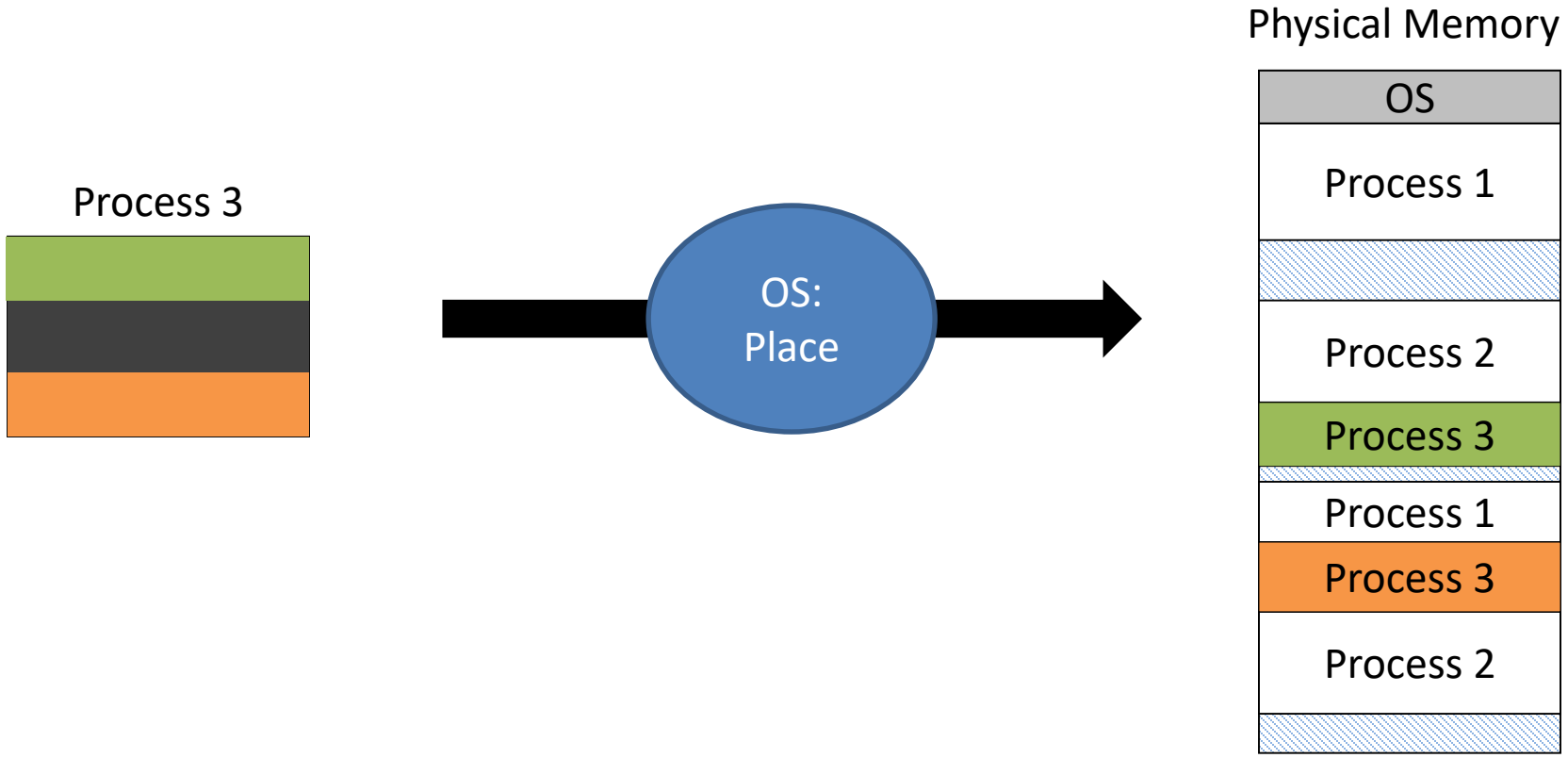
Problem Summary: Placement

- General solution: don't require all of a process's memory to be in one piece!



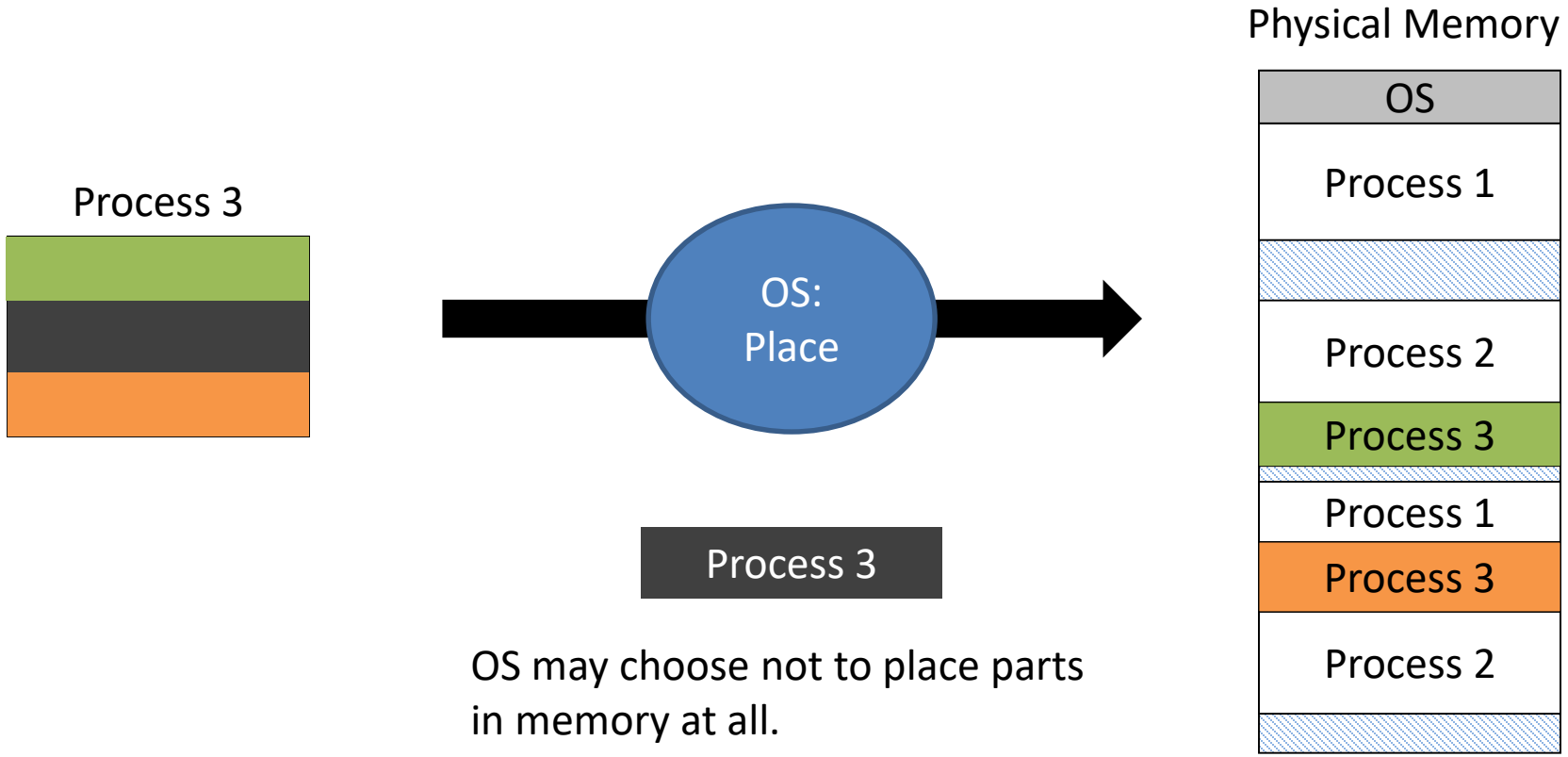
Problem Summary: Placement

- General solution: don't require all of a process's memory to be in one piece!



Problem Summary: Placement

- General solution: don't require all of a process's memory to be in one piece!

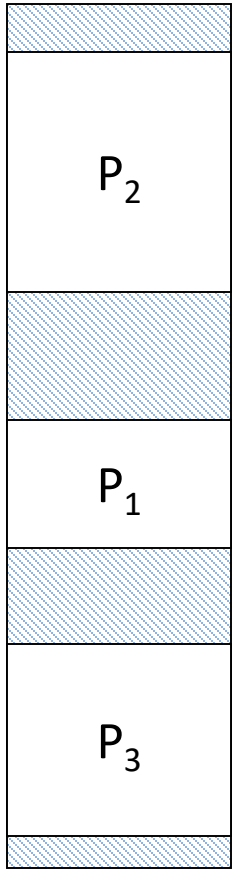


Problem: Addressing

- Where should process memories be placed?
  - Topic: “Classic” memory management
- How does the compiler model memory?
  - Topic: Logical memory model
- How to deal with limited physical memory?
  - Topics: Virtual memory, paging

(More) Problems with Memory Cohabitation

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



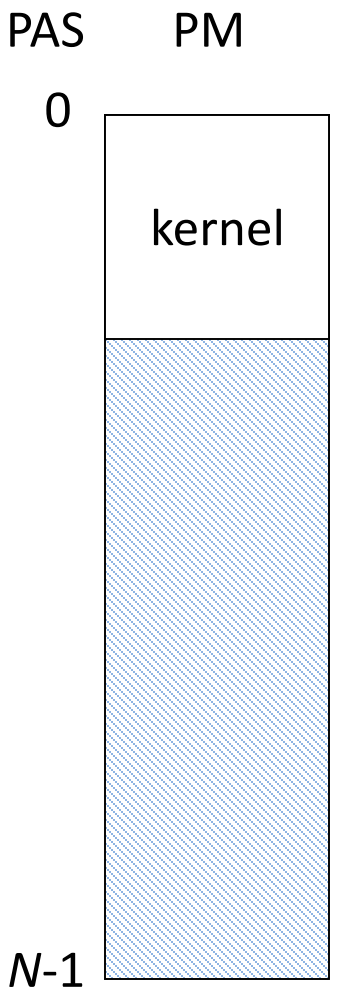
Compiler's View of Memory

- Compiler generates memory addresses
  - Needs empty region for text, data, stack
  - Ideally, very large to allow data and stack to grow
  - Alternative: three independent empty regions

- Without abstractions compiler would need to know…
  - Physical memory size
  - Where to place data (e.g., stack at high end)
    - Must avoid allocated regions in memory

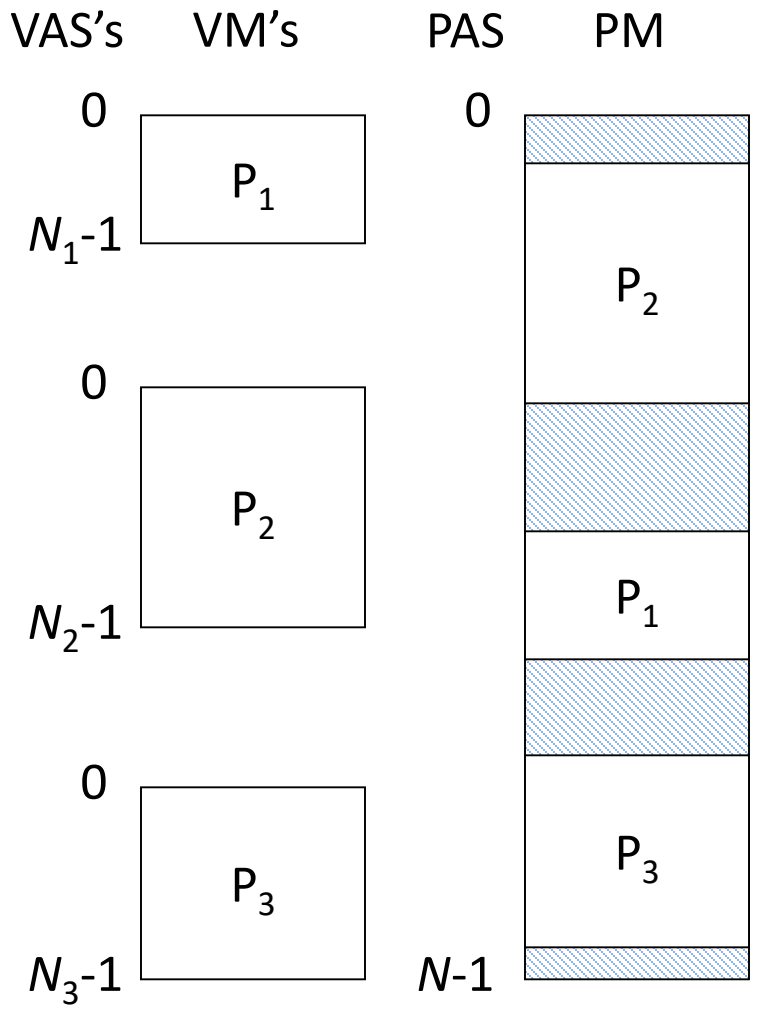
Address Spaces

- 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



Virtual vs. Physical Addressing

- Virtual addresses
  - Assumes separate memory starting at 0
  - Compiler generated
  - Independent of location in physical memory
- OS: Map virtual to physical



When should we perform the mapping from virtual to physical address? Why?

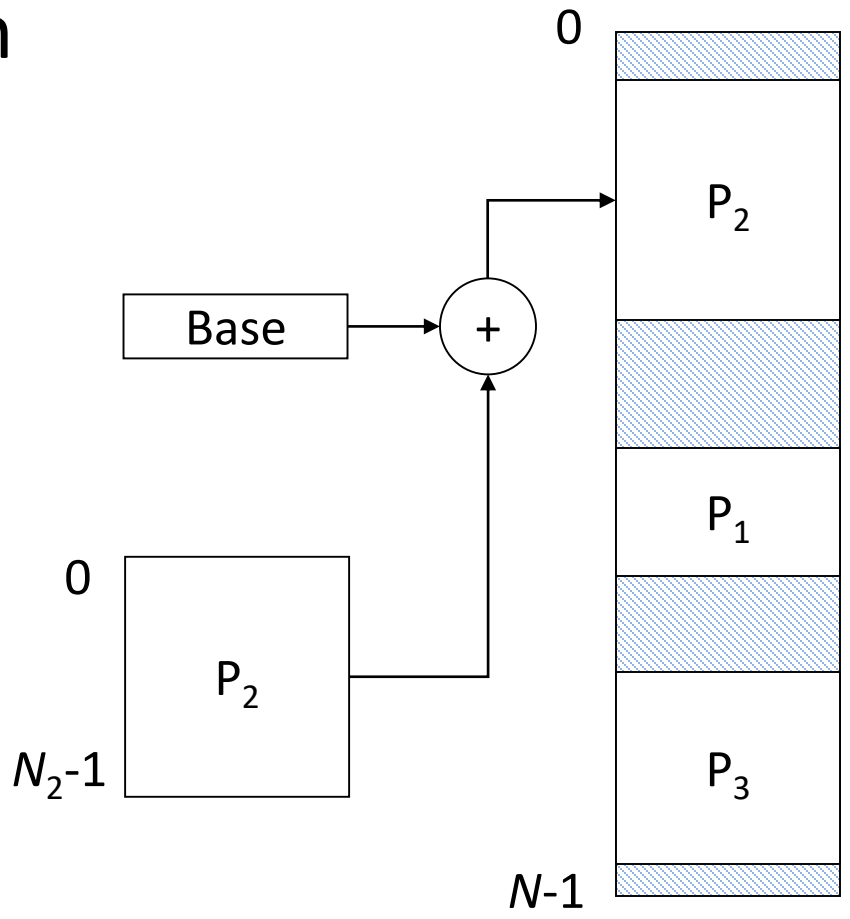
A. When the process is initially loaded: convert all the addresses to physical addresses

B. When the process is running: map the addresses as they're used.

C. Perform the mapping at some other time. When?

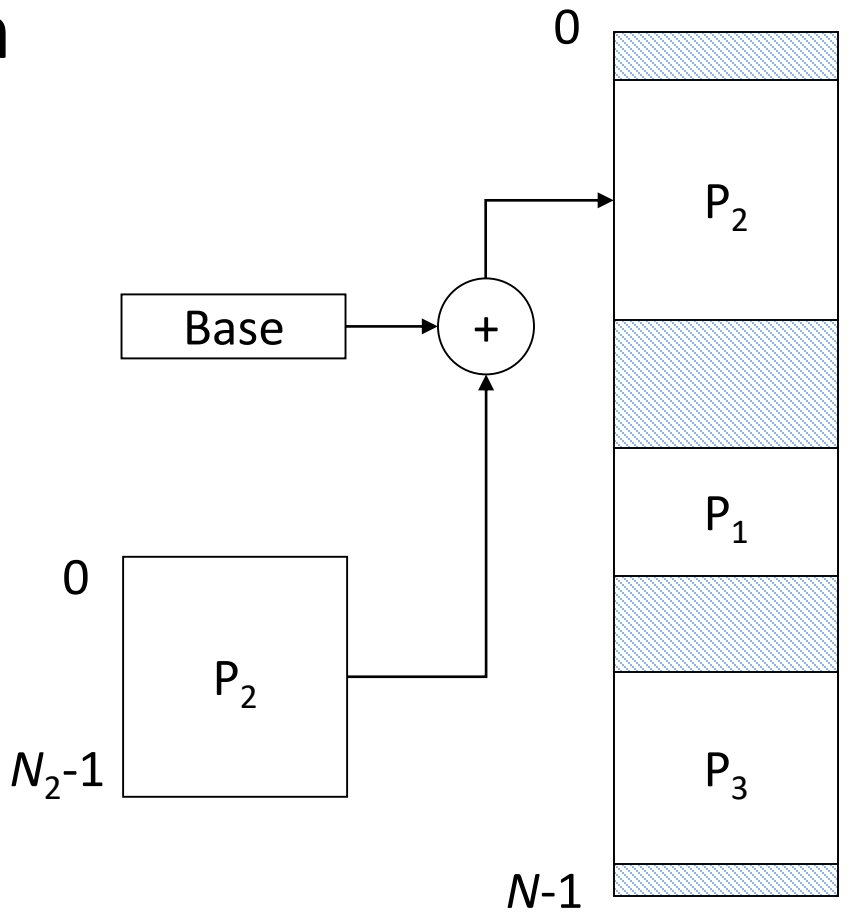
Hardware for Virtual Addressing

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



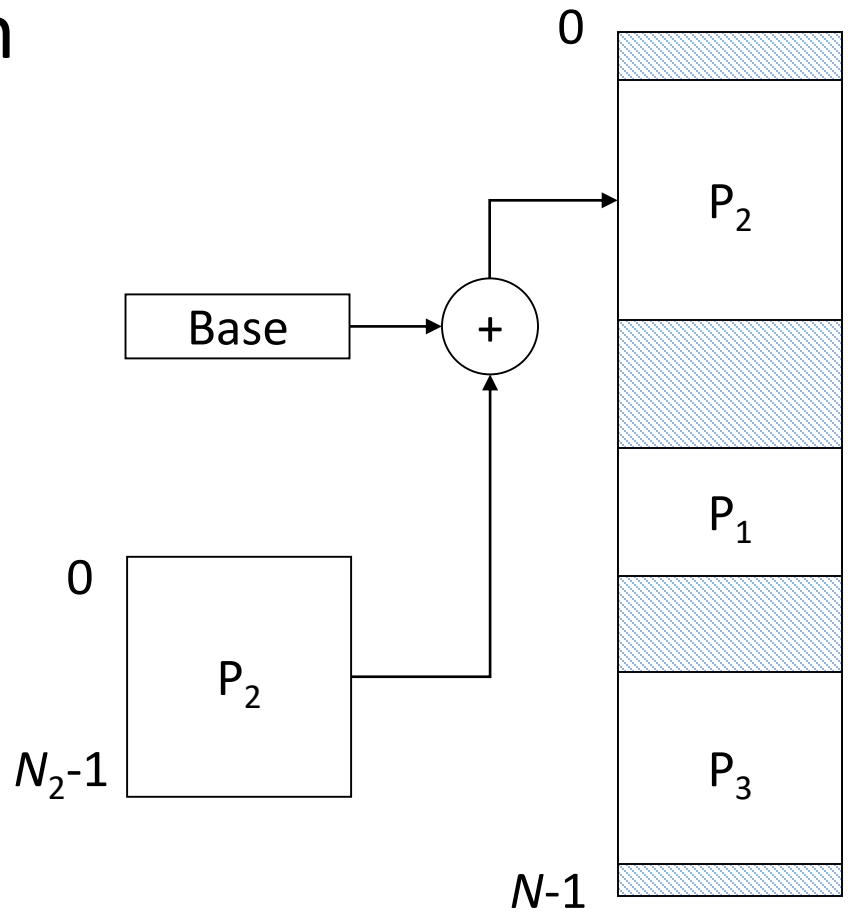
Hardware for Virtual Addressing

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



Hardware for Virtual Addressing

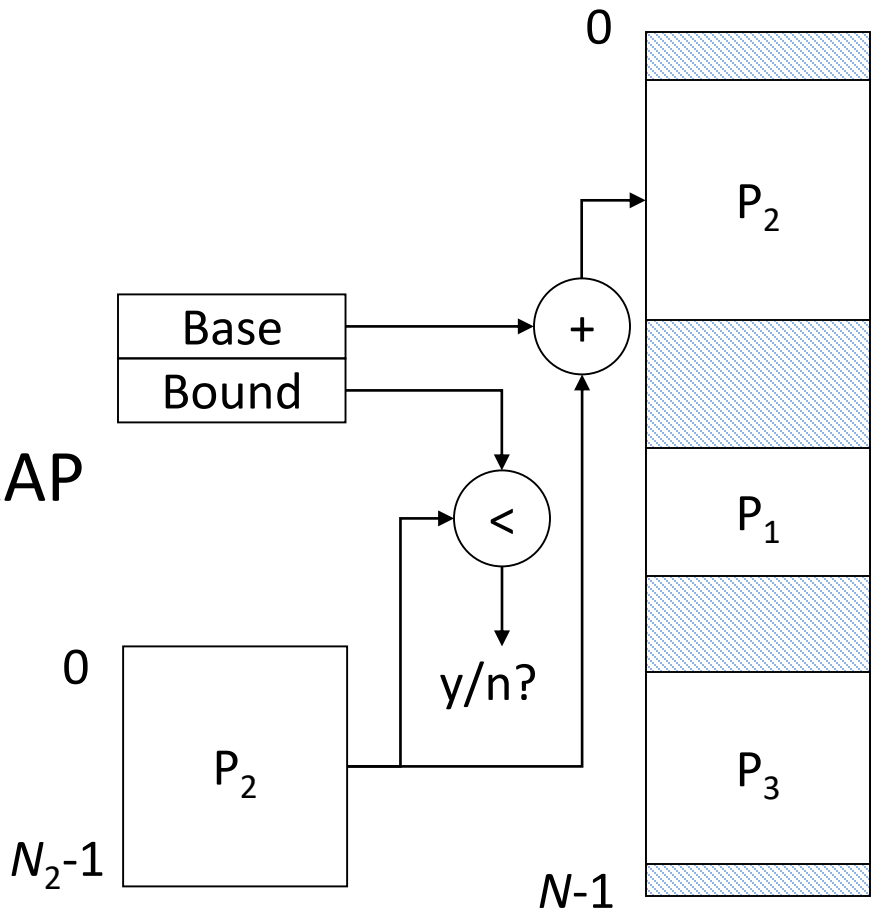
- Base register filled with start address
- To translate address, add base
- Achieves relocation
- To move process: change base
- Protection?



Protection

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

When would we need to update these base & bound registers?



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

Problem Summary: Addressing

- Compiler has no idea where, in physical memory, the process's data will be.
- Compiler generates instructions to access VAS.

- General solution: OS must translate process's VAS accesses to the corresponding physical memory location.

Problem Summary: Addressing

- General solution: OS must translate process's VAS accesses to the corresponding physical memory location.

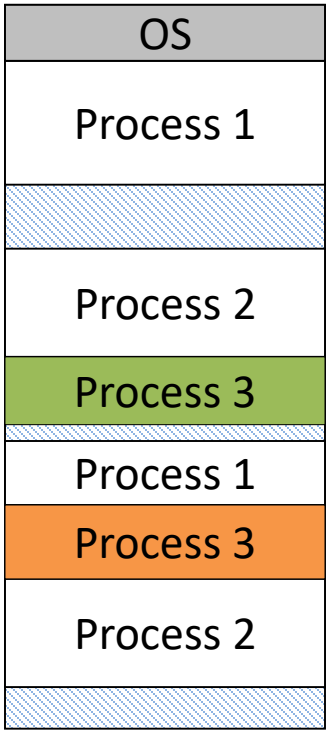
When the process tries to access a virtual address, the OS translates it to the corresponding physical address.

Process 3

movl (address 0x74), %eax

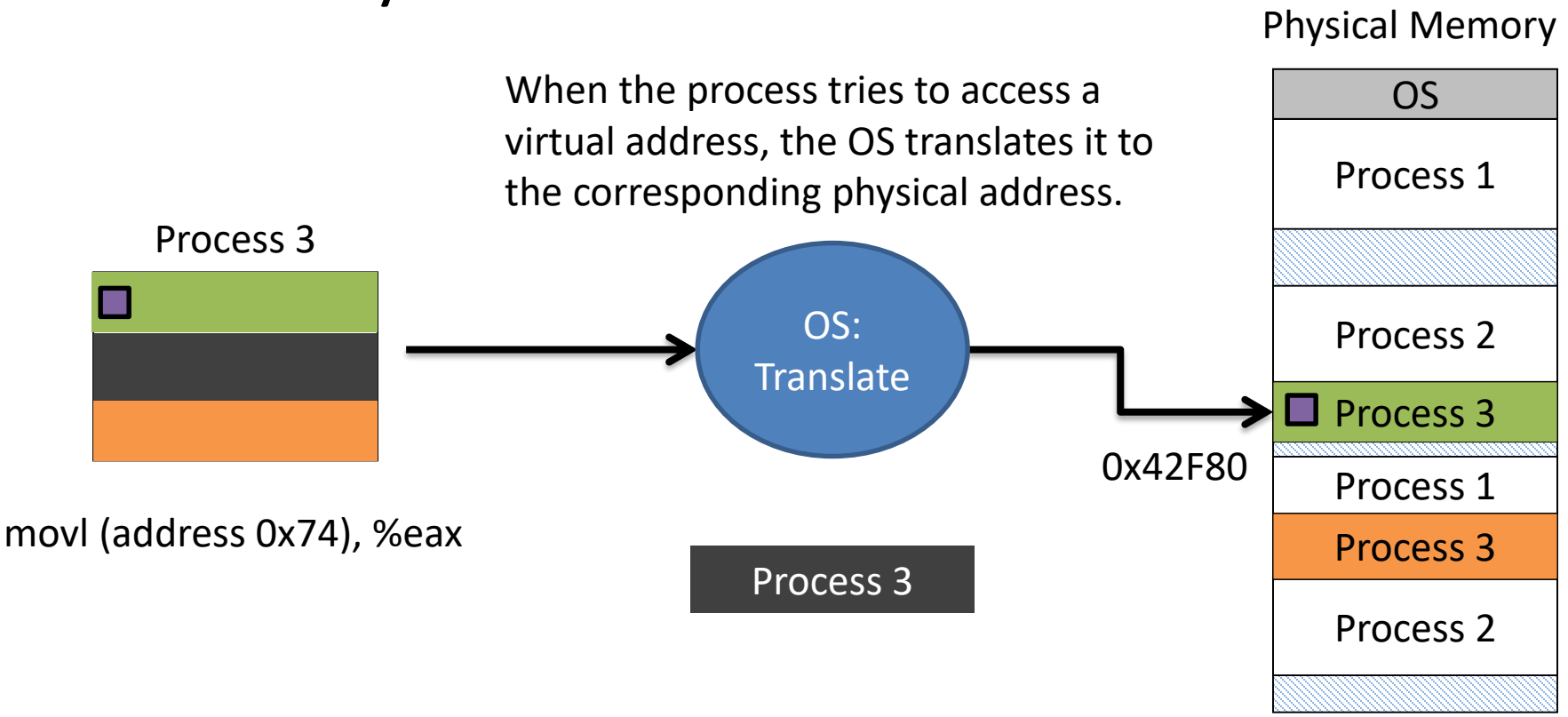
Process 3

Physical Memory



Problem Summary: Addressing

- General solution: OS must translate process's VAS accesses to the corresponding physical memory location.



Let's combine these ideas:

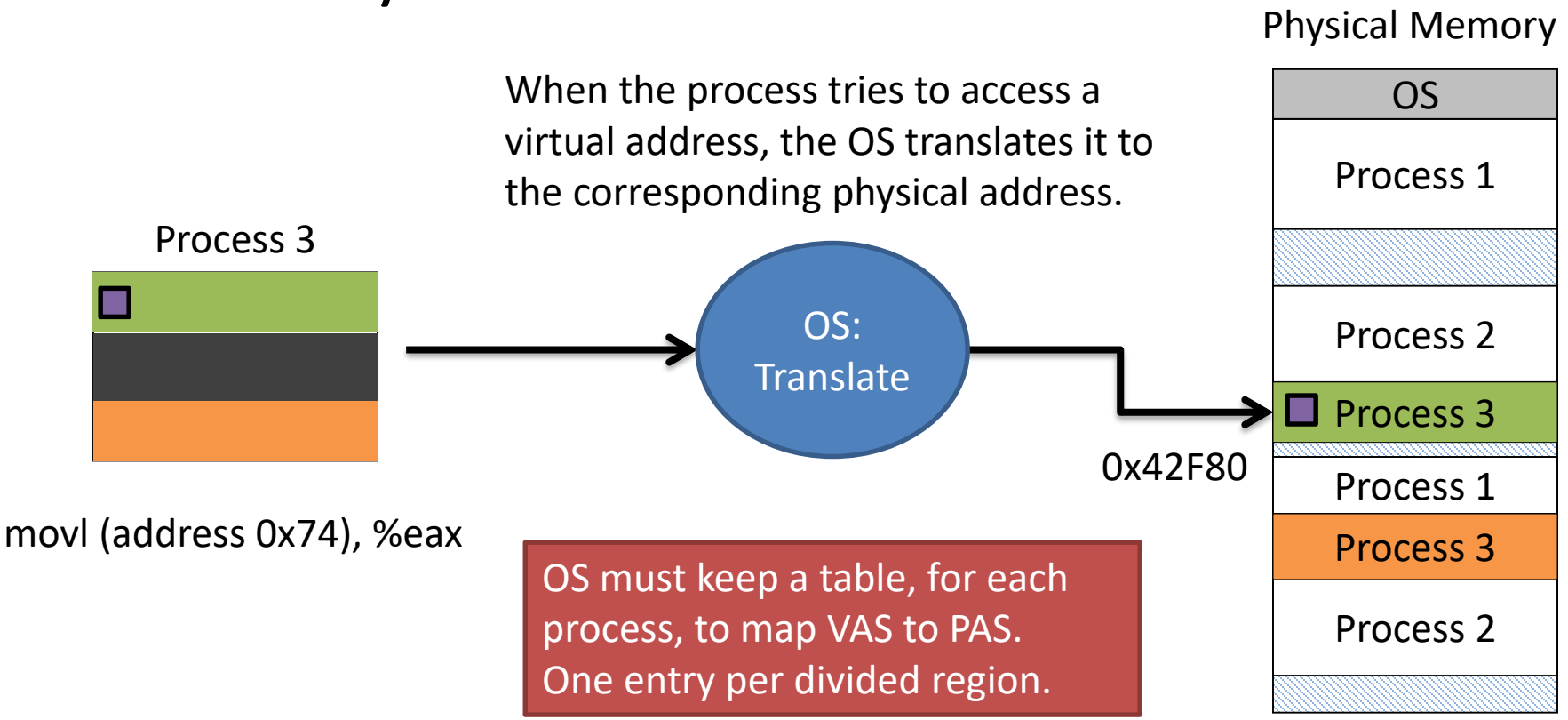
1. Allow process memory to be divided up into multiple pieces.

2. Keep state in OS (+ hardware/registers) to map from virtual addresses to physical addresses.

Result: Keep a table to store the mapping of each region.

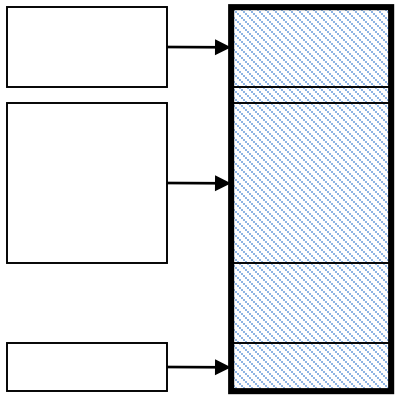
Problem Summary: Addressing

- General solution: OS must translate process's VAS accesses to the corresponding physical memory location.

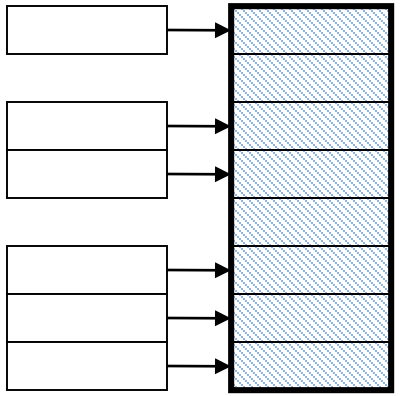


Two (Real) Approaches

- Segmented address space/memory
- Partition address space and memory into segments
- Segments are generally different sizes



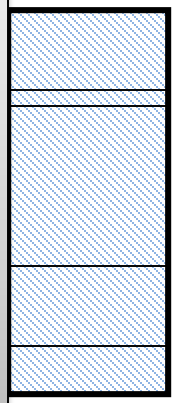
- Paged address space/memory
- Partition address space and memory into pages
- All pages are the same size



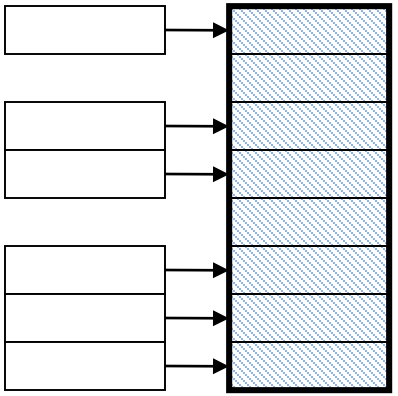
Two (Real) Approaches

- Se
- Pa
  int
- Se

In this class, we're only going to look at paging, the most common method today.



- Paged address space/memory
- Partition address space and memory into pages
- All pages are the same 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.

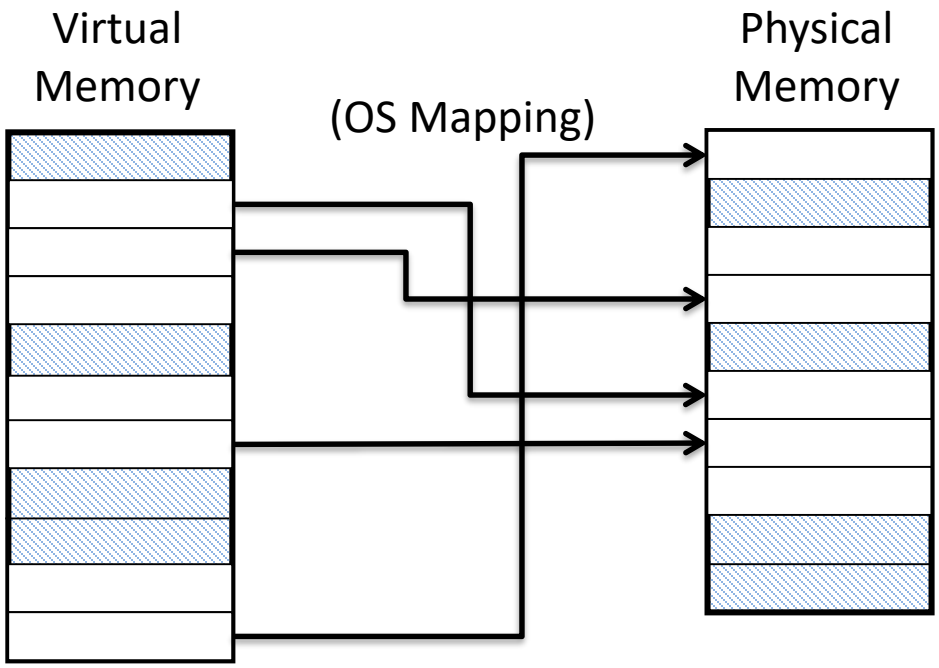
Main Idea

- 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?

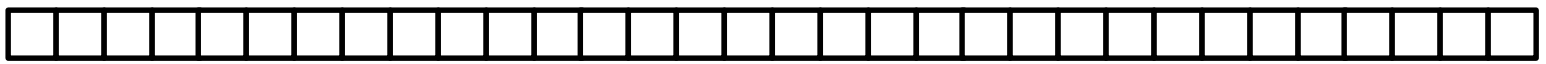
External: goes away. No more awkwardly-sized, unusable gaps.

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.

Example: 32-bit virtual addresses

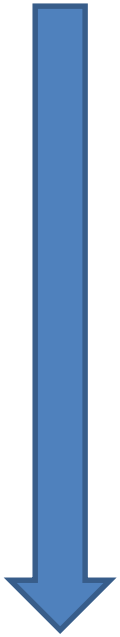
- Suppose we have 8-KB (8192-byte) pages.
- We need enough bits to individually address each byte in the page.
  - How many bits do we need to address 8192 items?
  - $2^{13} = 8192$, so we need 13 bits.
  - Lowest 13 bits: offset within page.
- Remaining 19 bits: page number.

Address Partitioning

Virtual address:

We'll call these bits *p*.

We'll call these bits *i*.

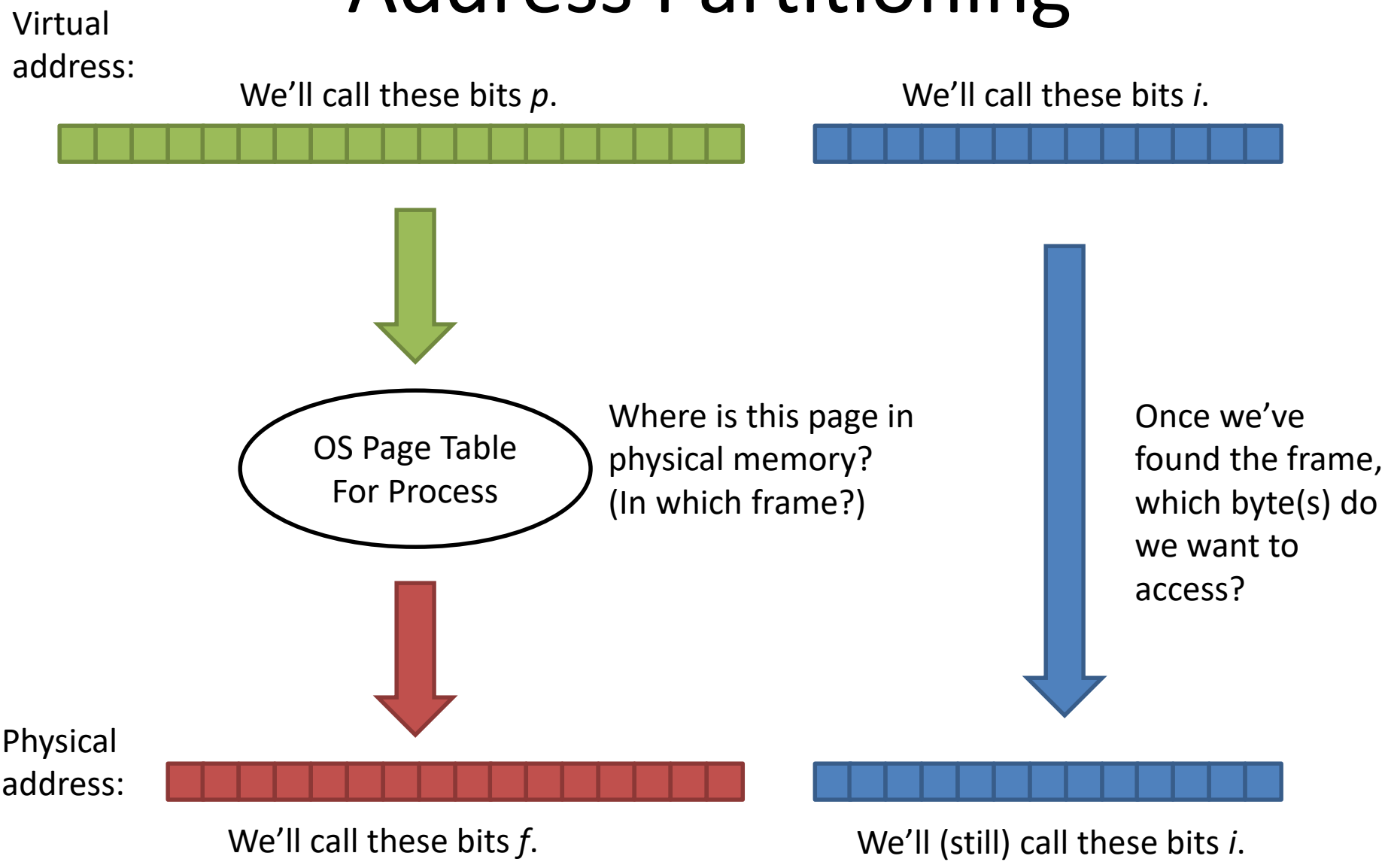


Once we've found the frame, which byte(s) do we want to access?

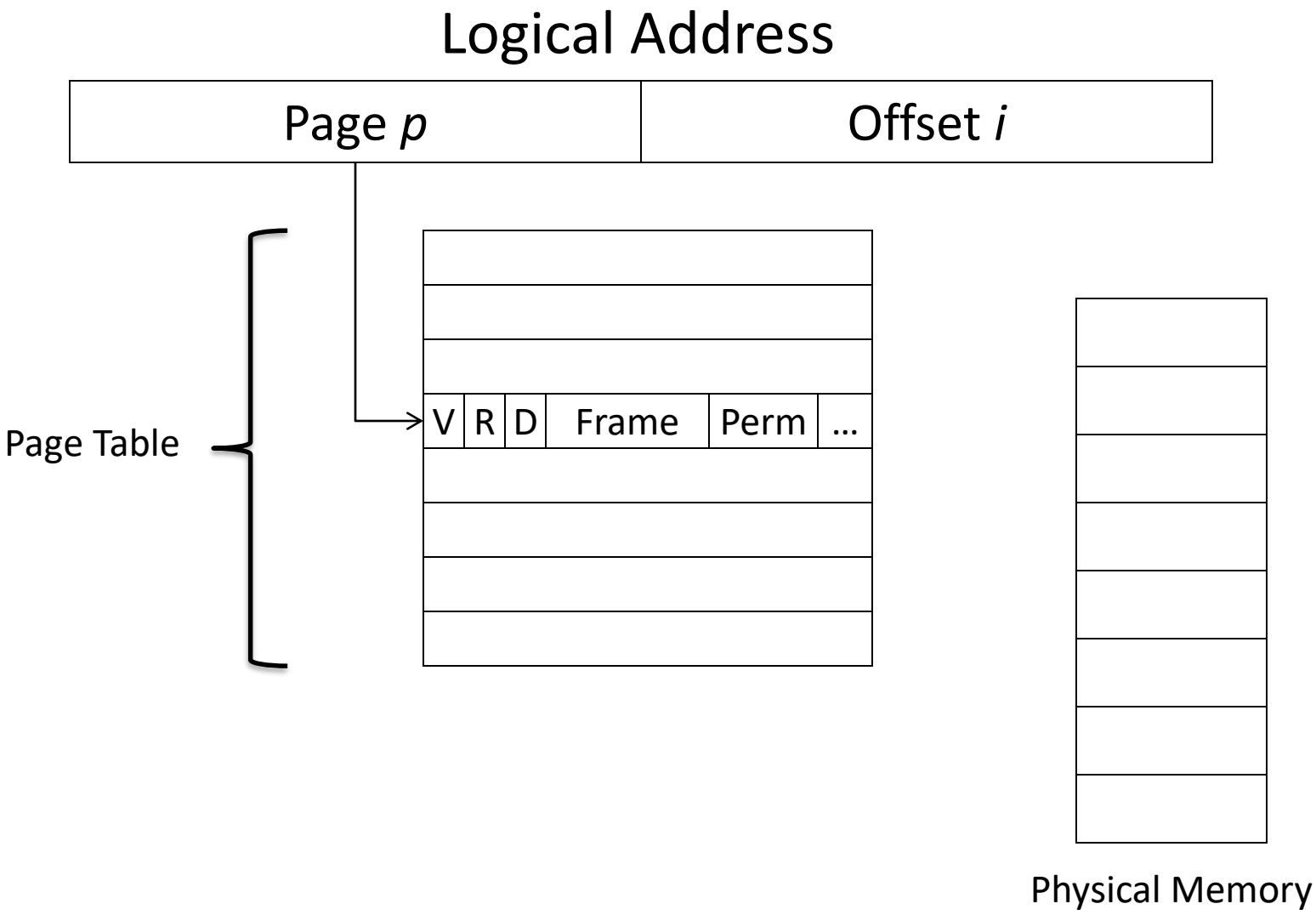
Physical address:

We'll (still) call these bits *i*.

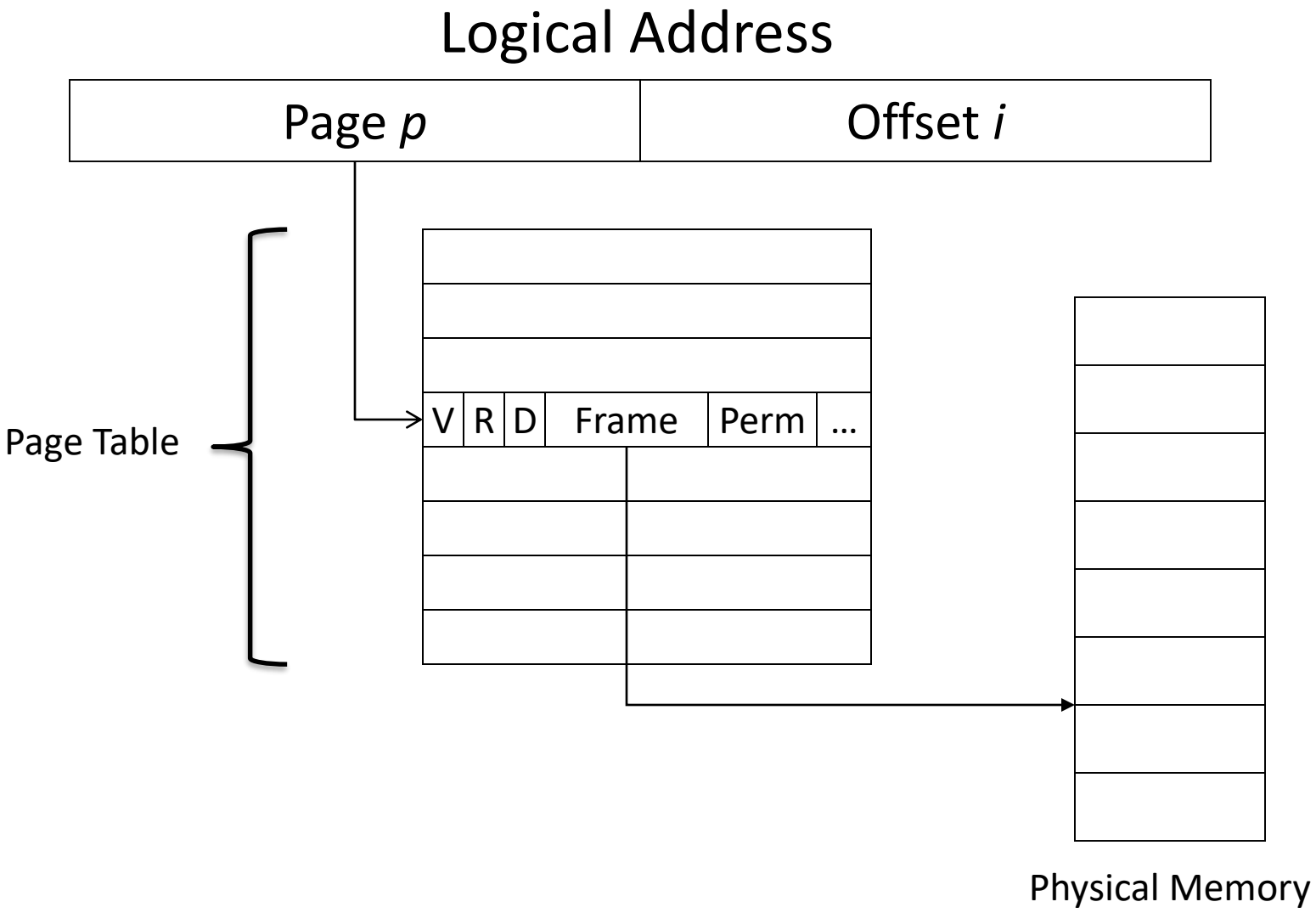
Address Partitioning



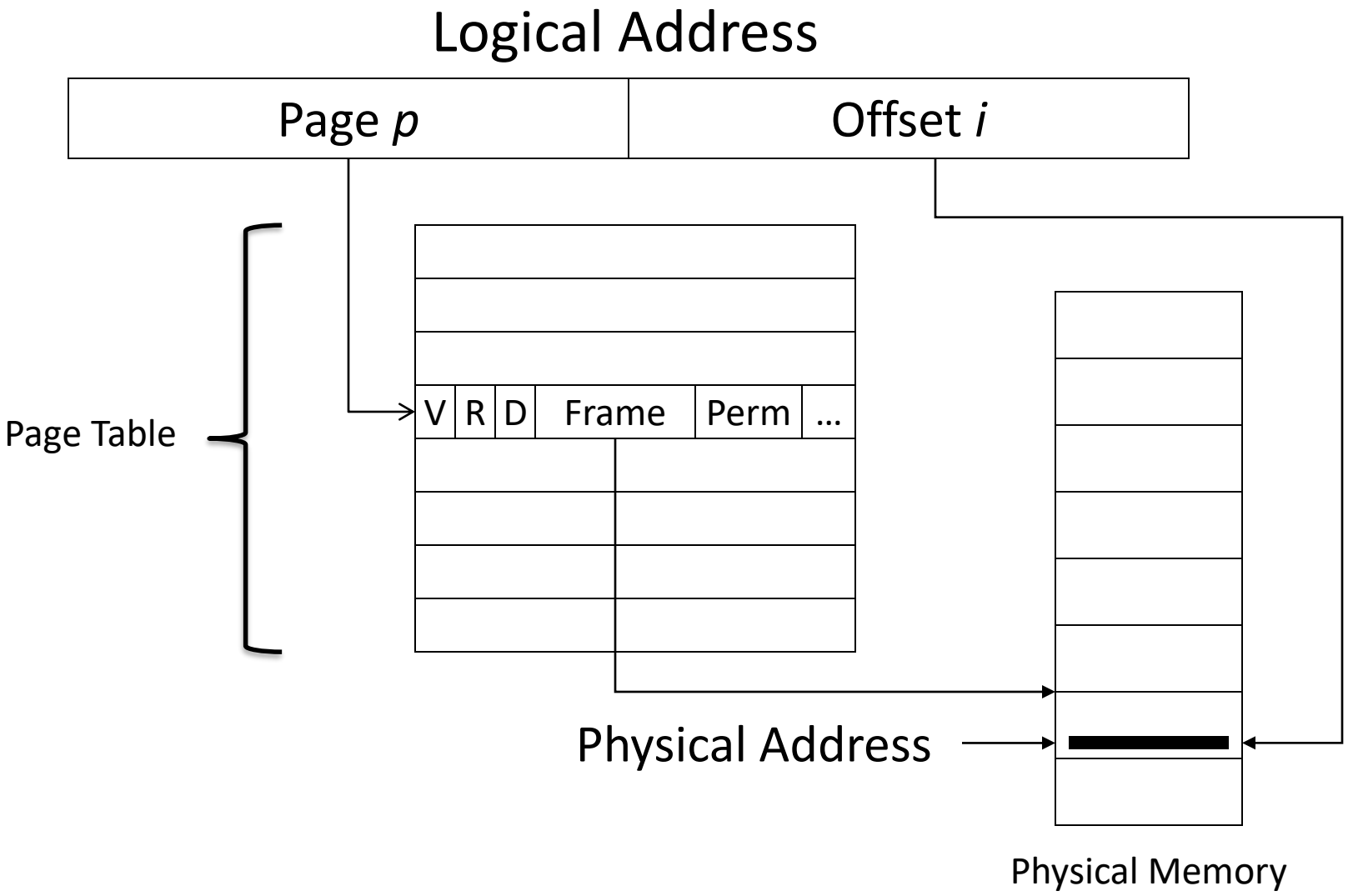
Address Translation



Address Translation

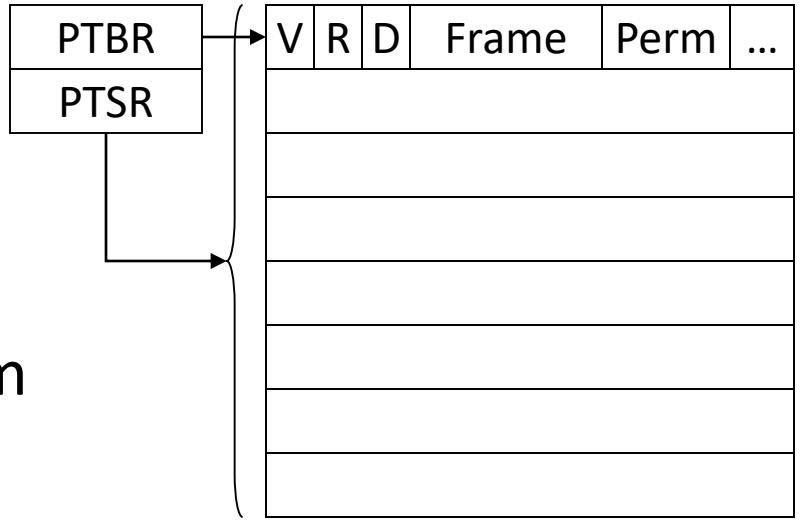


Address Translation



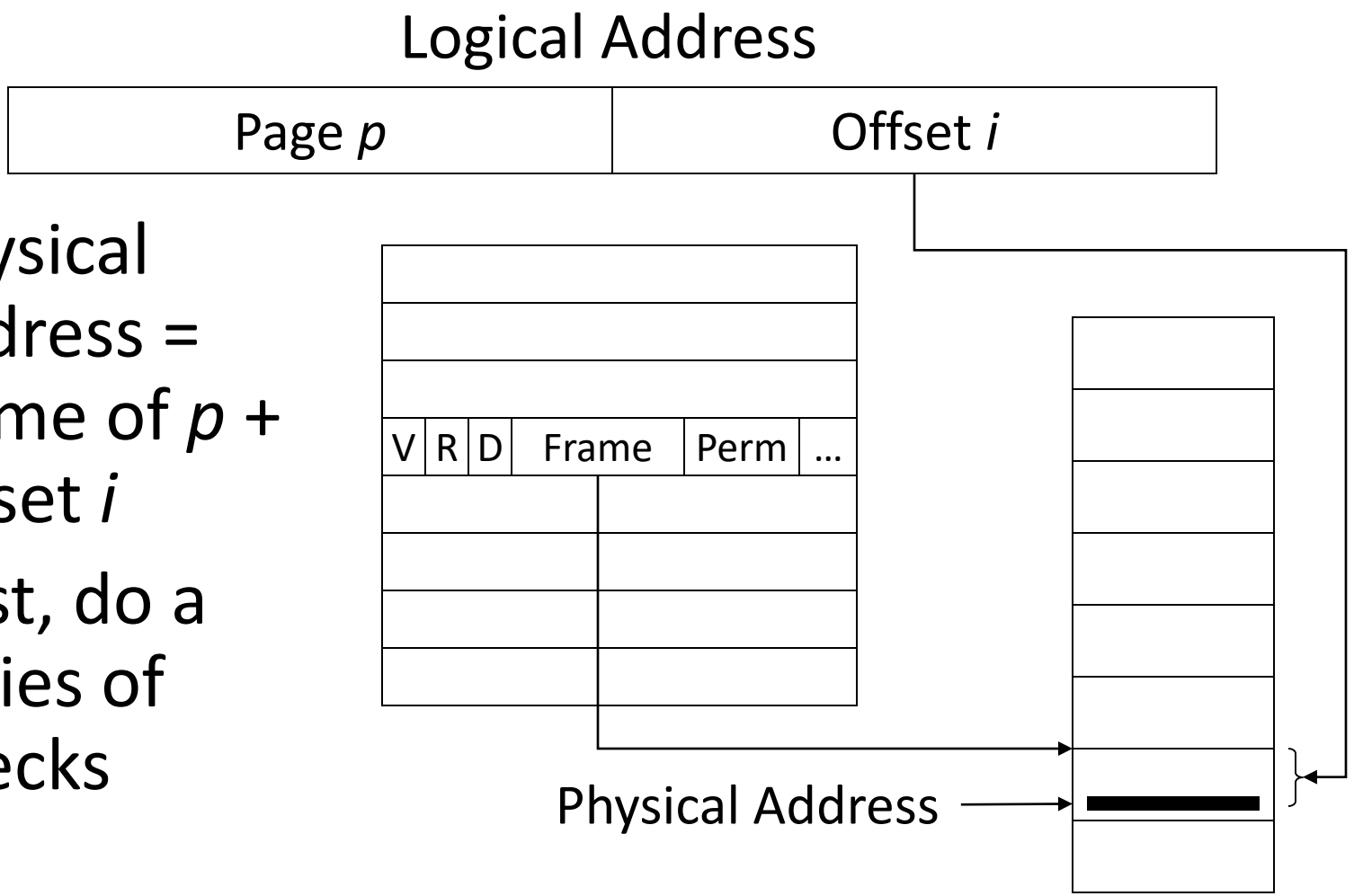
Page Table

- 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

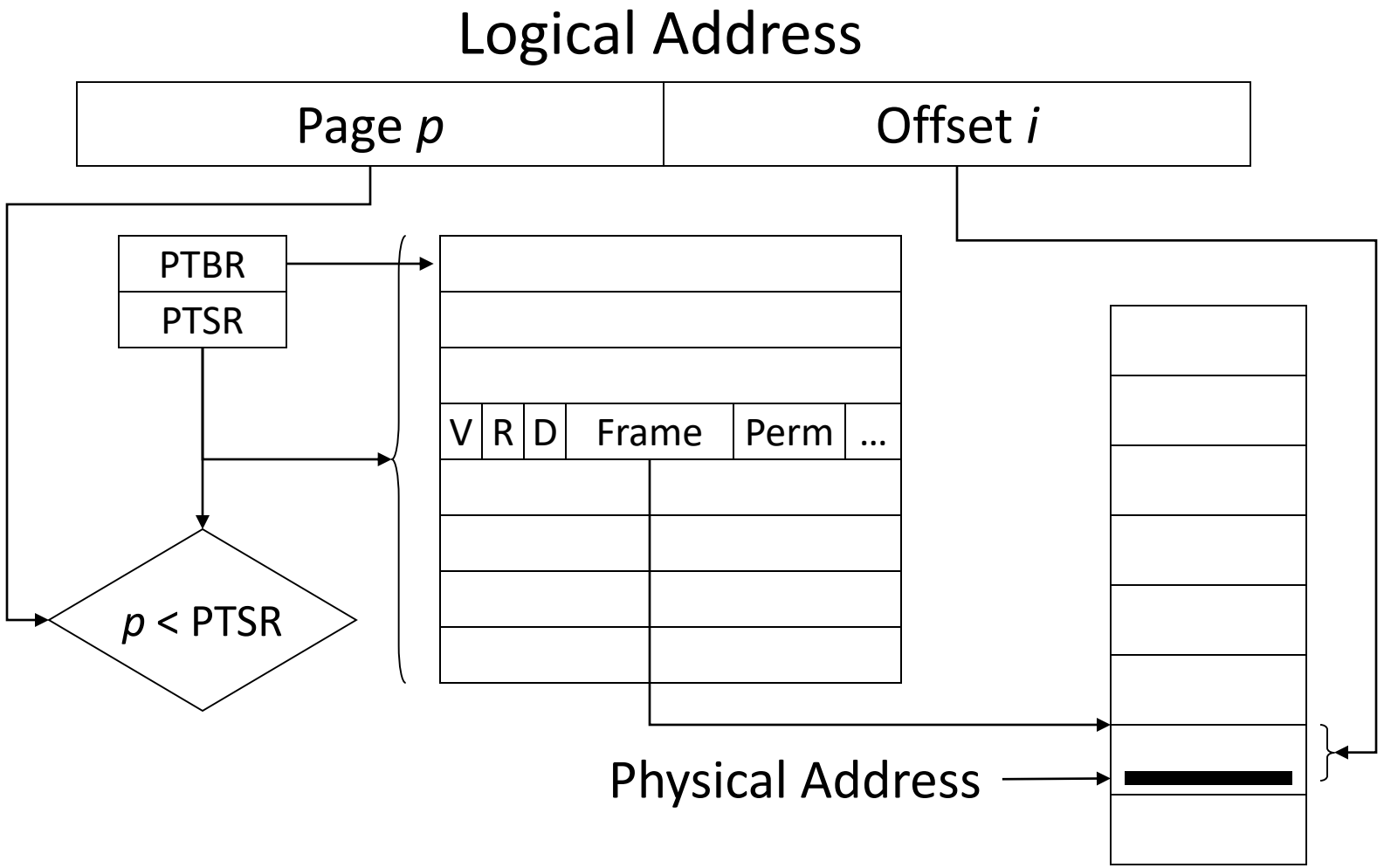


Address Translation

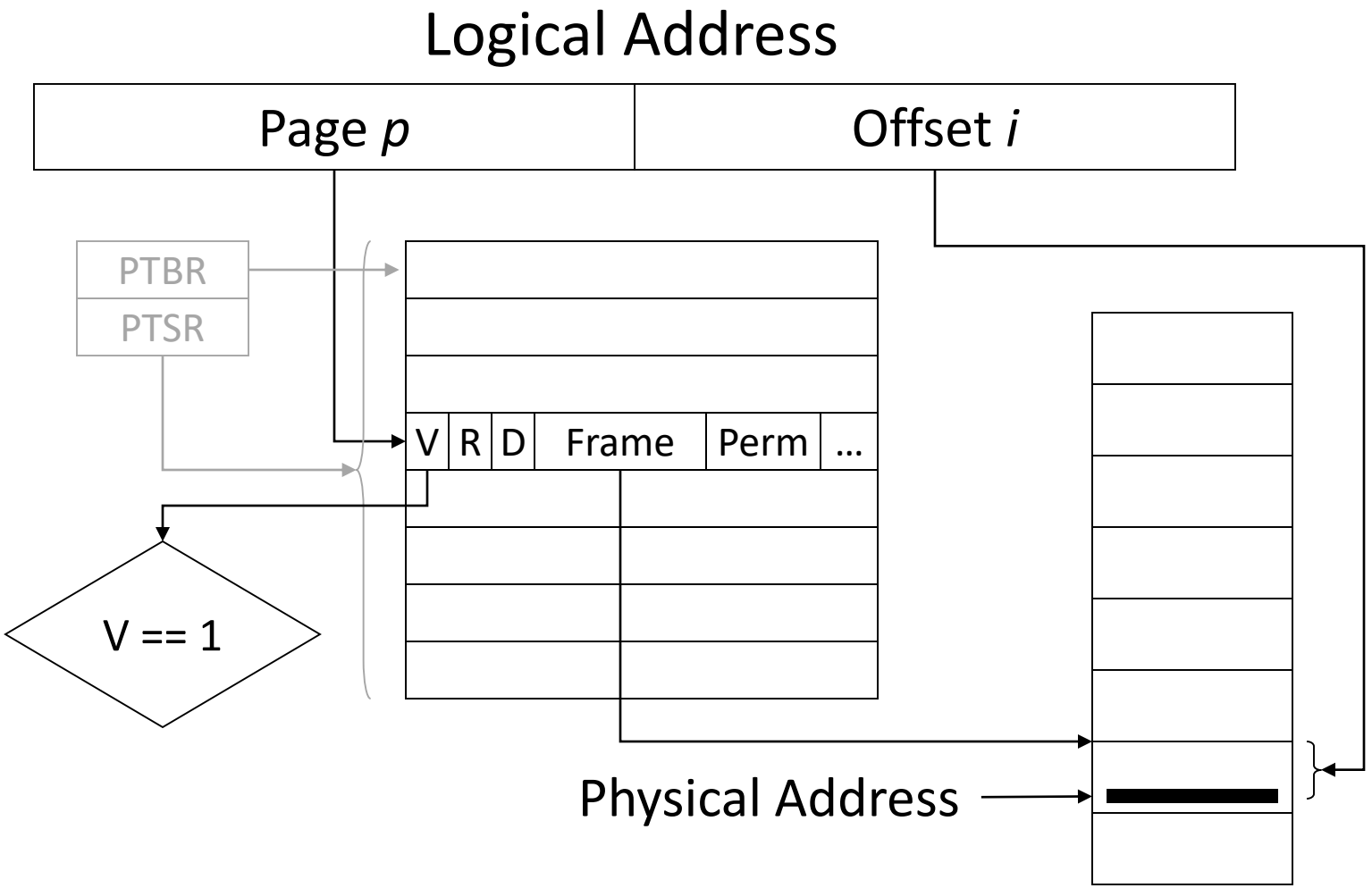
- Physical address = frame of *p* + offset *i*
- First, do a series of checks



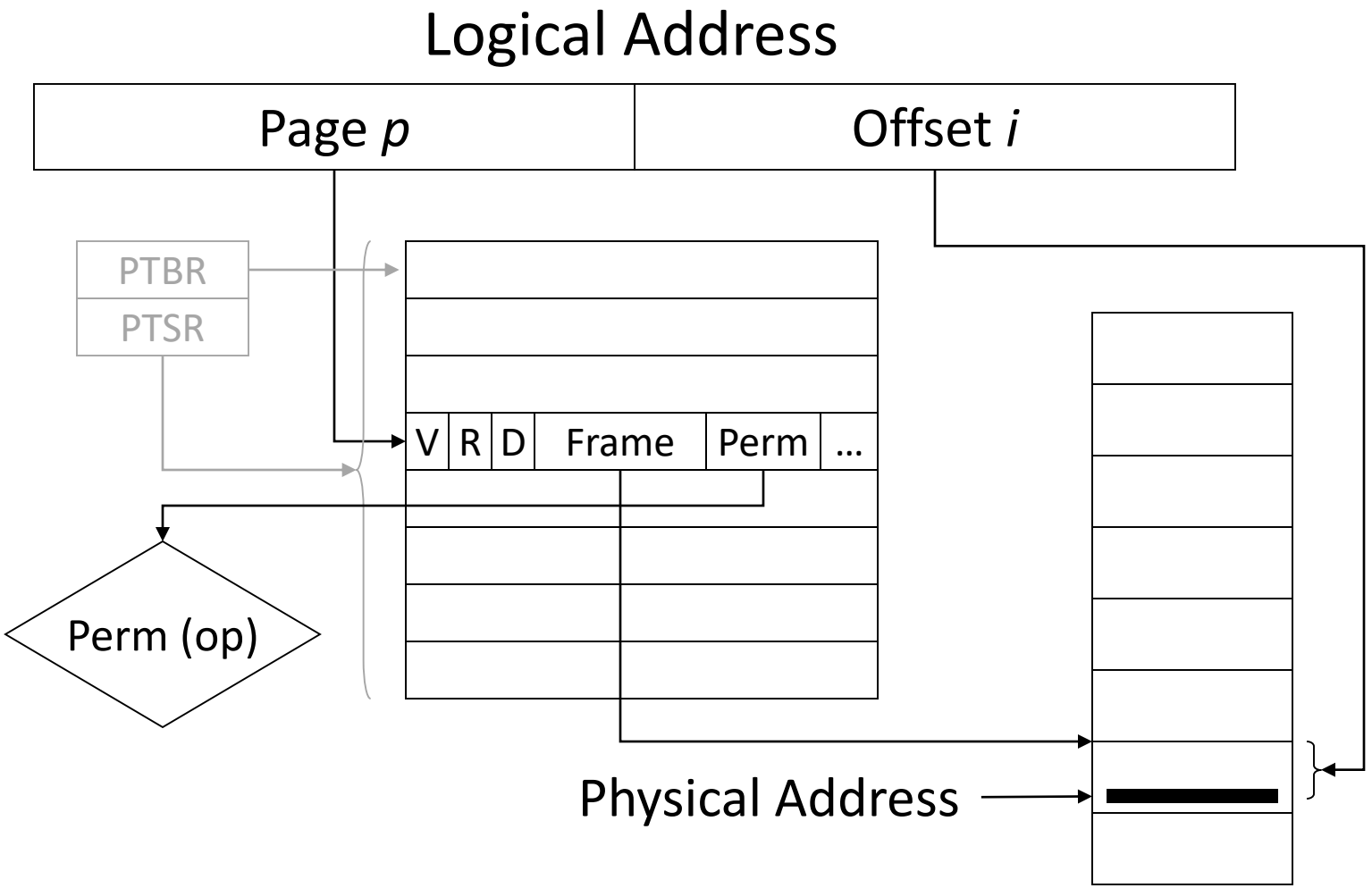
Check if Page *p* is Within Range



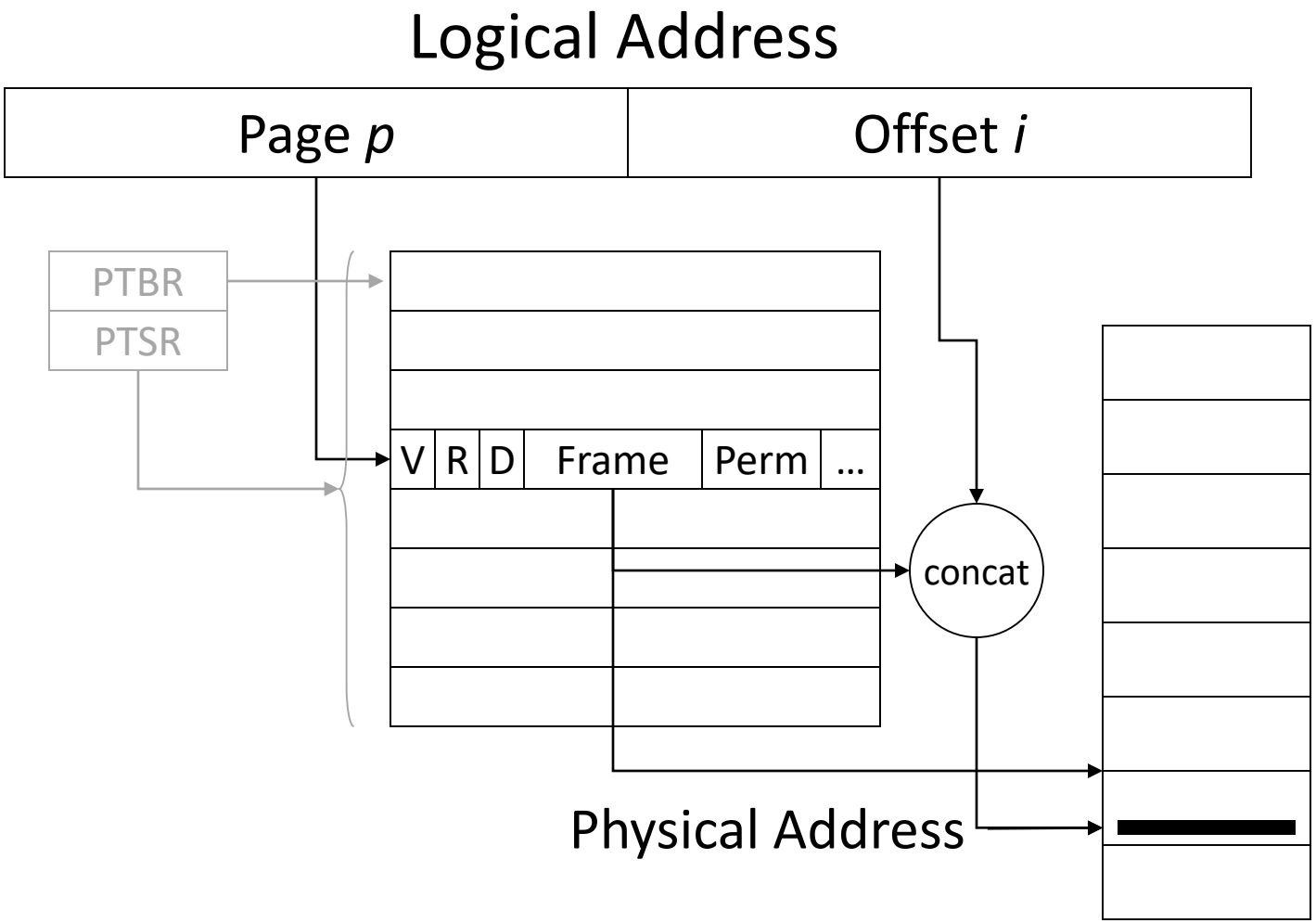
Check if Page Table Entry *p* is Valid



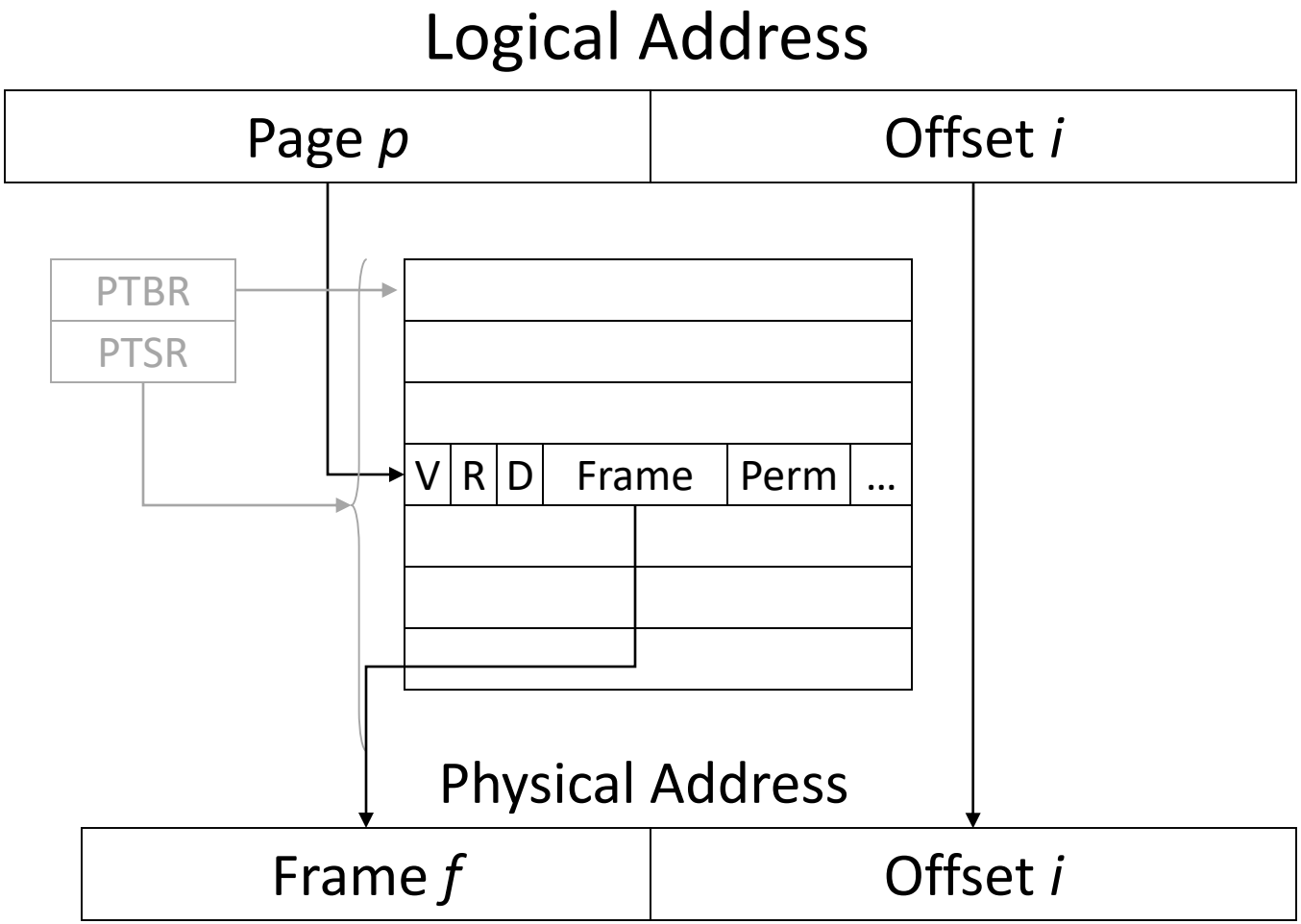
Check if Operation is Permitted



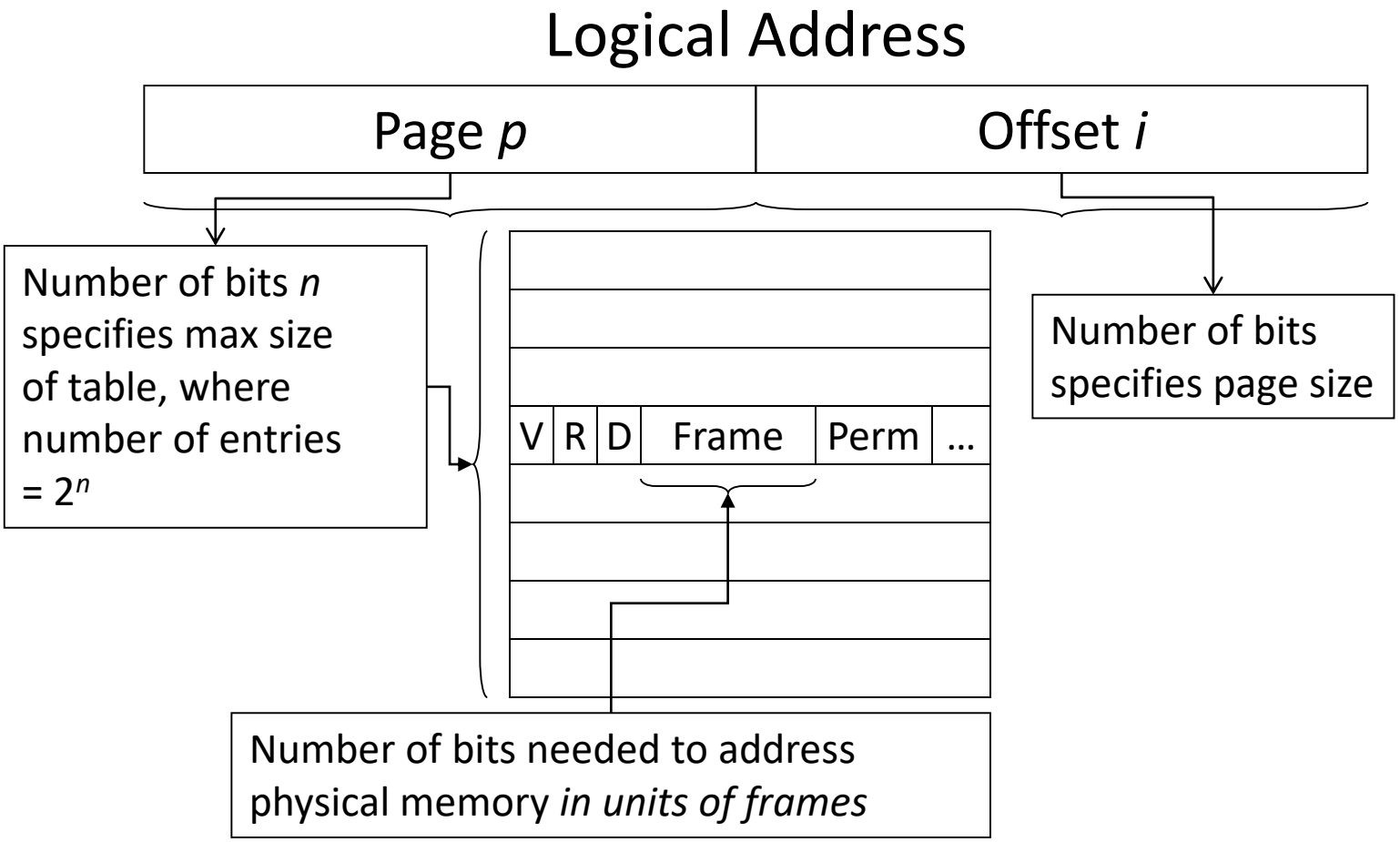
Translate Address



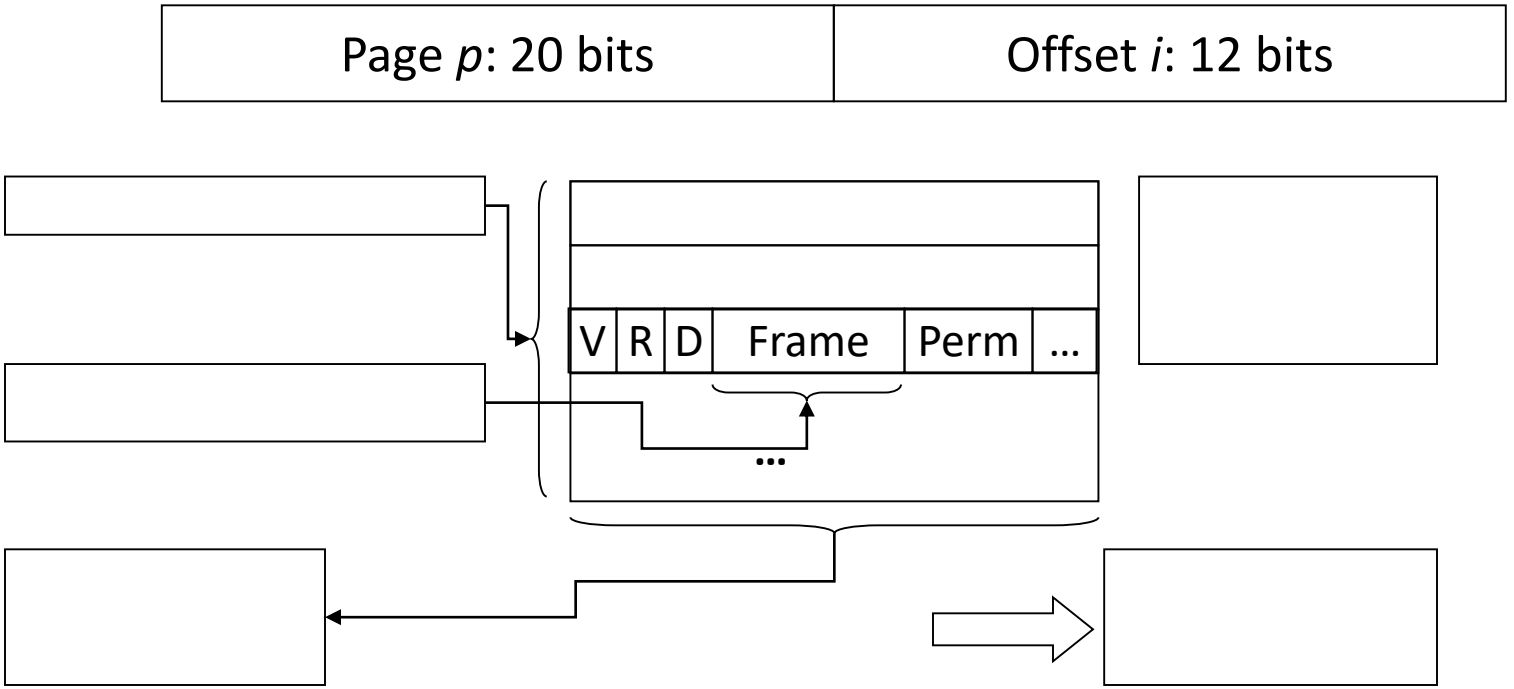
Physical Address by Concatenation



Sizing the Page Table

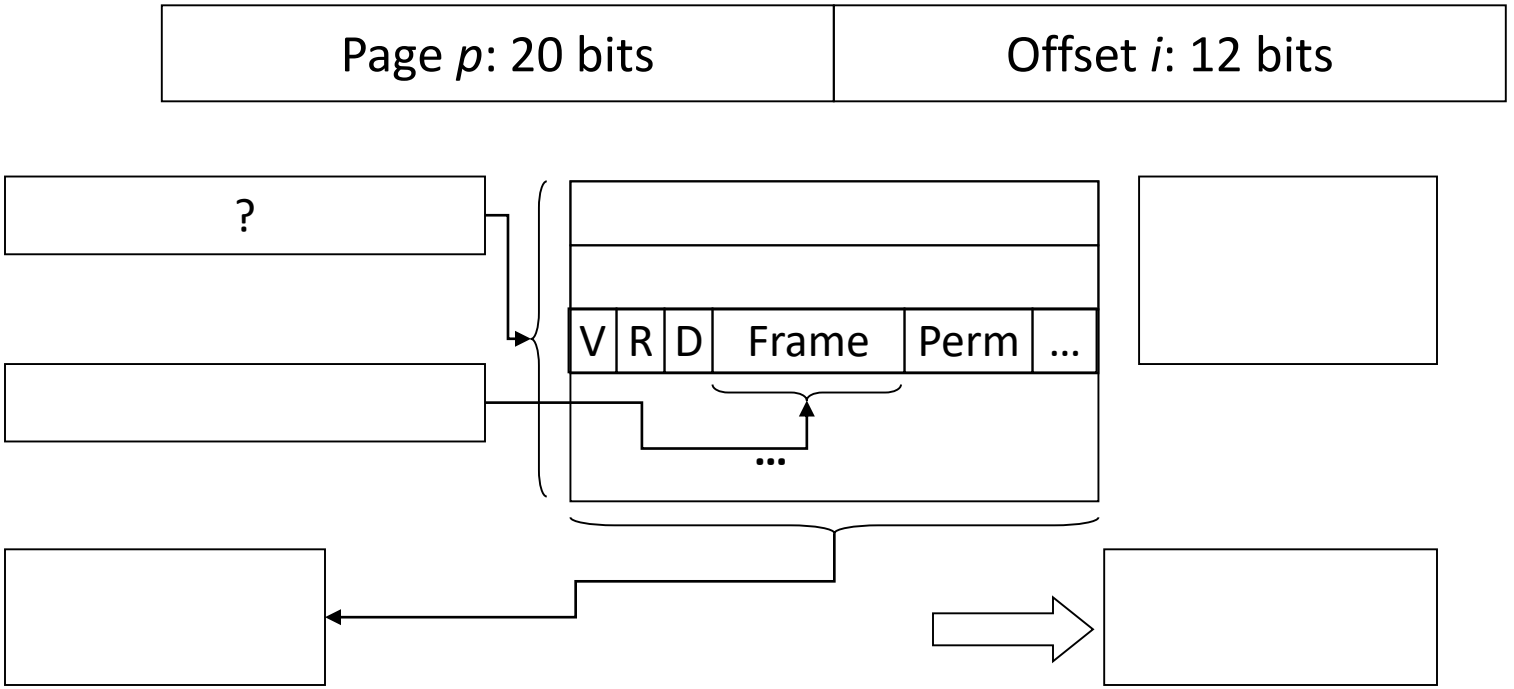


Example of Sizing the Page Table



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

Example of Sizing the Page Table

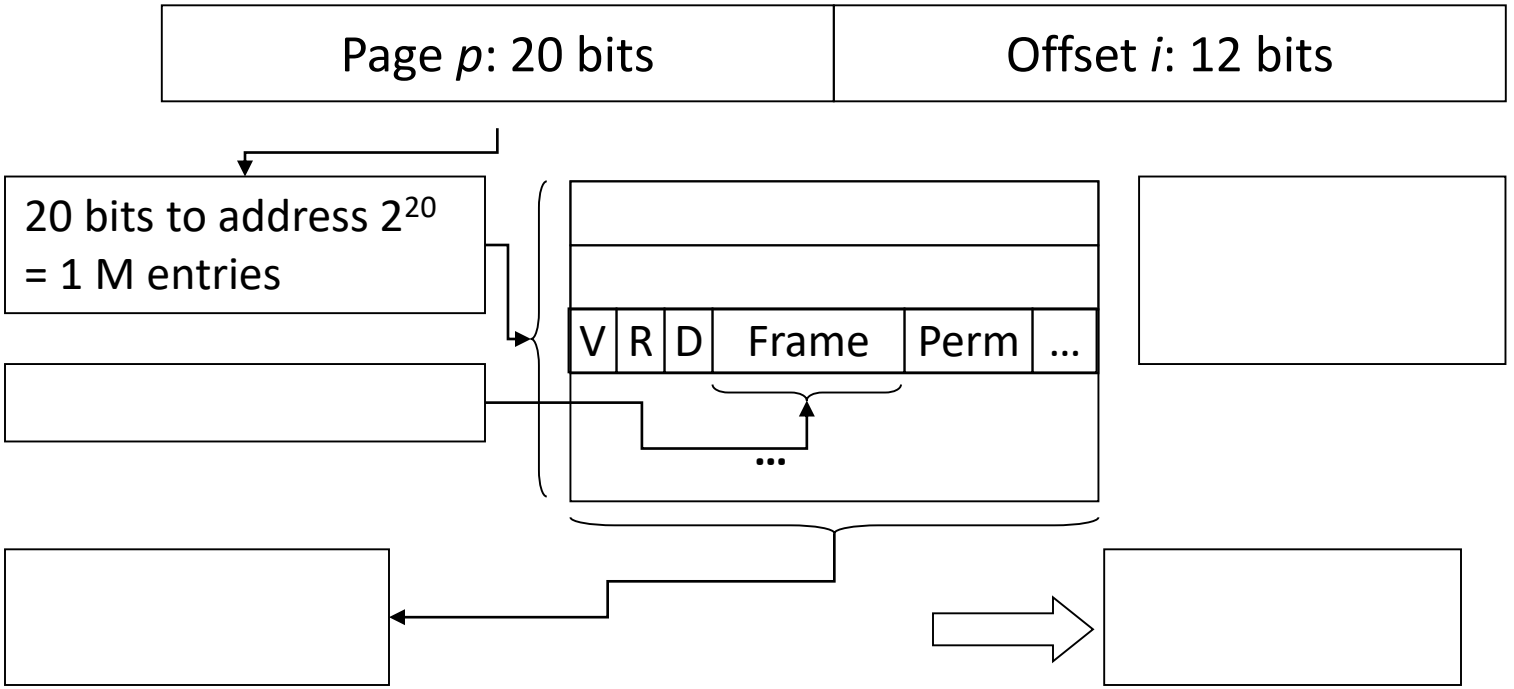


- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset

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

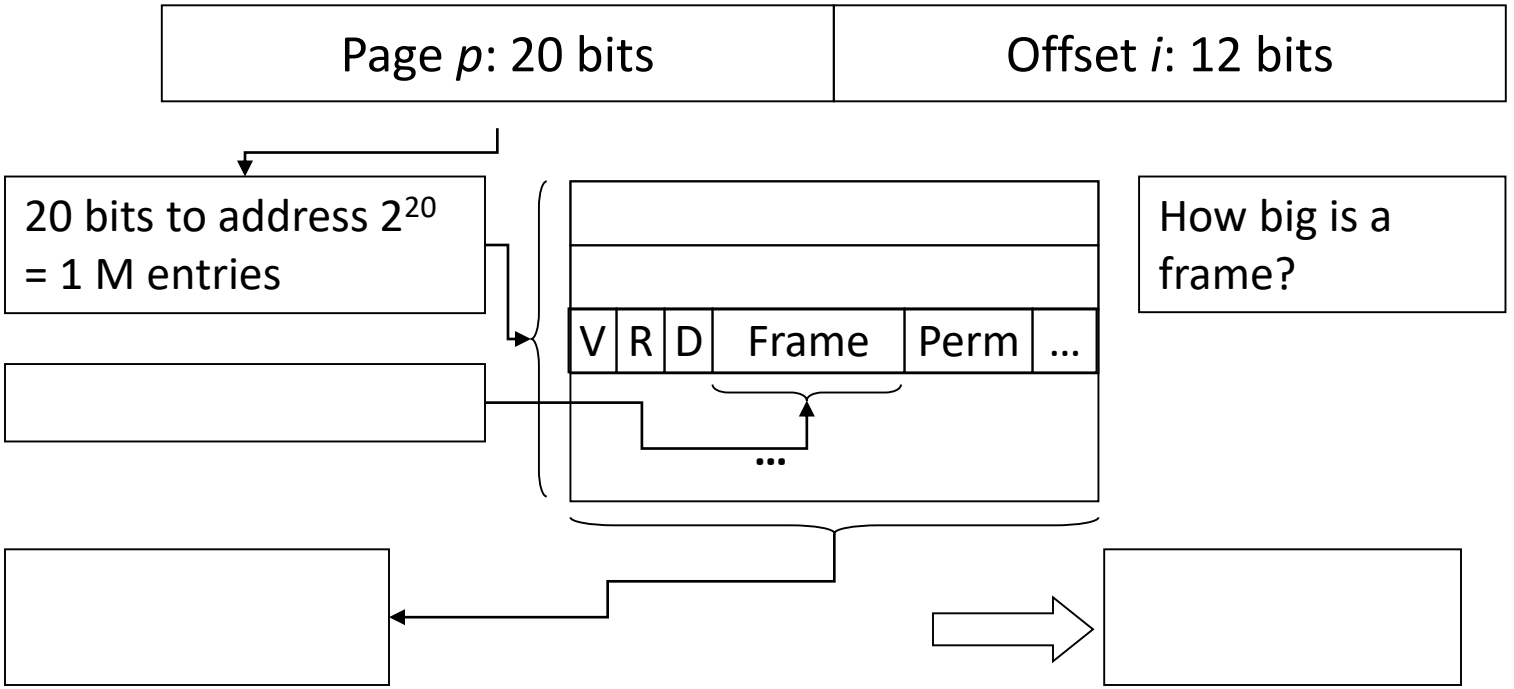
A. 2^{12}, because that's how many the offset field can address
B. 2^{20}, because that's how many the page field can address
C. 2^{30}, because that's how many we need to address 1 GB
D. 2^{32}, because that's the size of the entire address space

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

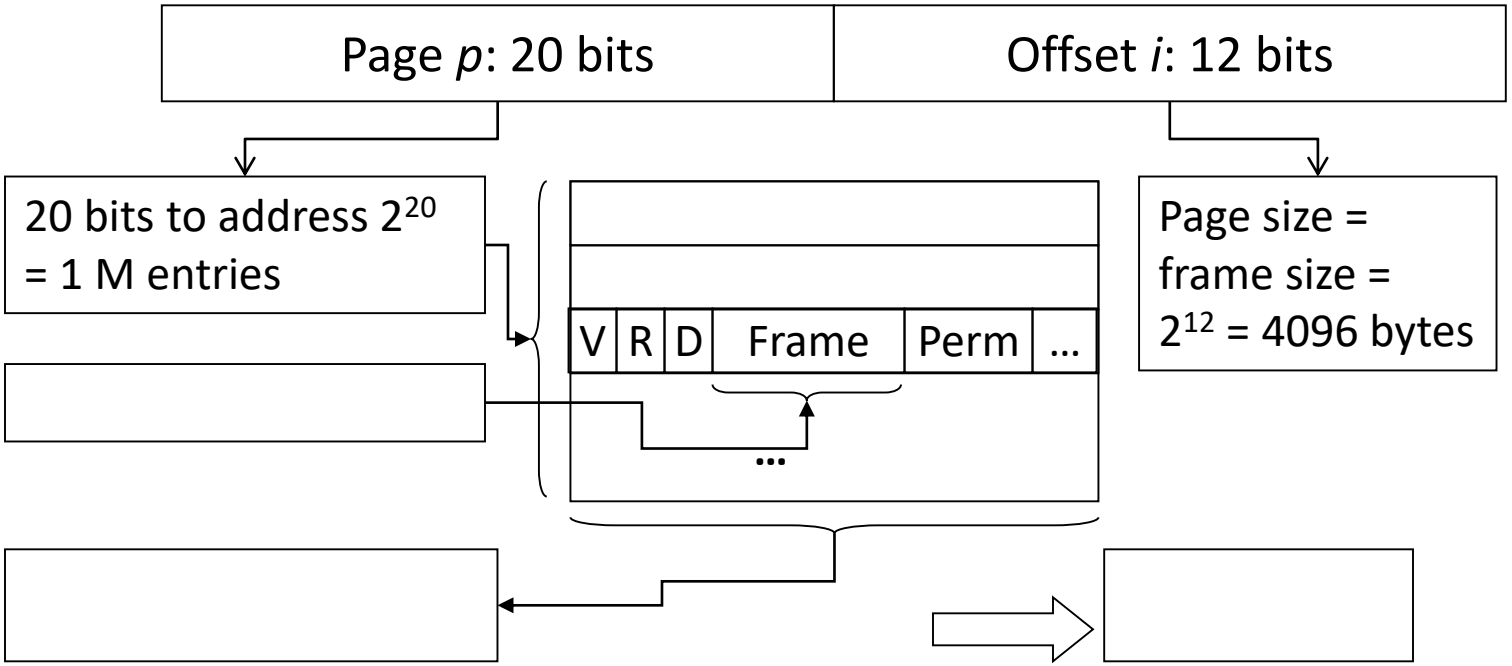


- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset

What will be the frame size, in bytes?

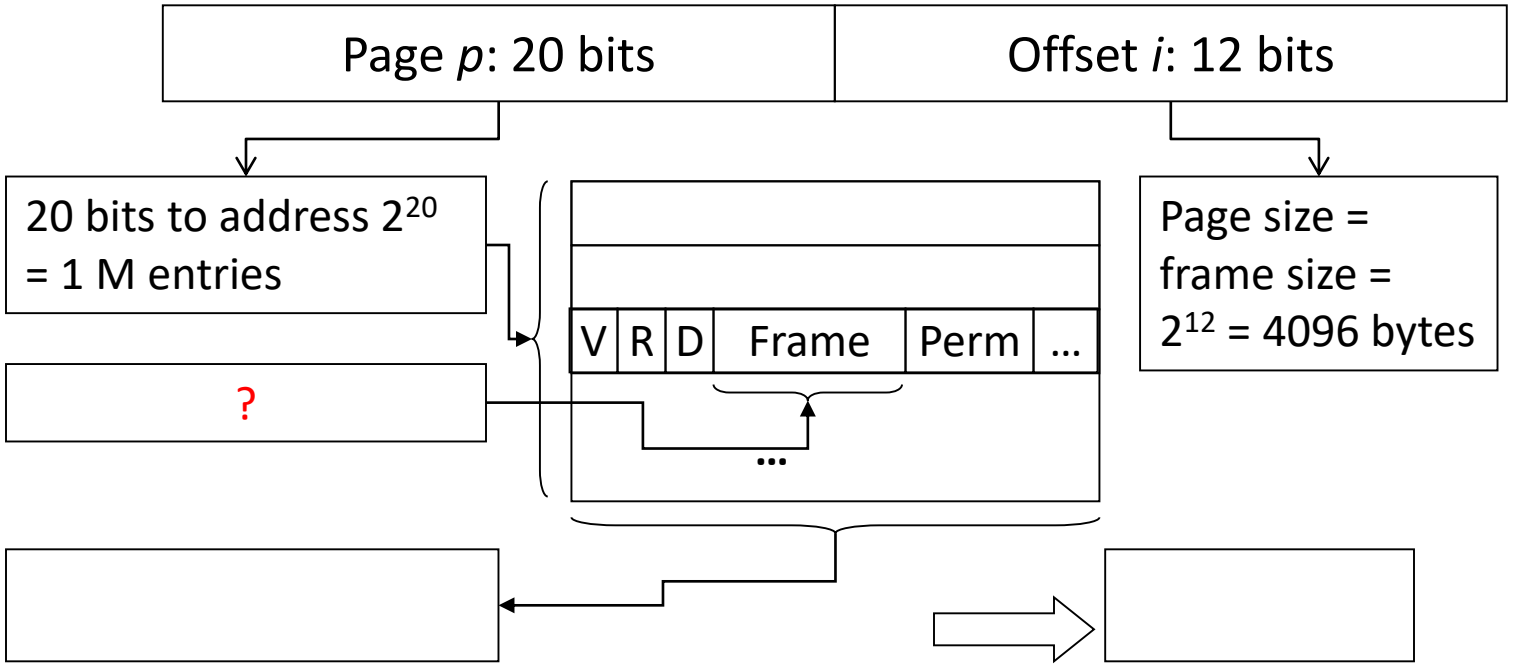
A. 2^{12}, because that's how many bytes the offset field can address
B. 2^{20}, because that's how many bytes the page field can address
C. 2^{30}, because that's how many bytes we need to address 1 GB
D. 2^{32}, because that's the size of the entire address space

Example of Sizing the Page Table



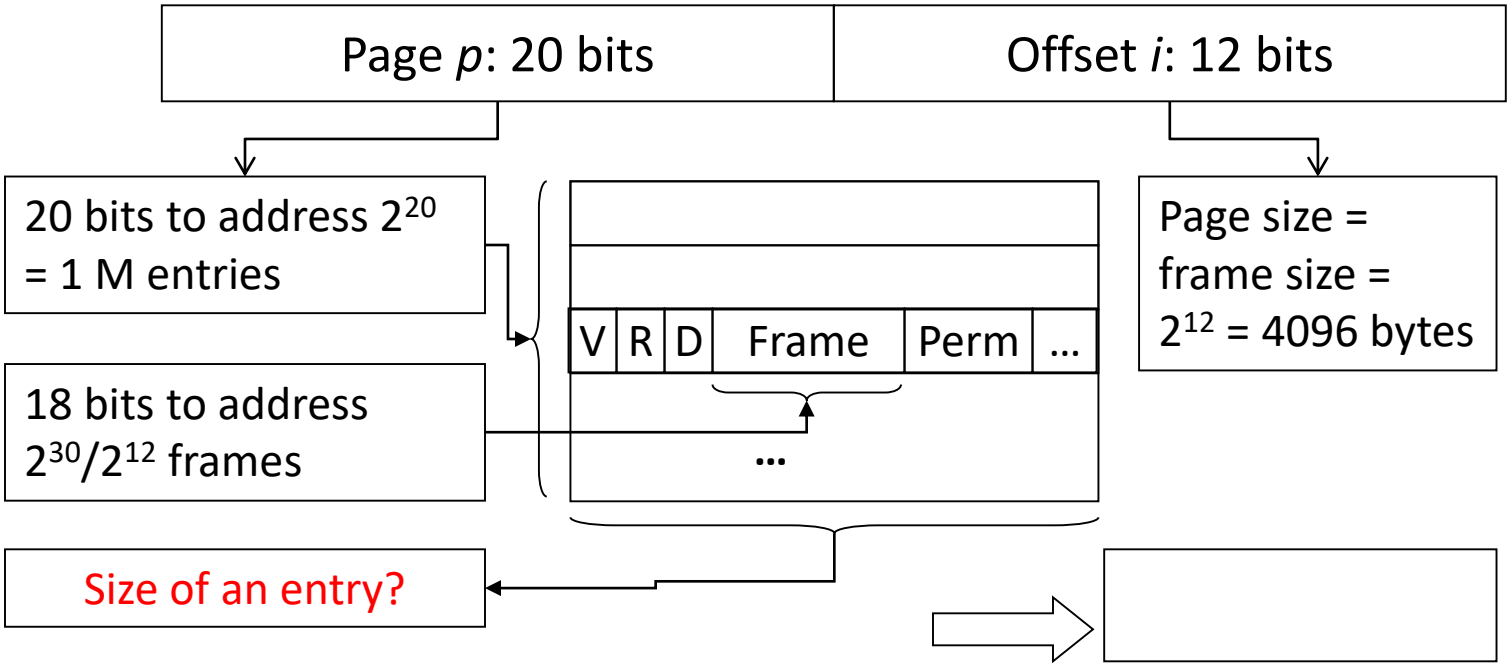
- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset

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



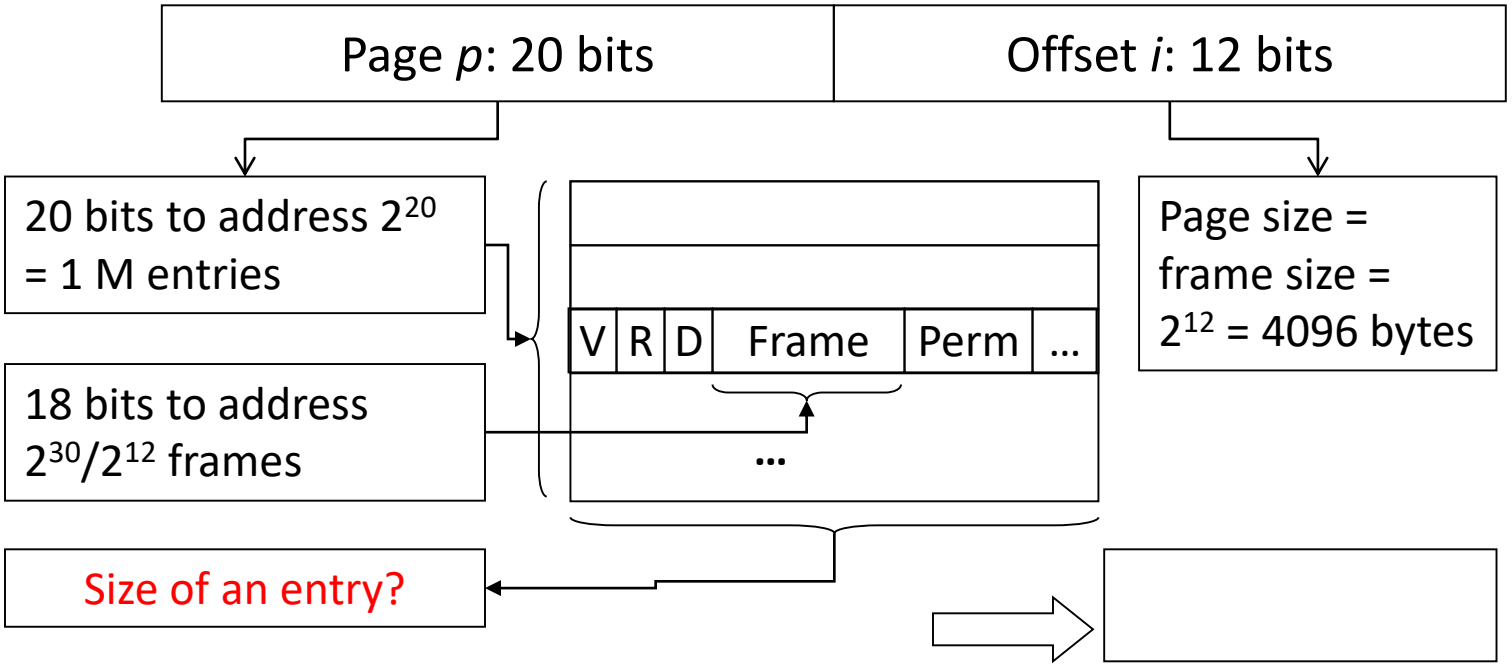
- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset
- A: 12 B: 18 C: 20 D: 30 E: 32

Example of Sizing the Page Table



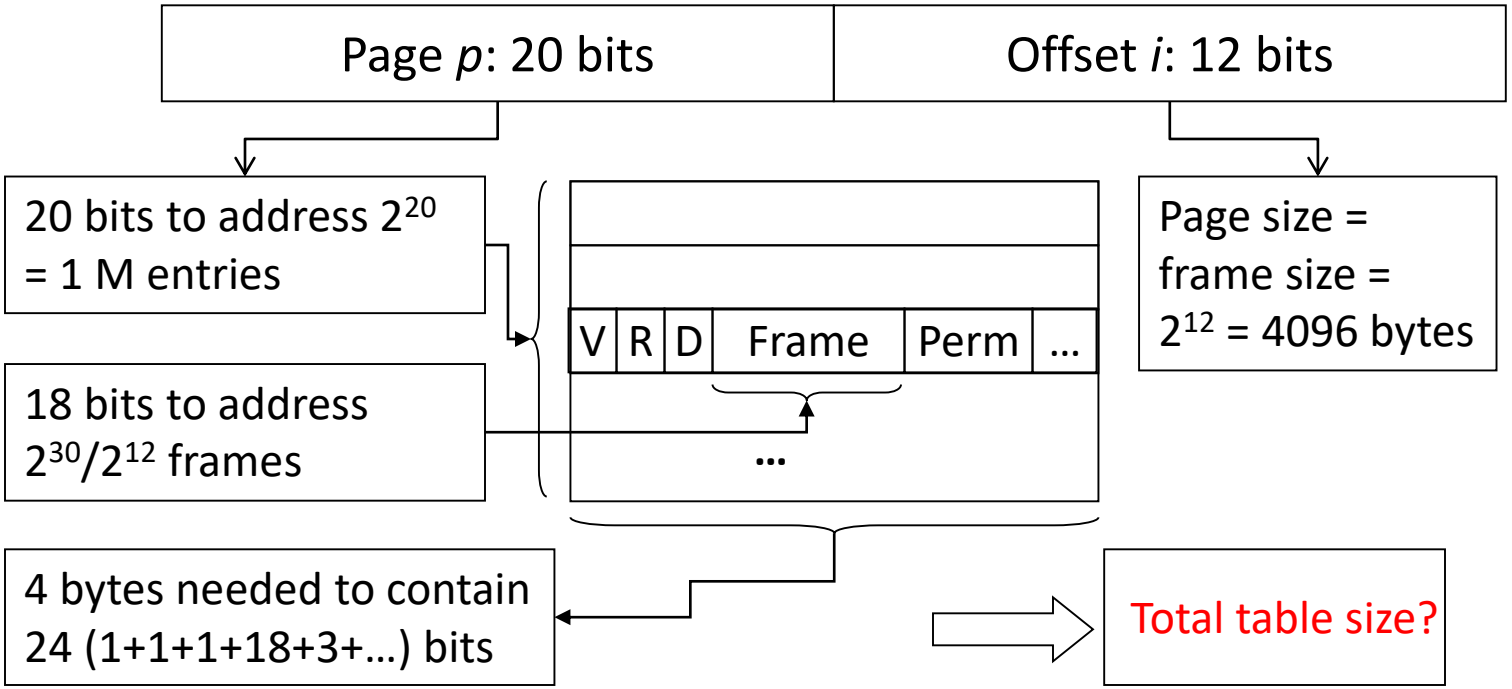
- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset

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



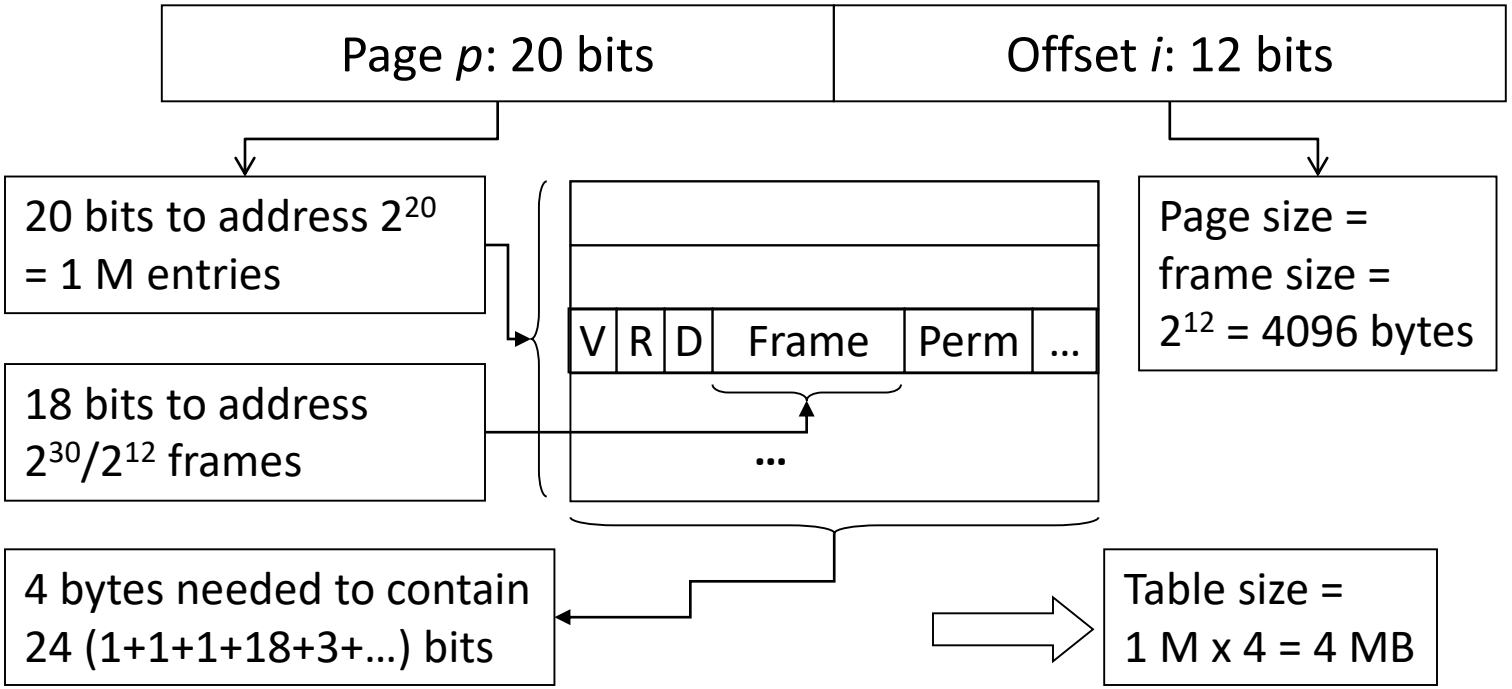
- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset
- A: 1 B: 2 C: 4 D: 8

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



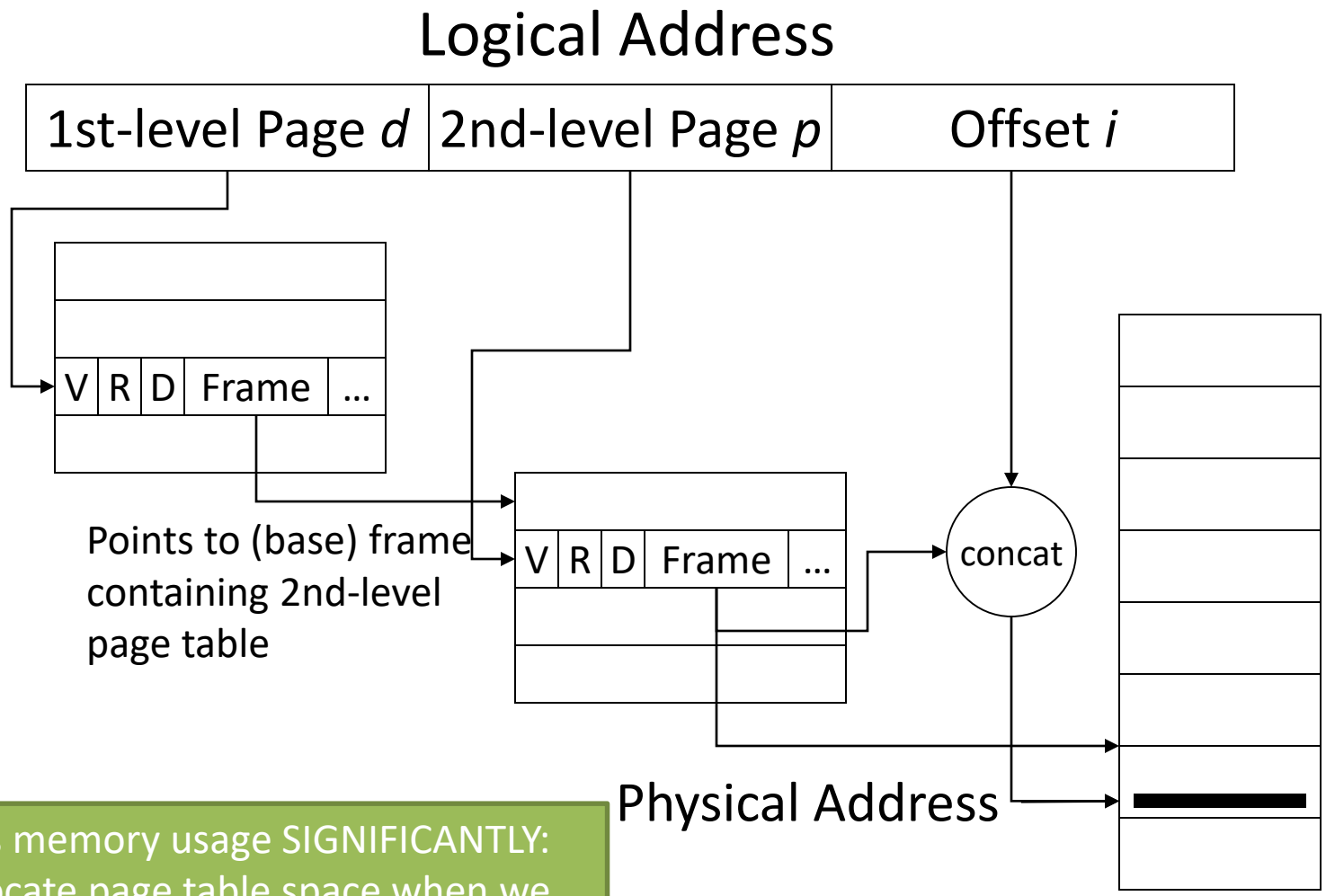
- 4 MB of bookkeeping for *every process*?
  - 200 processes -> 800 MB just to store page tables…

Concerns

- We’re going to need a ton of memory just for page tables…
- Wait, if 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.)



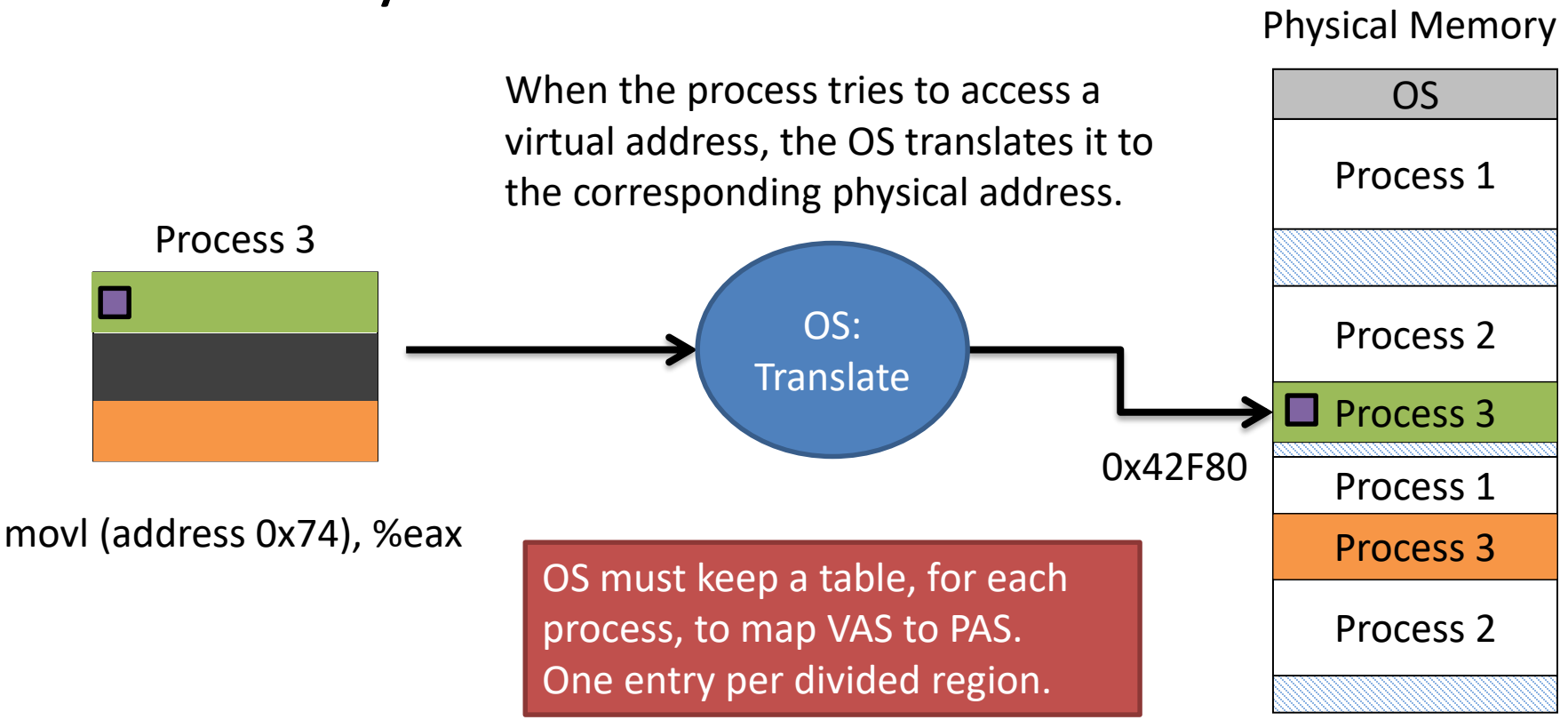
Reduces memory usage SIGNIFICANTLY: only allocate page table space when we need it. More memory accesses though…

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

Problem Summary: Addressing

- General solution: OS must translate process's VAS accesses to the corresponding physical memory location.



Problem: Storage

- Where should process memories be placed?
  - Topic: “Classic” memory management
- How does the compiler model memory?
  - Topic: Logical memory model
- How to deal with limited physical memory?
  - Topics: Virtual memory, paging

Recall “Storage Problem”

- We must keep multiple processes in memory, but how many?
  - Lots of processes: they must be small
  - Big processes: can only fit a few

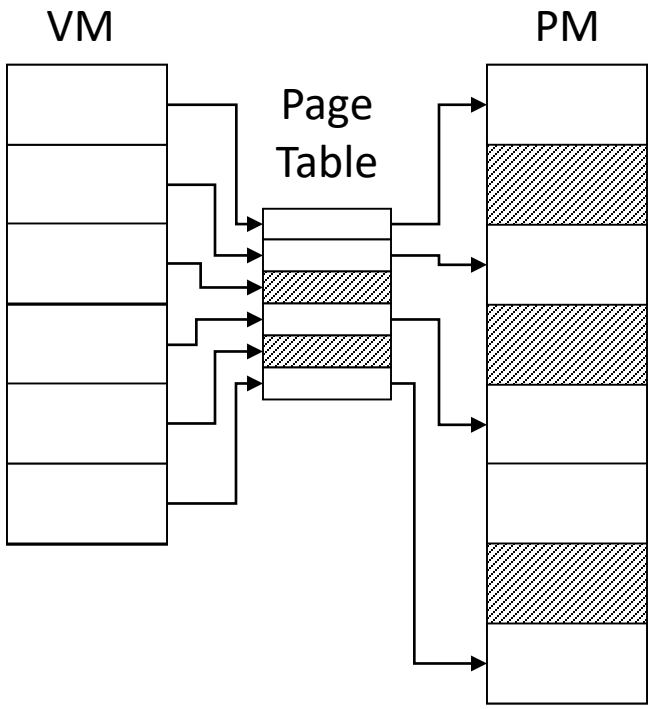
- How do we balance this tradeoff?

Locality to the rescue!

VM Implications

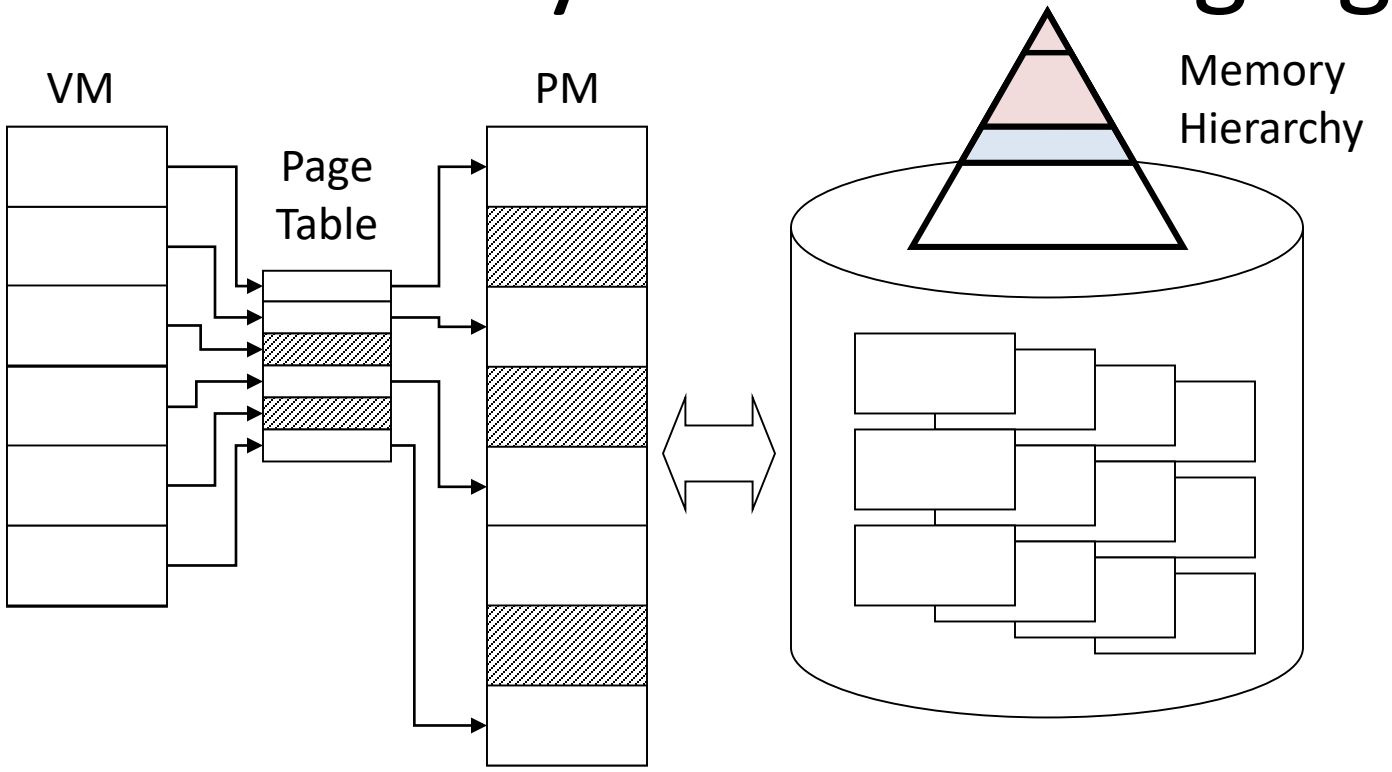
- Not all pieces need to be in memory
  - Need only piece being referenced
  - Other pieces can be on disk
  - Bring pieces in only when needed
- Illusion: there is much more memory
- What's needed to support this idea?
  - A way to identify whether a piece is in memory
  - A way to bring in pieces (from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

Virtual Memory based on Paging



- Before
  - All virtual pages were in physical memory

Virtual Memory based on Paging



- Now
  - All virtual pages reside on disk
  - Some also reside in physical memory (which ones?)
- Ever been asked about a swap partition on Linux?

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

A page fault occurs. What must we do in response?

A. Find the faulting page on disk.

B. Evict a page from memory and write it to disk.

C. Bring in the faulting page and retry the operation.

D. Two of the above

E. All of the above

Address Translation and Page Faults

- Get entry: index page table with page number
- If valid bit is off, page fault
  - Trap into operating system
  - Find page on disk (kept in kernel data structure)
  - Read it into a free frame
    - may need to make room: page replacement
  - Record frame number in page table entry, set valid
  - Retry instruction (return from page-fault trap)

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 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 location (e.g., in page table).
  - One page table for every process
  - TLB hardware helps to speed up translation
- Provides the abstraction of very large memory: not all pages need be resident in memory
  - Bring pages in from disk on demand