CS 31: Intro to Systems C Programming

L20-21: Virtual Memory

Vasanta Chaganti & Kevin Webb

Swarthmore College

Nov 21, 28, 2023

Announcements

- HW 6 is out!
- Pre-registration is Tuesday – Thursday 30th November
  - Must pre-register to get into a CS course
- Senior Poster Sessions! Support your seniors!
  - Tuesday – Thursday: 7 – 9 PM
  - CS Hallway
  - Food and Snacks!

Reading Quiz

OS Big Picture Goals

- OS is a layer of code between user programs and hardware.
- Goal: Make life easier for users and programmers.
- How can the OS do that?

Key OS Responsibilities

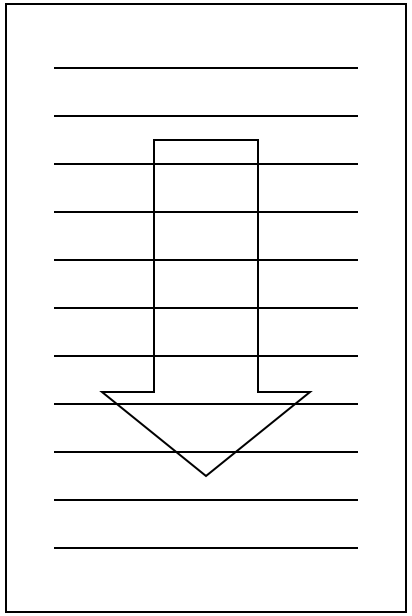
1. Simplifying abstractions for programs

2. Resource allocation and/or sharing

3. Hardware gatekeeping and protection

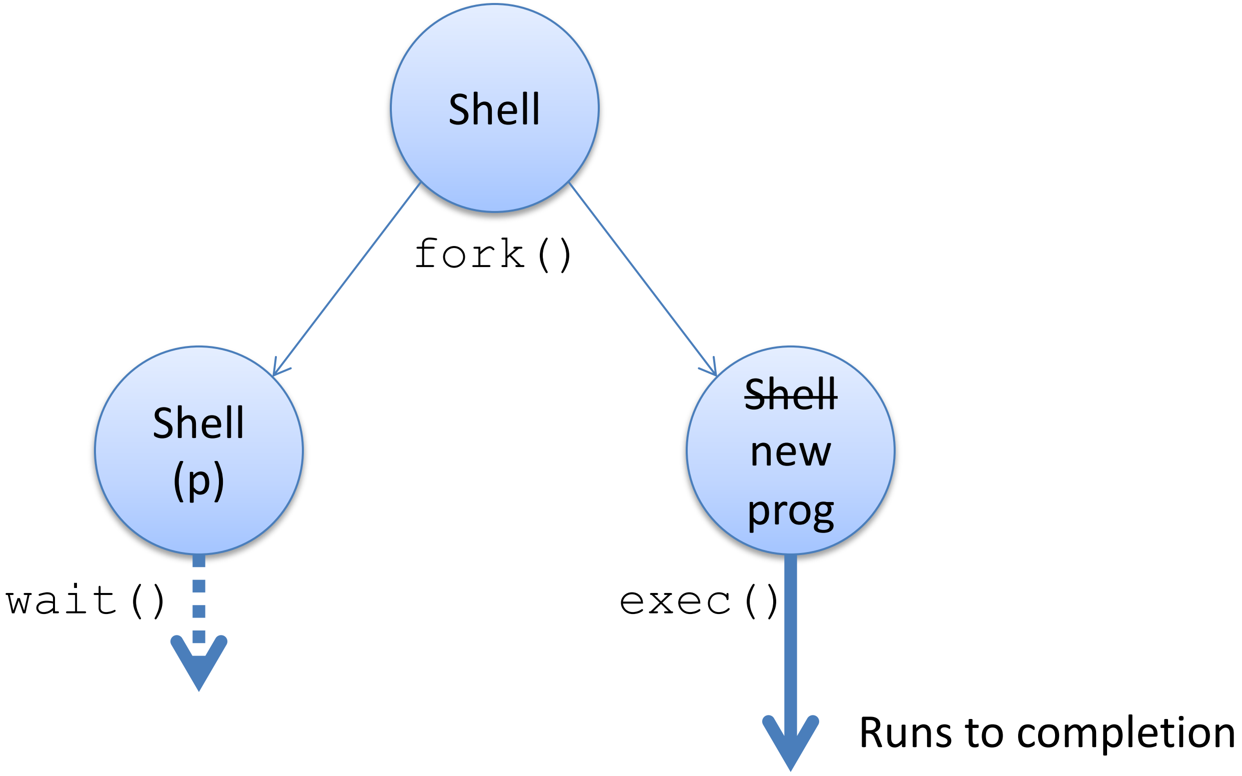
Anatomy of a Process

- Abstraction of a running program
  - a dynamic “program in execution”
- OS keeps track of process state
  - What each process is doing
  - Which one gets to run next
- Basic operations
  - Suspend/resume (context switch)
  - Start (spawn), terminate (kill)



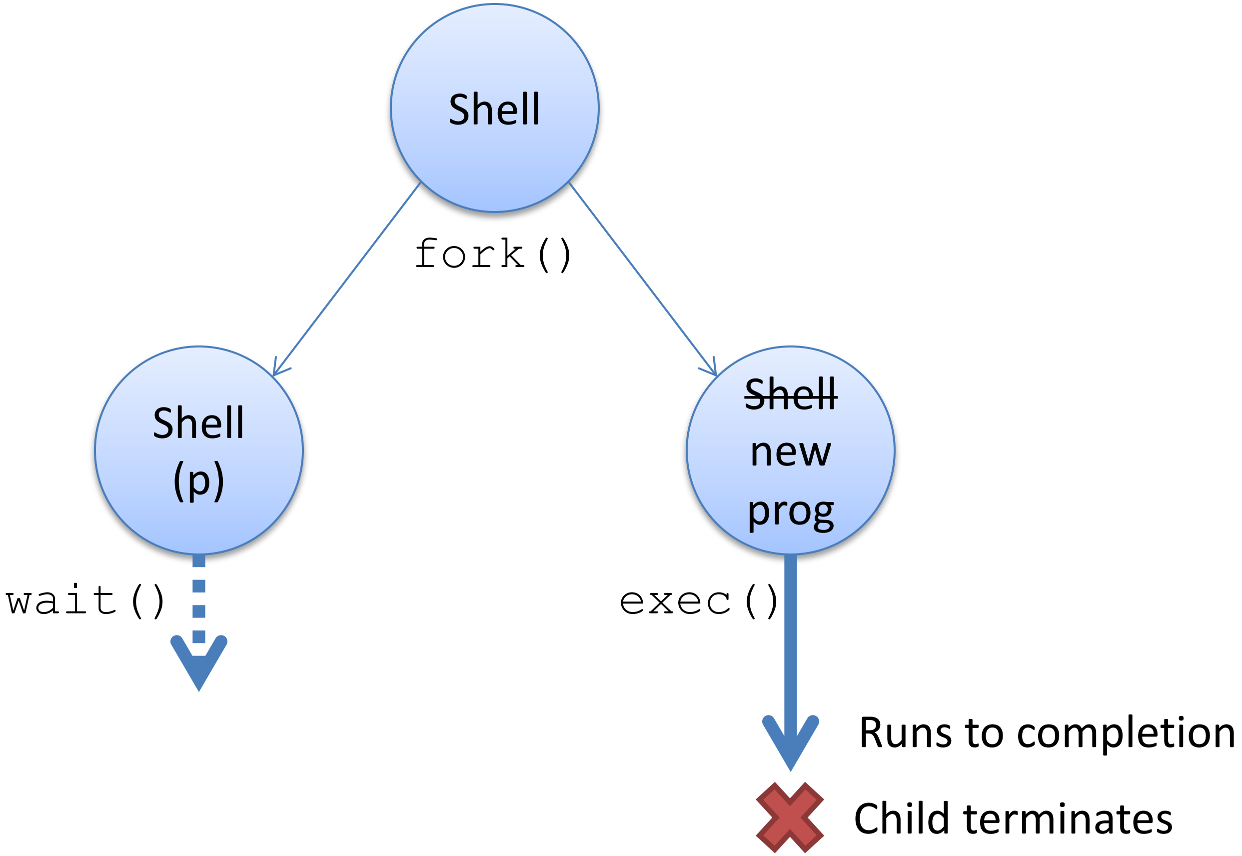
Common `fork()` usage: Shell

2. child: `exec()` user-requested program



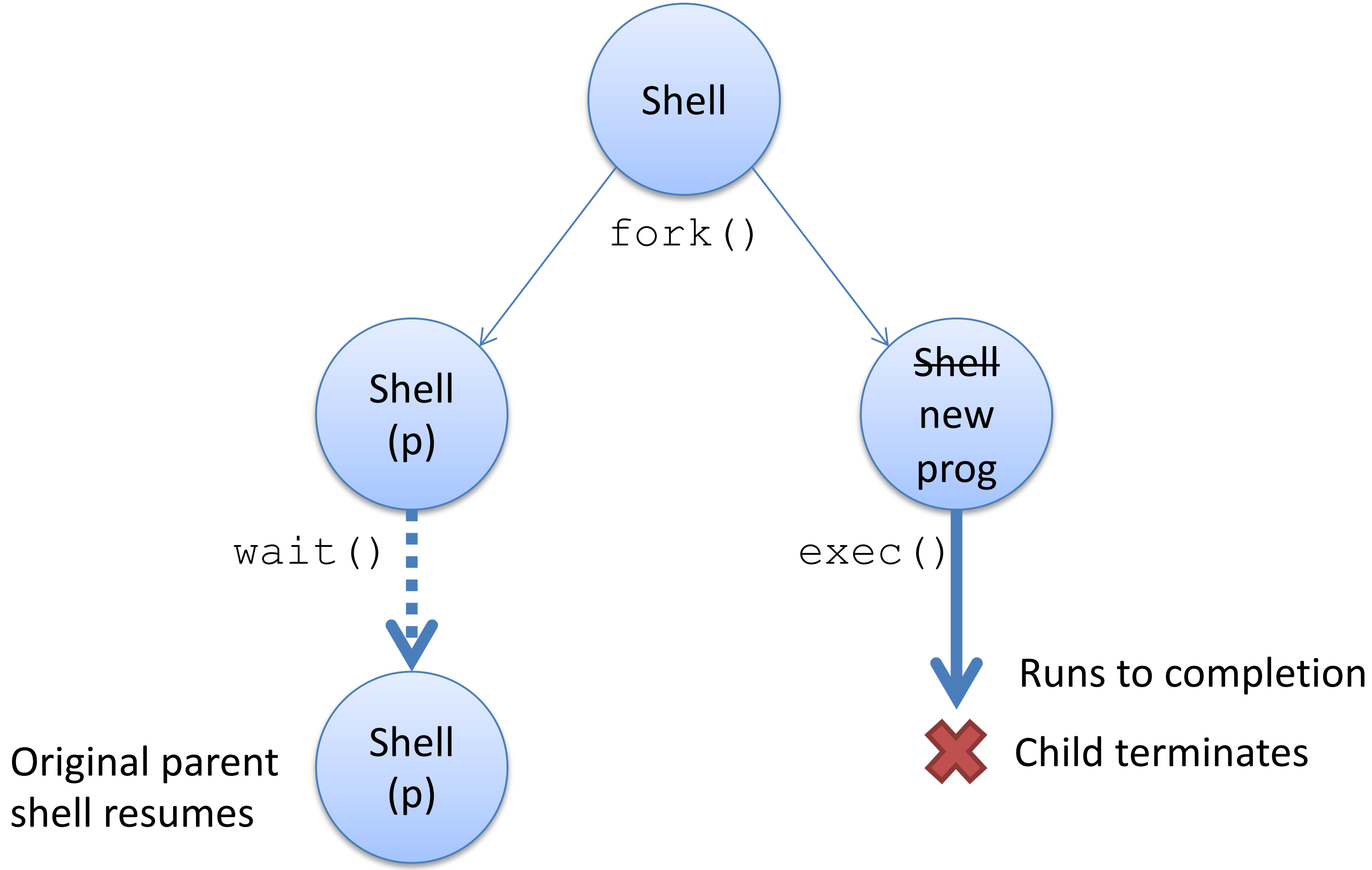
Common `fork()` usage: Shell

3. child program terminates, cycle repeats



Common `fork()` usage: Shell

3. child program terminates, cycle repeats



Process Management: Summary

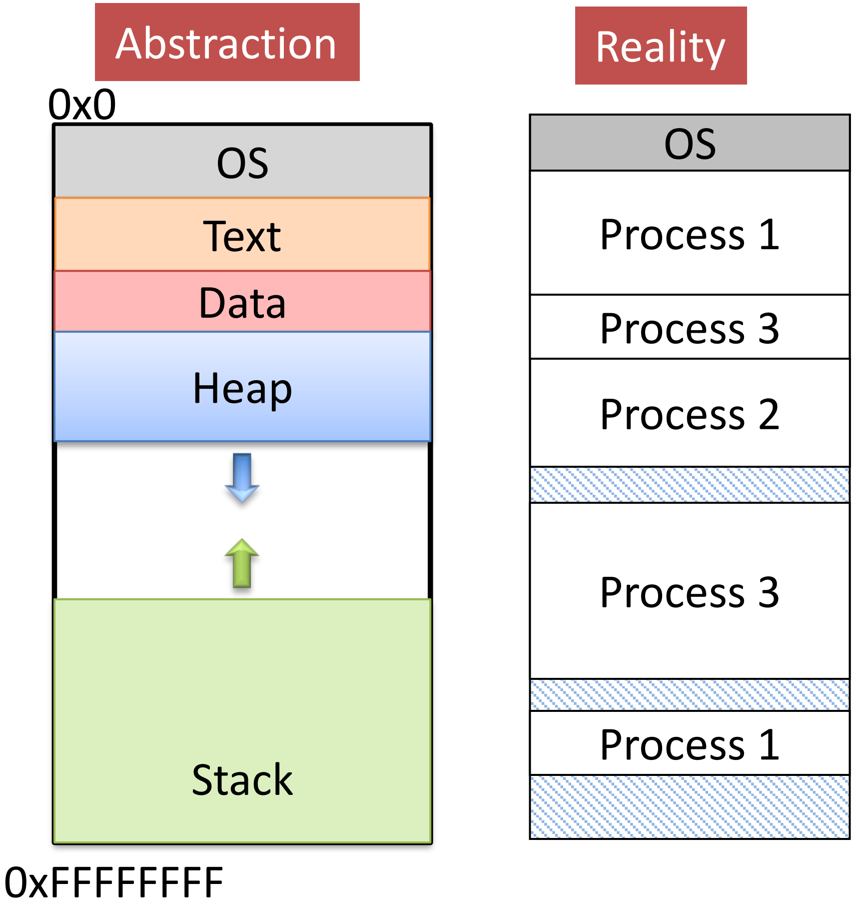
- A process is the unit of execution.
- Processes are represented as Process Control Blocks in the OS
  - PCBs contain process state, scheduling and memory management information, etc
- A process is either New, Ready, Waiting, Running, or Terminated.
- On a uniprocessor, there is at most one running process at a time.
- The program currently executing on the CPU is changed by performing a context switch
- Processes communicate either with message passing or shared memory

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.

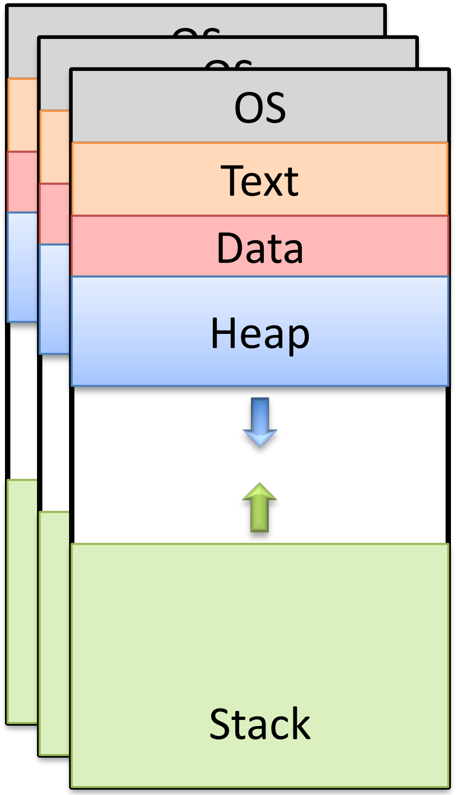
OS (with help from hardware) will keep track of who's using each memory region.



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!

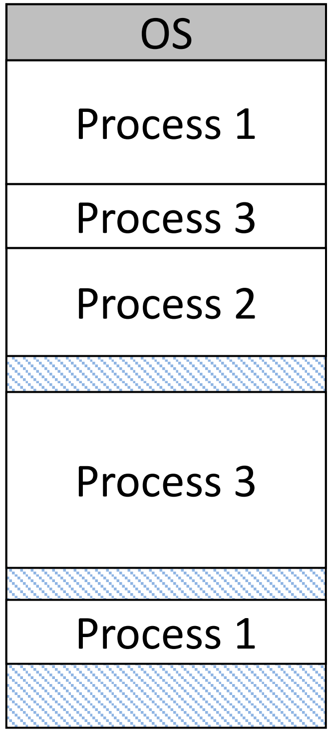


0x0

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

The set of available storage locations.

0x0



Virtual address space (VAS): fixed size.

0xFFFFFFFF

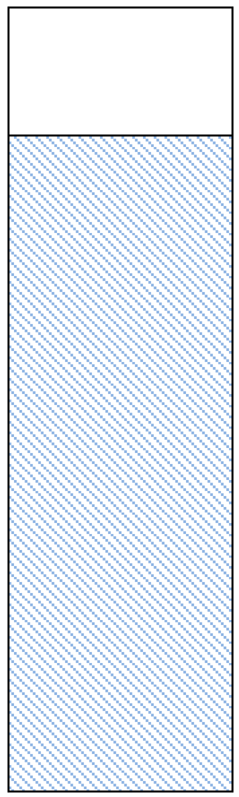
0x…
(Determined by amount of installed RAM.)

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

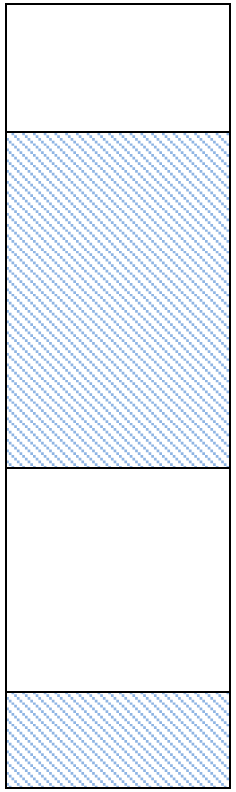
- Physical memory starts as one big empty space.



Memory Management

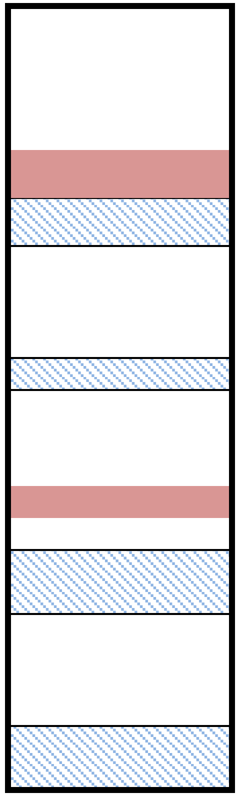
- Physical memory starts as one big empty space.

- Processes need to be in memory to execute.



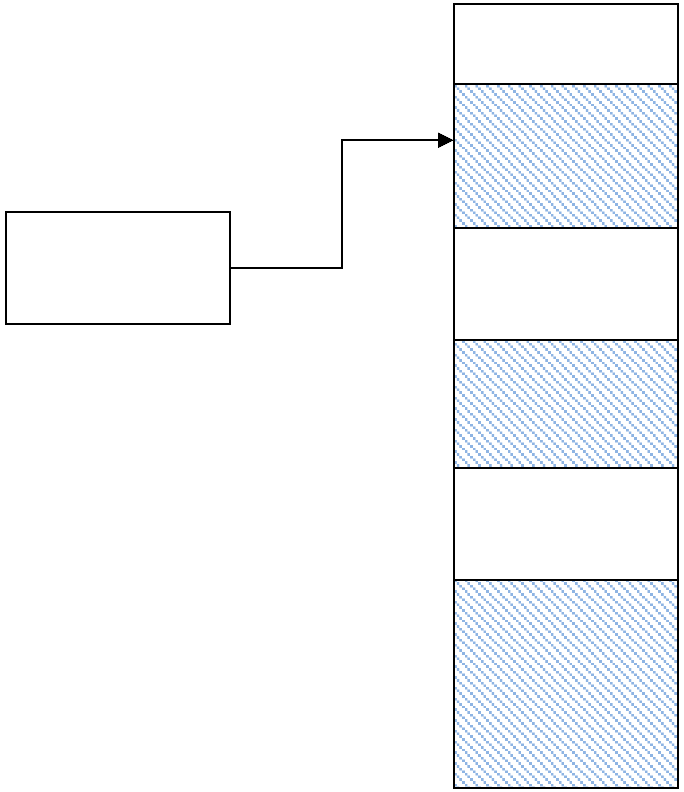
Fragmentation

- 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?)



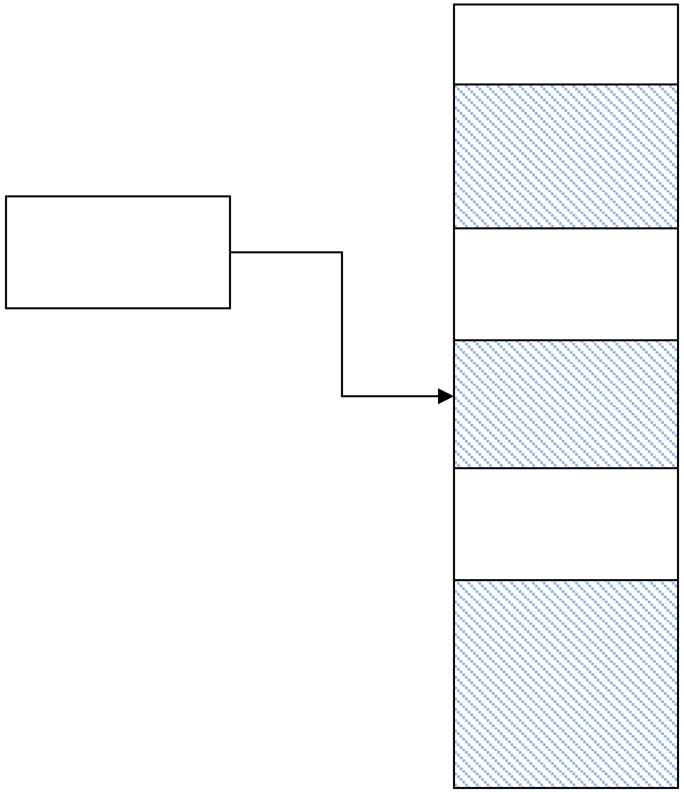
Placing Memory

- When searching for space, what if there are multiple options?
- Algorithms
  - *First (or next) fit*
  - Best fit
  - Worst fit



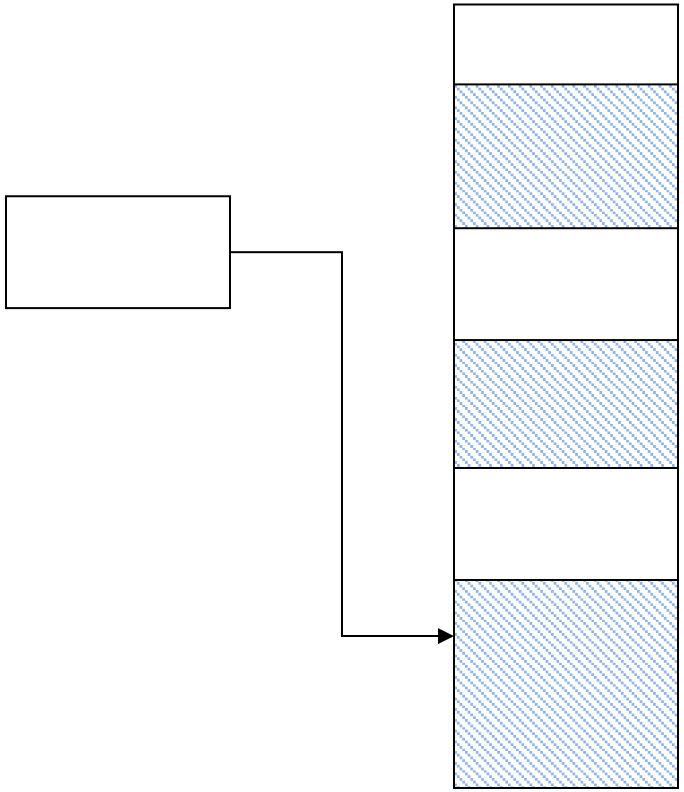
Placing Memory

- When searching for space, what if there are multiple options?
- Algorithms
  - First (or next) fit
  - *Best fit*
  - Worst fit



Placing Memory

- When searching for space, what if there are multiple options?
- Algorithms
  - First (or next) fit
  - Best fit
  - *Worst fit*



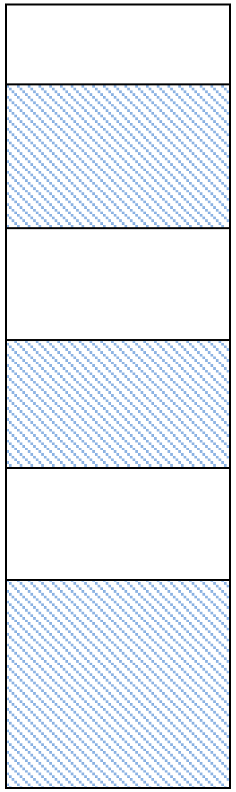
Which memory allocation algorithm would you choose? Why?

A. first-fit

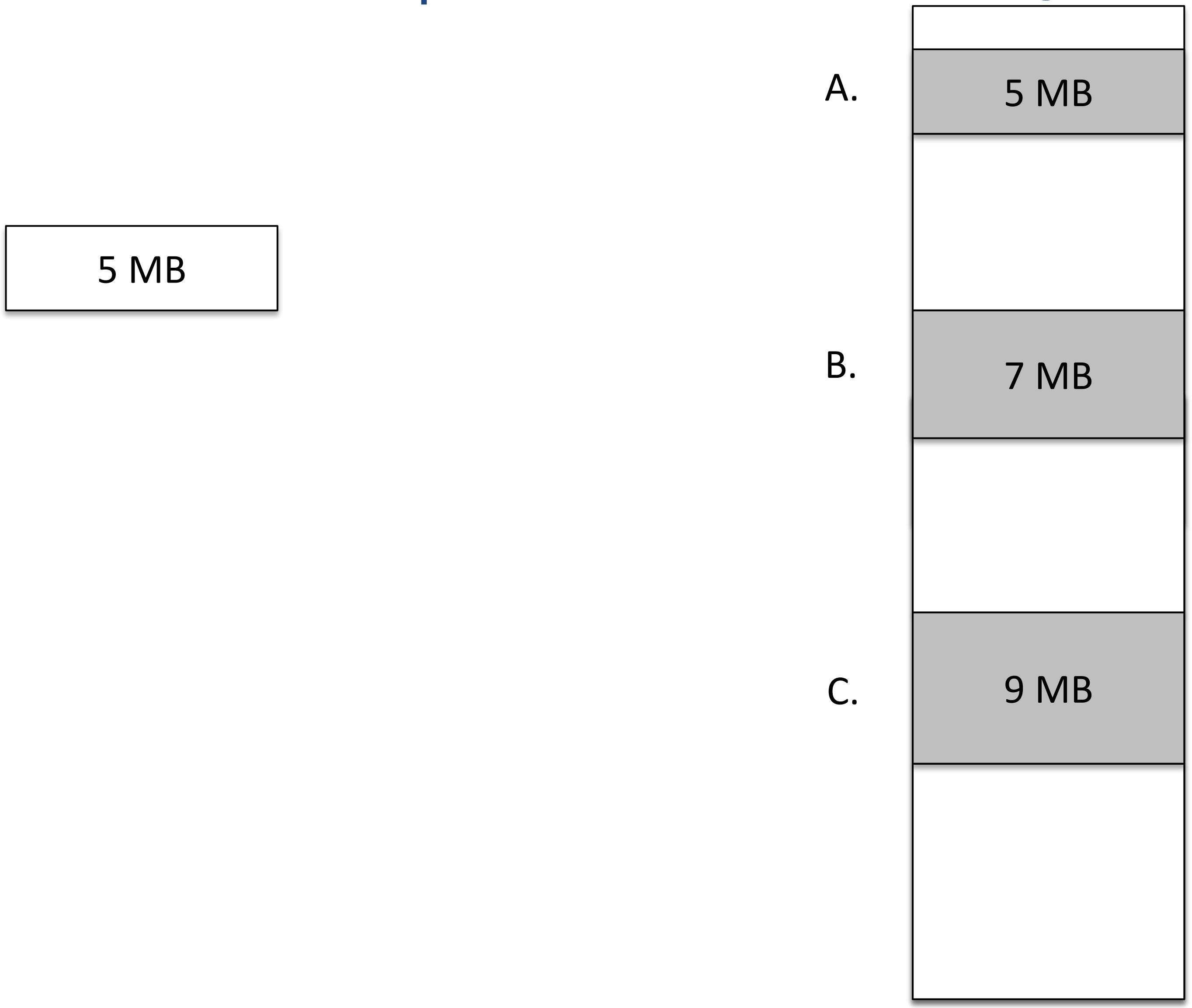
B. worst-fit

C. best-fit

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

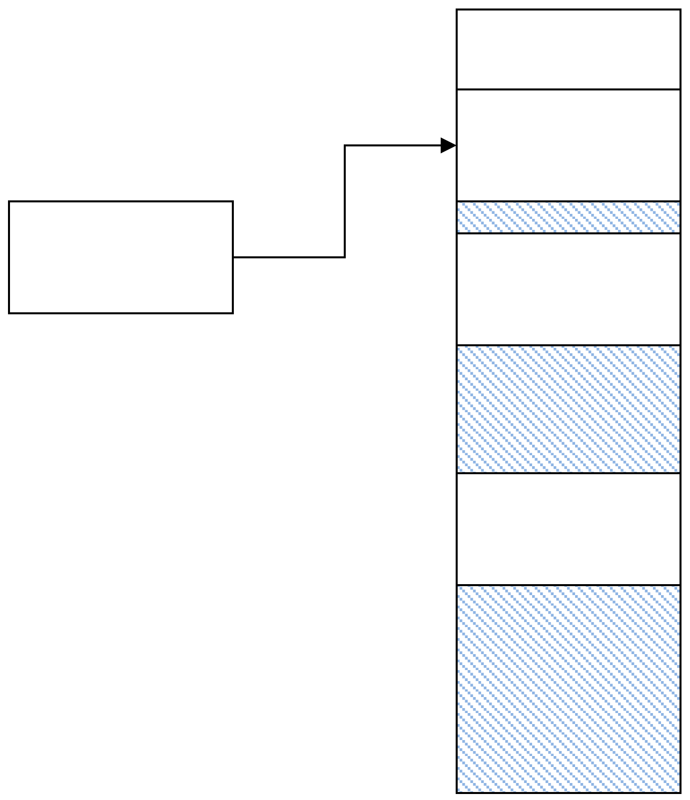


Where would worst-fit place this memory chunk?



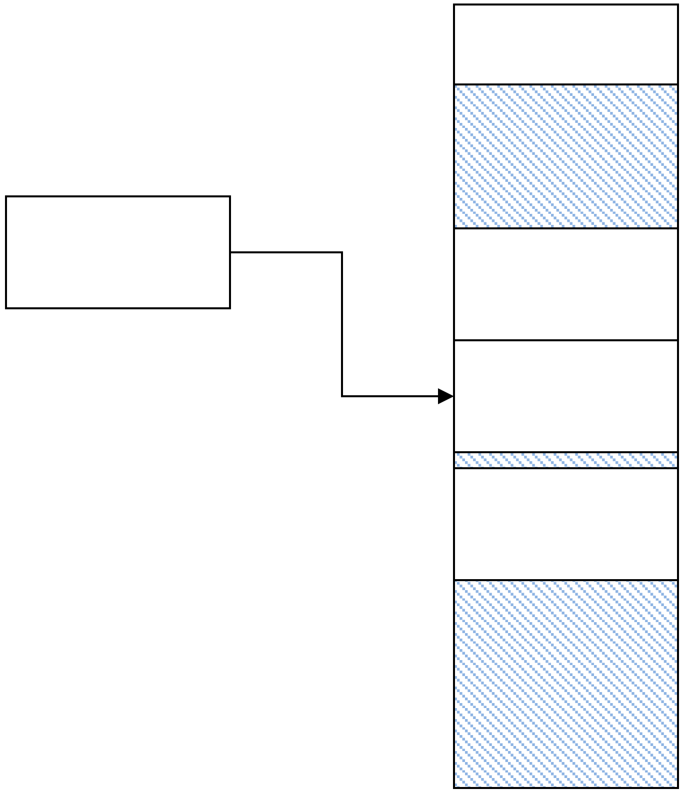
Placing Memory

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



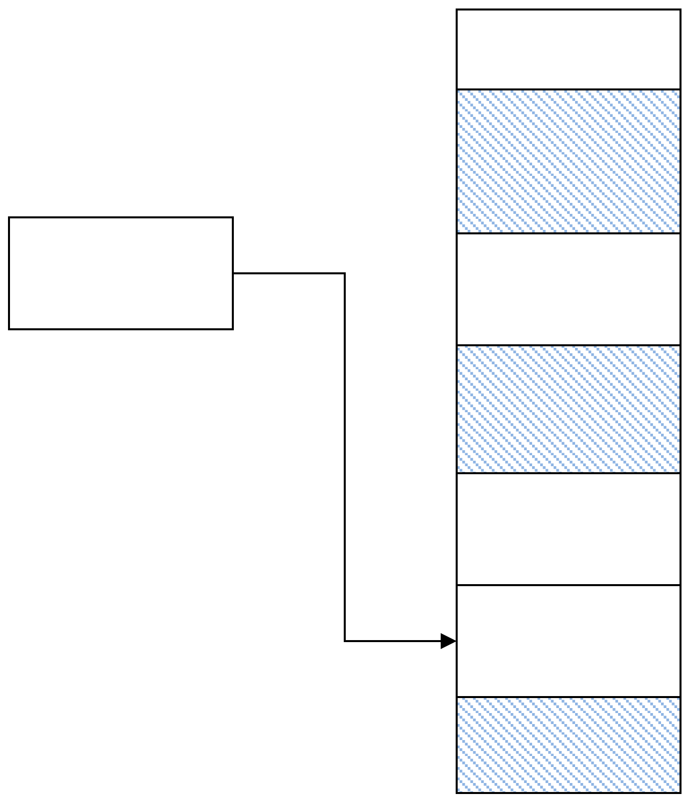
Placing Memory

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



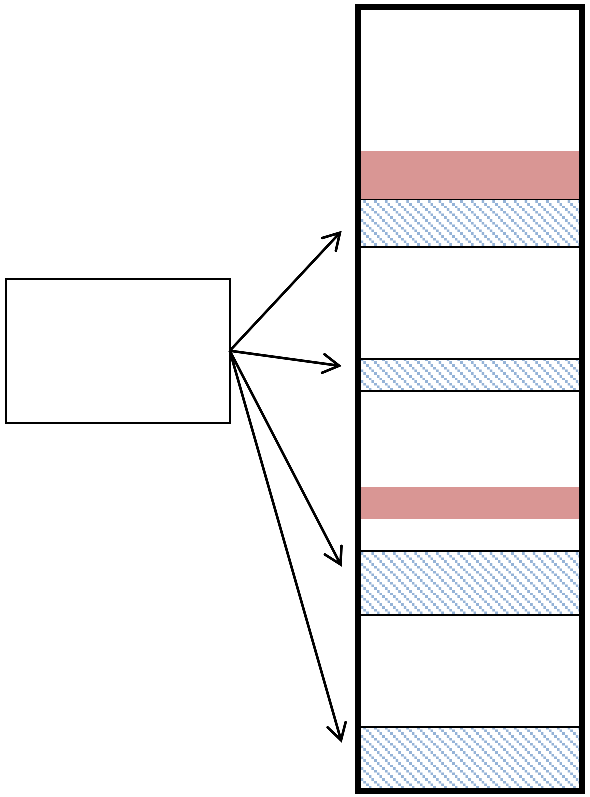
Placing Memory

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



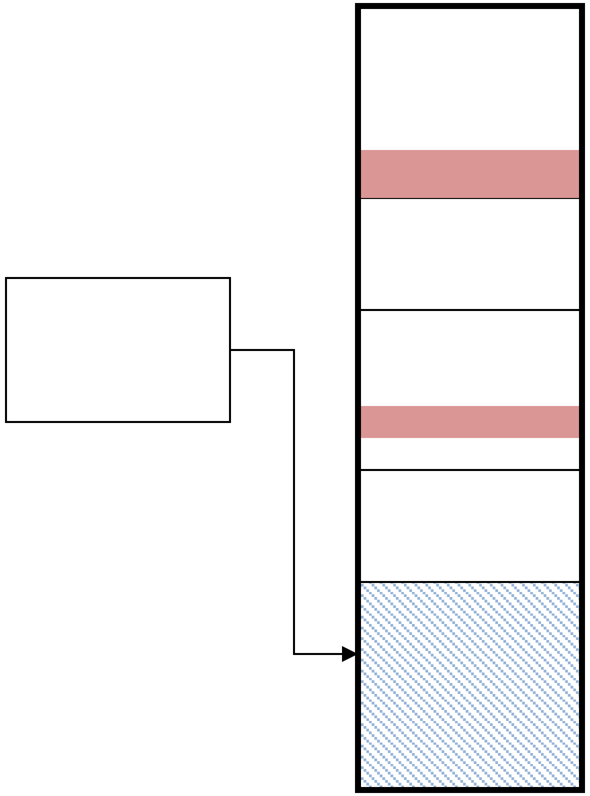
What if it doesn't fit?

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



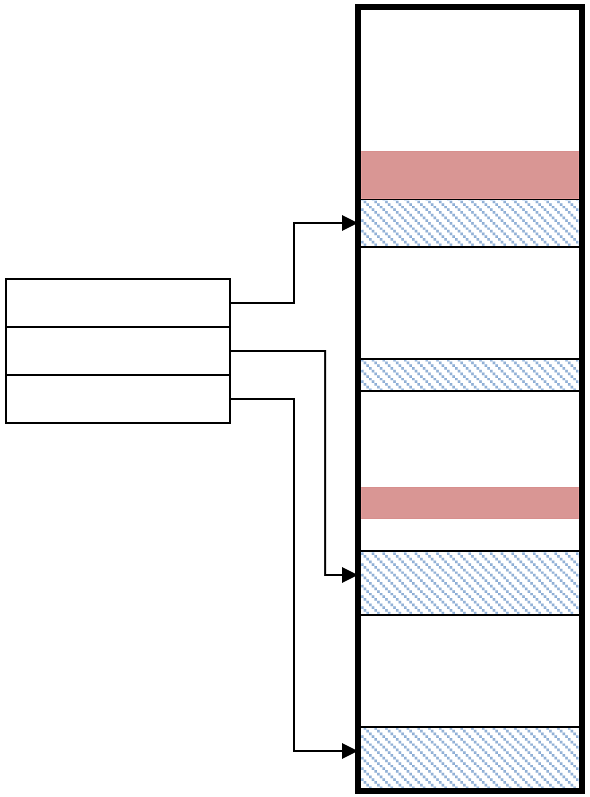
What if it doesn't fit?

- 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.



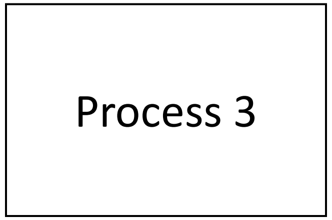
Problem Summary: Placement

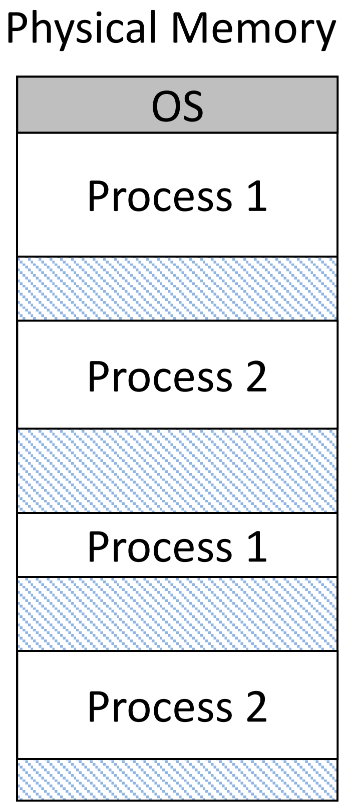
- difficult to find a large free region in physical memory for a process.
- fragmentation makes this harder over time
  - free pieces get smaller, spread out

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

Problem Summary: Placement

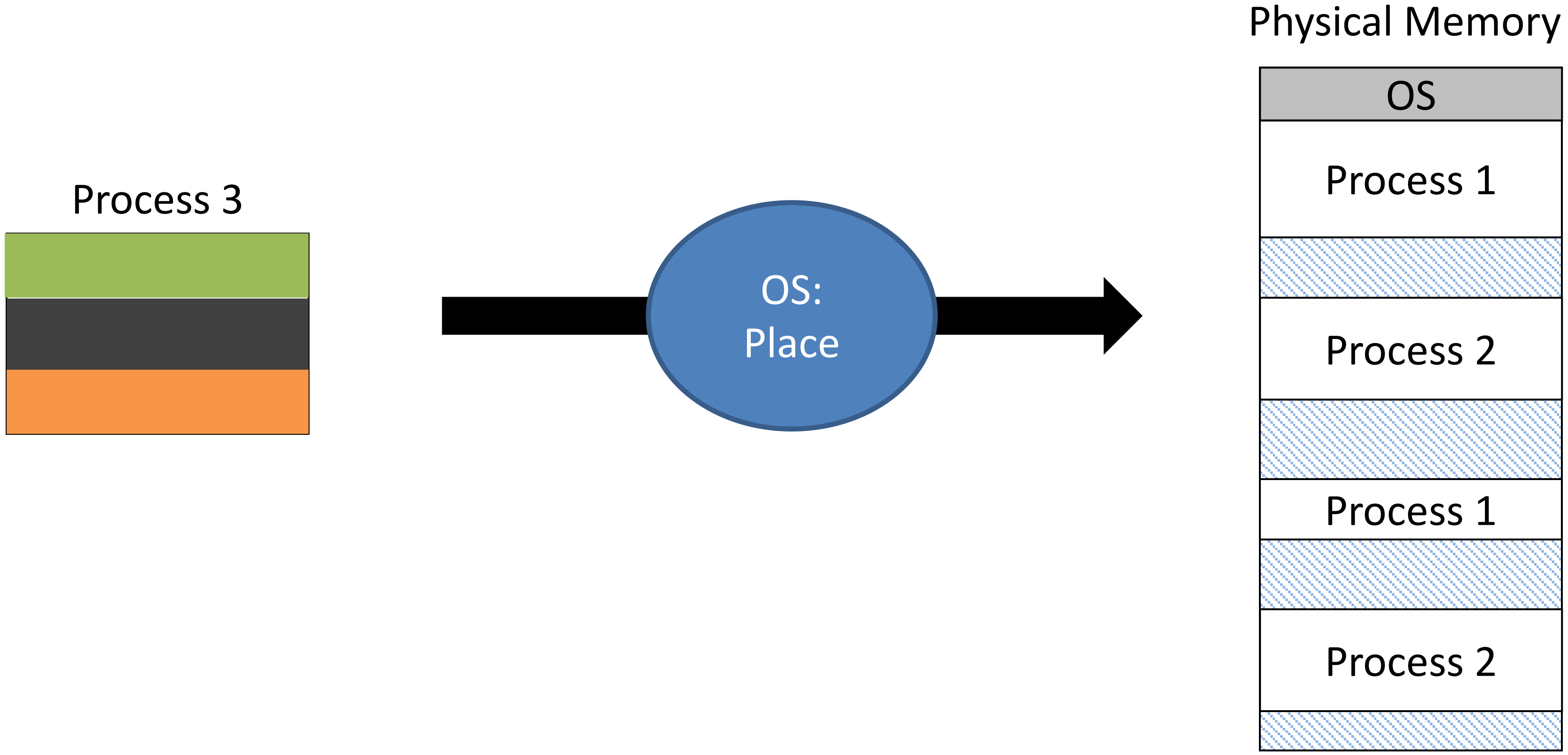
- General solution: don't require all of a process's memory to be in one piece!





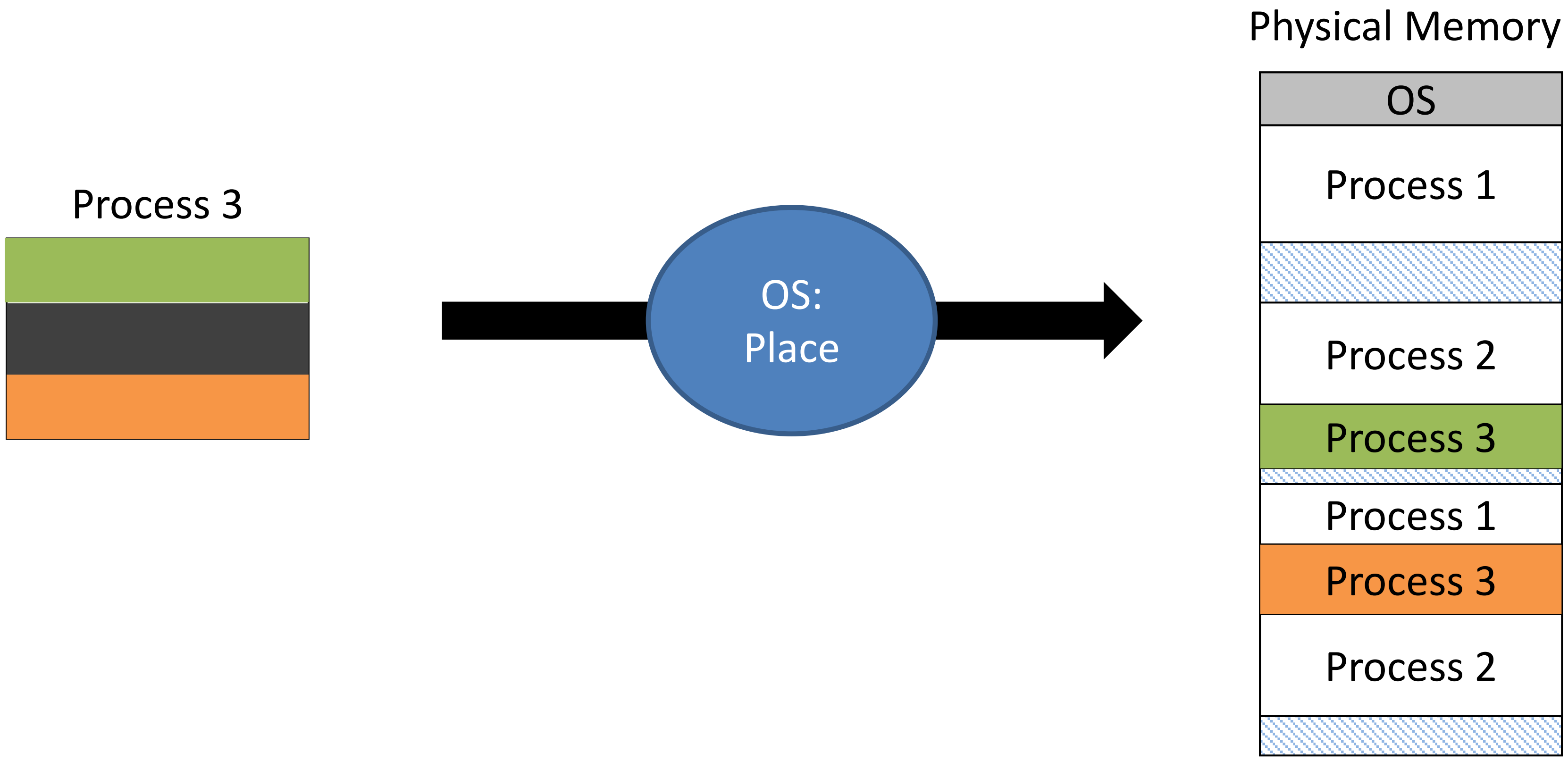
Problem Summary: Placement

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



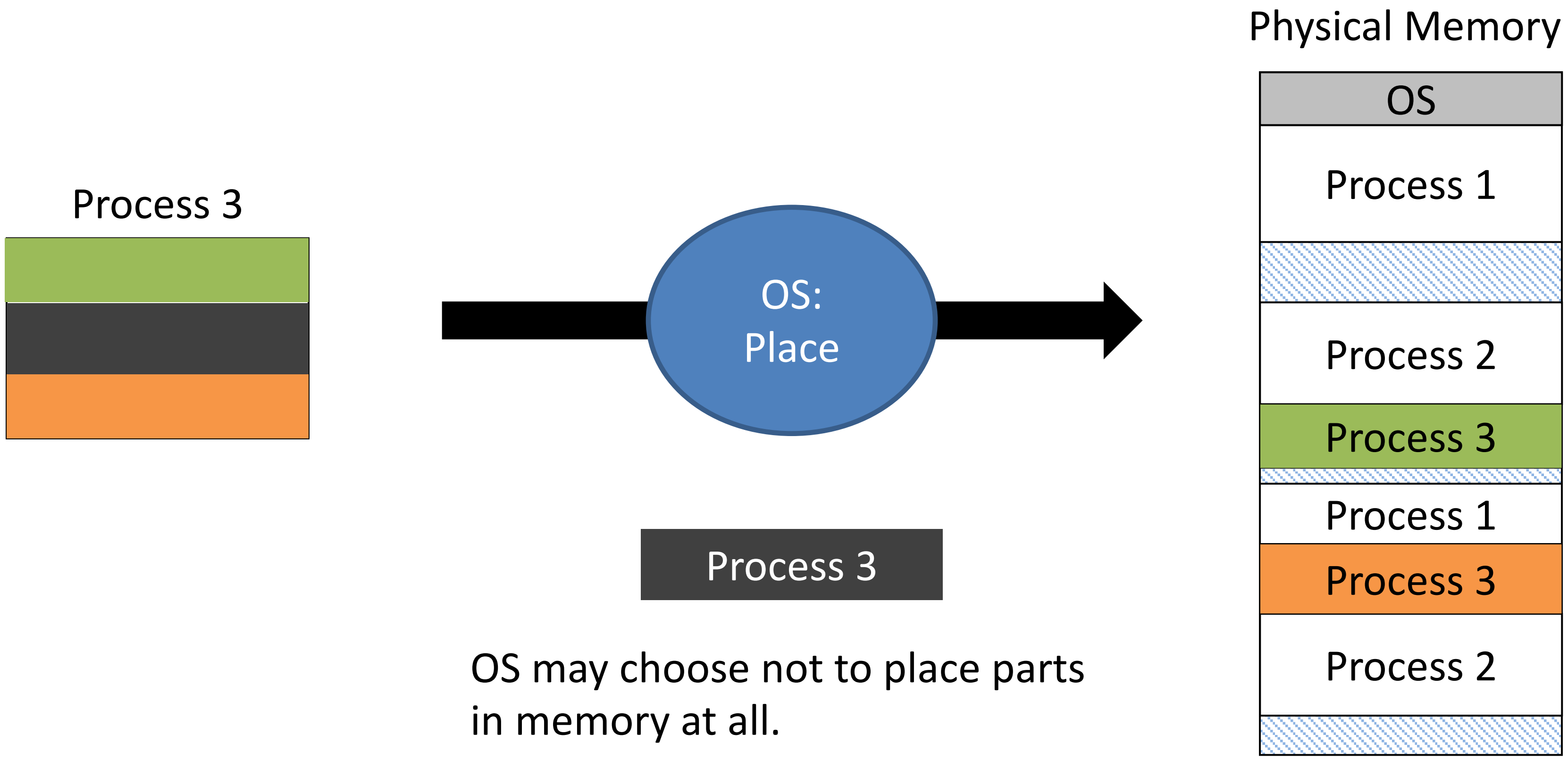
Problem Summary: Placement

- 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!



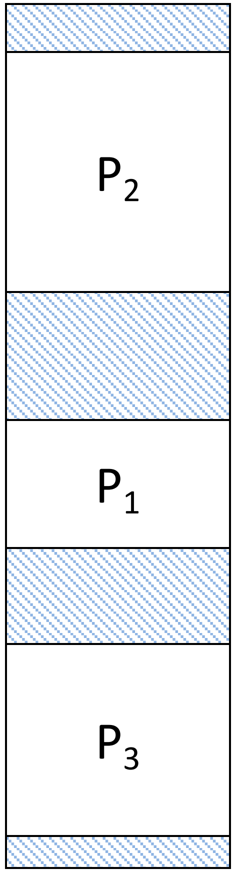
OS may choose not to place parts in memory at all.

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
- Sharing Space:
  - The more processes there are, the less memory each individually can have

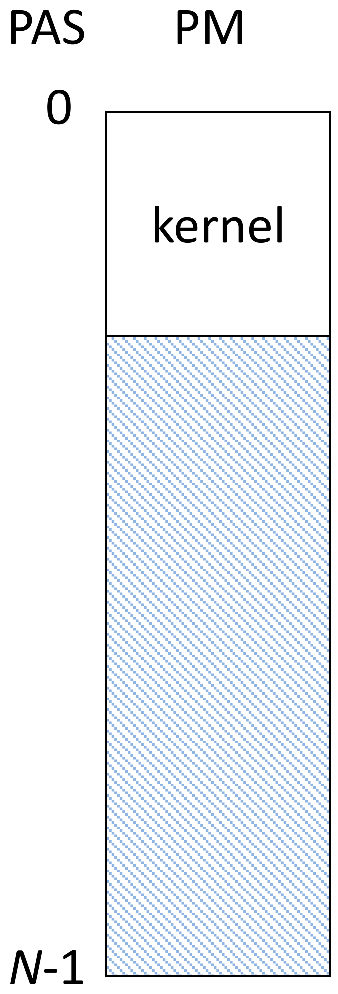


Compiler's View of Memory

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

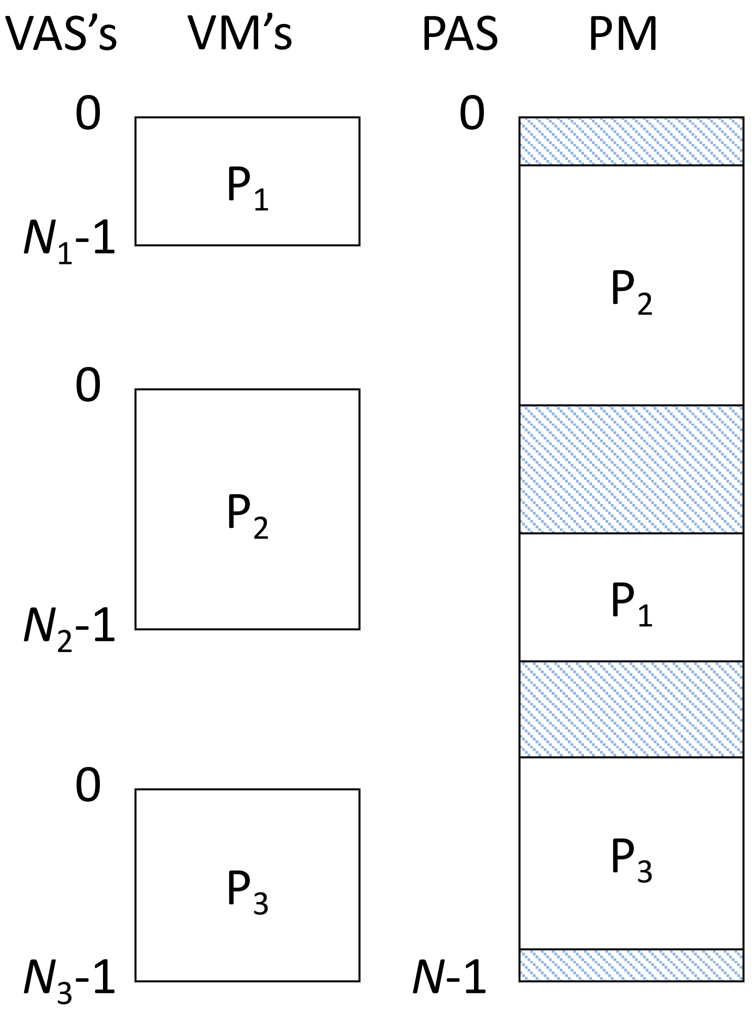
Address Spaces

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



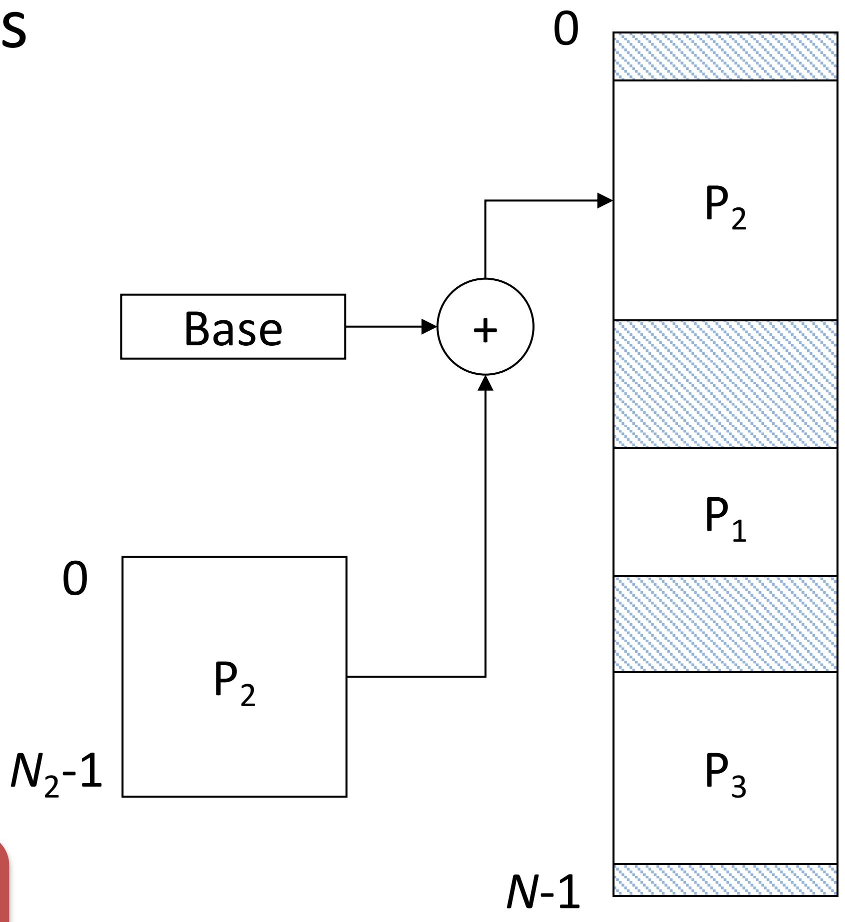
Virtual vs. Physical Addressing

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



Hardware for Virtual Addressing

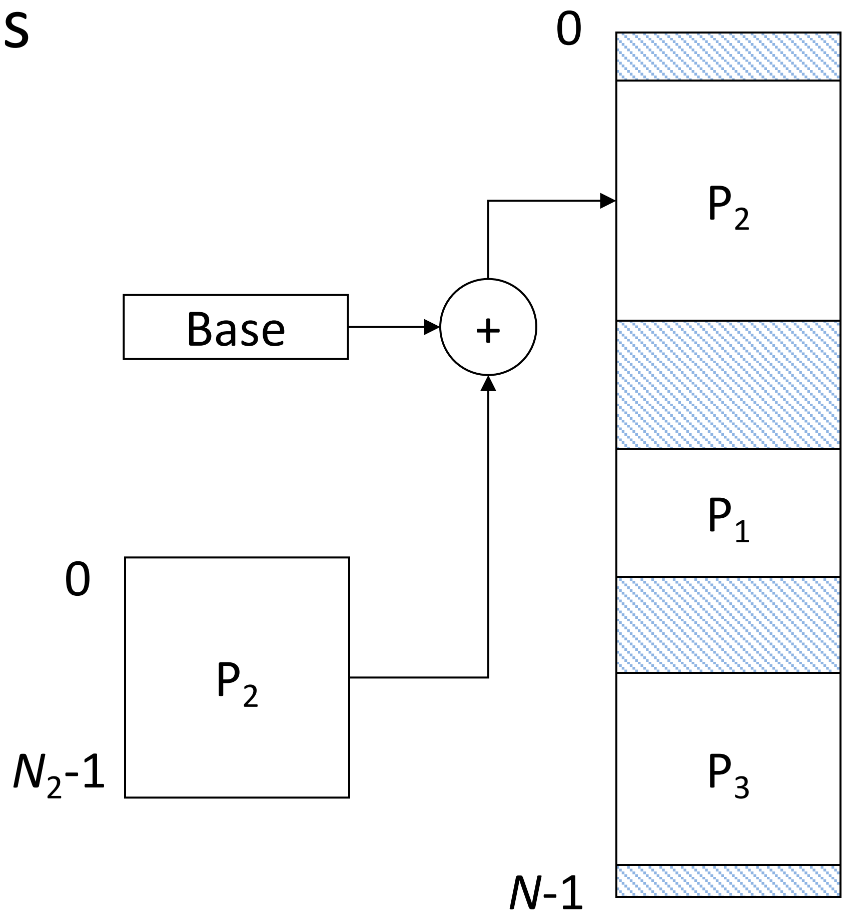
- Base register filled with start address
- To translate address, add base
- Achieves “relocation”: process’s physical memory location could be moved.
- To move process: change base



Note: This is a simpler model than what we do in real systems today. We’re still working toward the real thing.

Hardware for Virtual Addressing

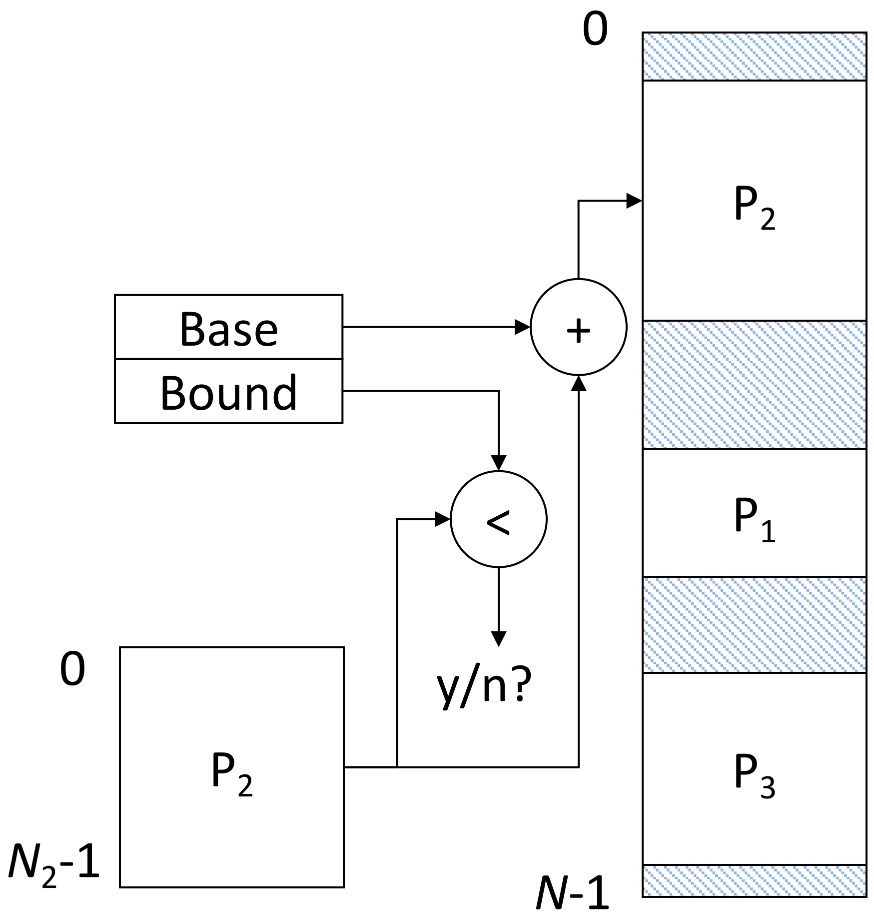
- Base register filled with start address
- To translate address, add base
- Achieves "relocation": process's physical memory location could be moved.
- To move process: change base
- Protection?



Protection

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

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



Given what we currently know about memory, what must we do during a context switch?

- A. Allocate memory to the switching process
- B. Load the base and bound registers
- C. Convert logical to physical memory addresses

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

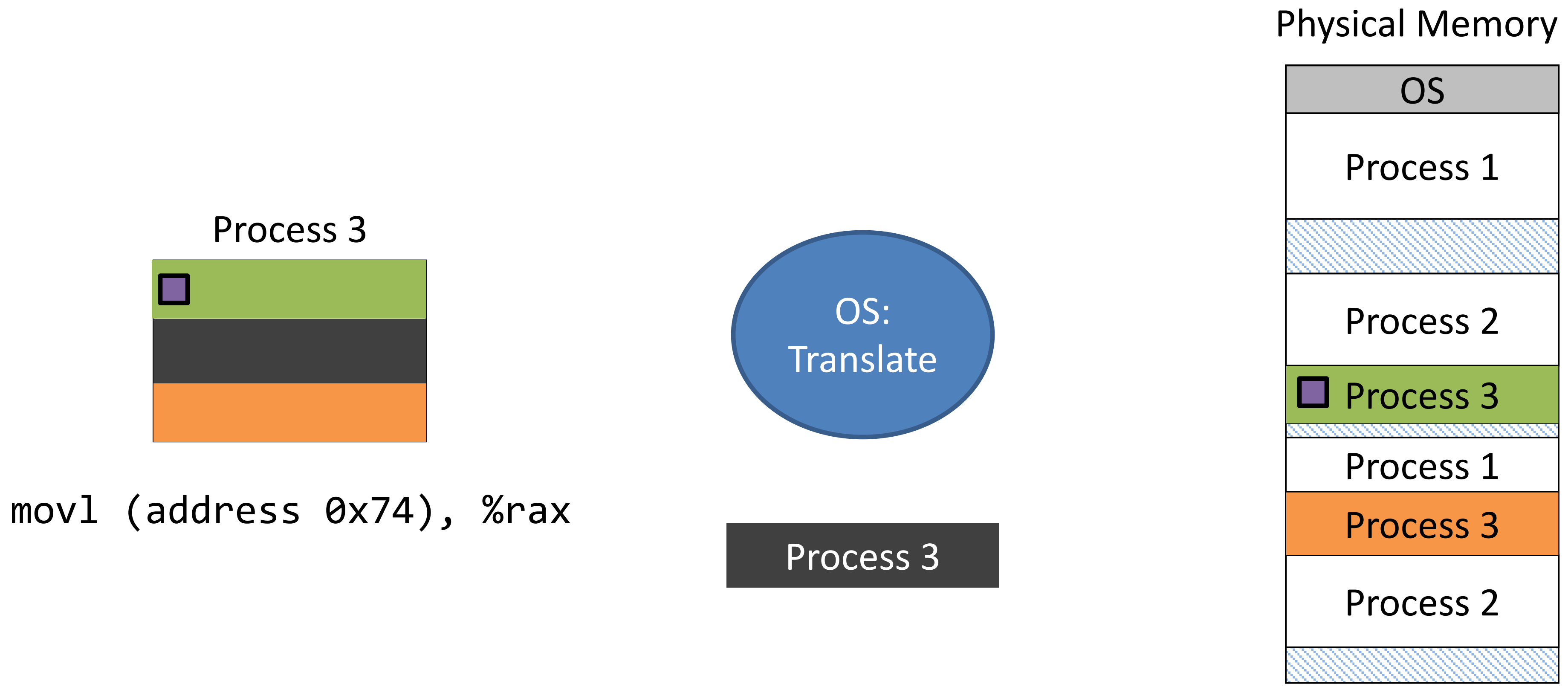
Problem Summary: Addressing

- 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.

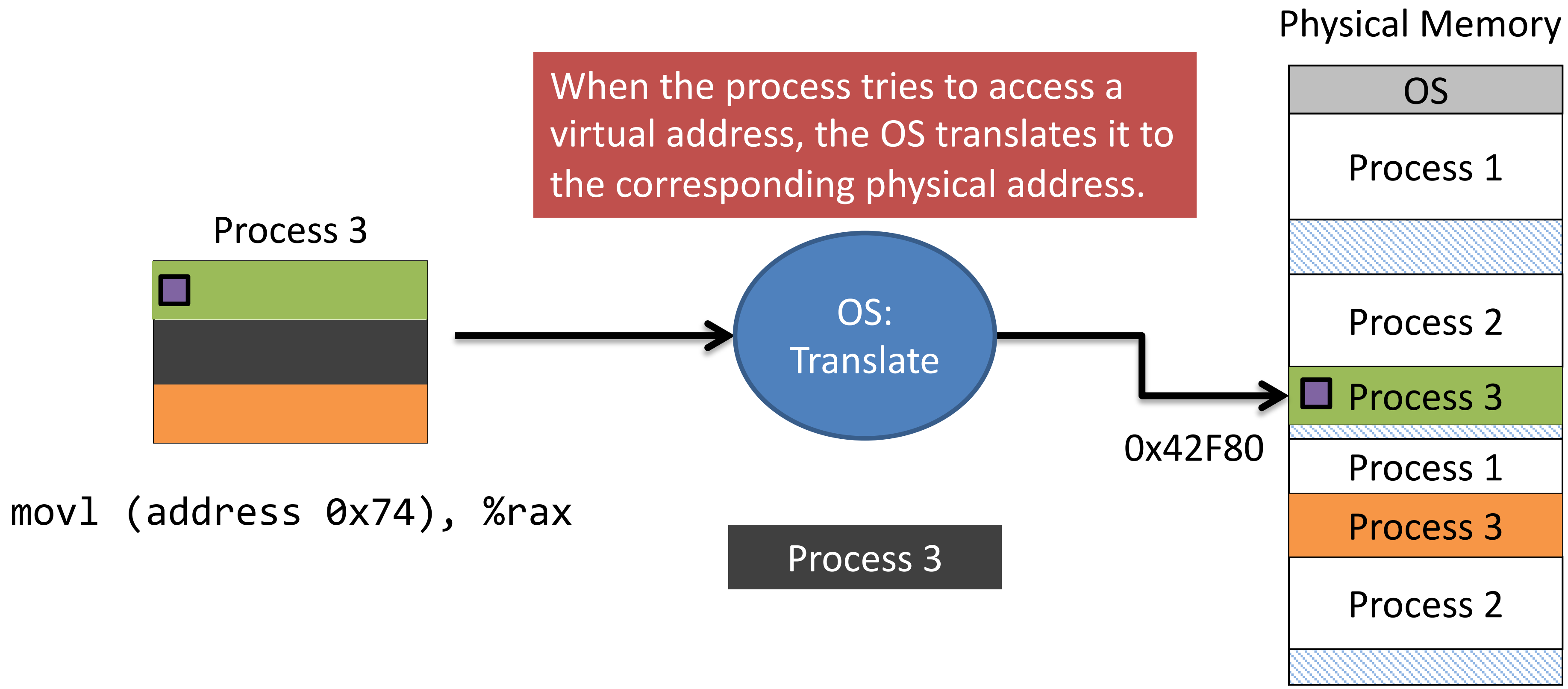
Problem Summary: Addressing

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



Problem Summary: Addressing

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



Let's combine these ideas:

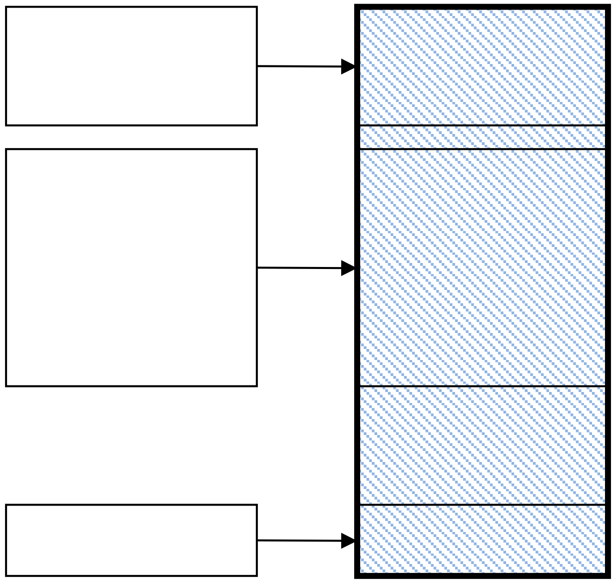
1. Allow process memory to be divided up into multiple pieces.

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

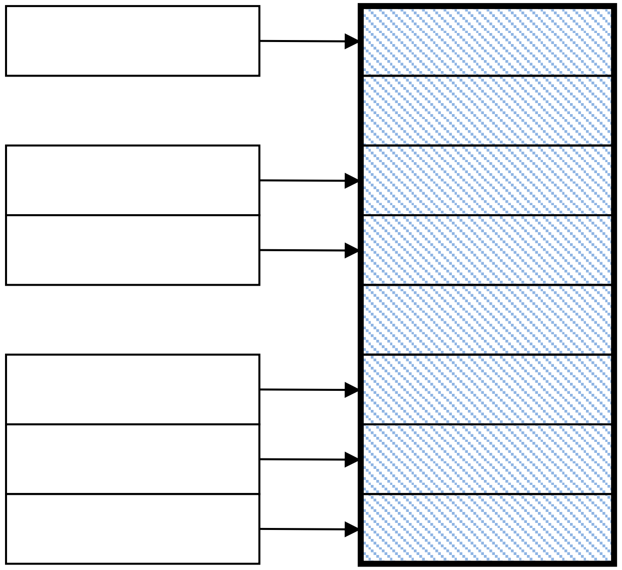
Result: Keep a table to store the mapping of each region.

Two (Real) Approaches

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



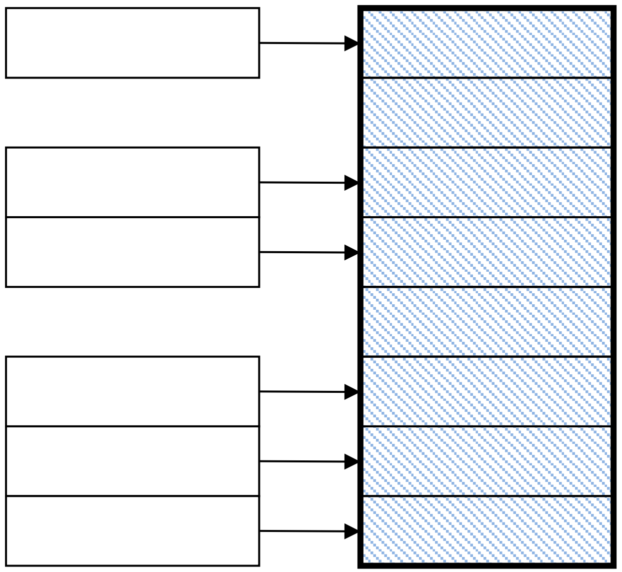
- Paged address space/memory
- Partition address space and memory into pages
- <u>All pages are the same size</u>



Paging

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
- <u>All pages are the same size</u>



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.

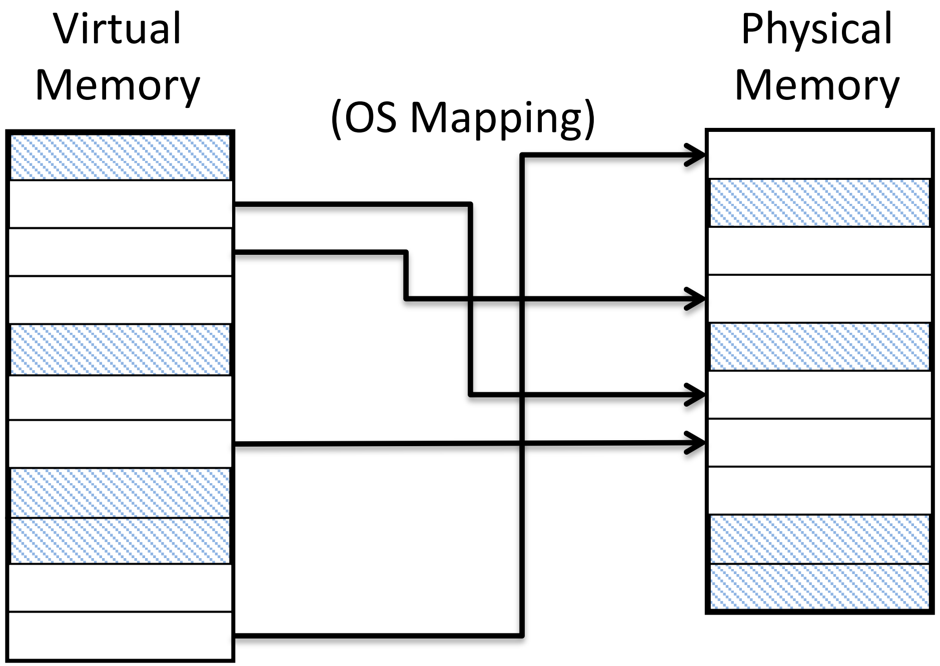
Main Idea

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

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

Main Idea

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



Implications for fragmentation?

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

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

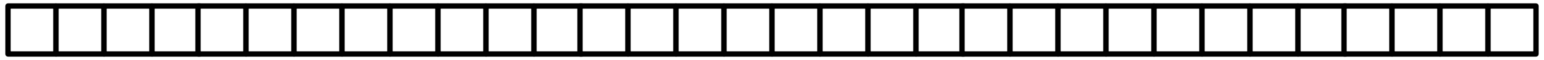
Addressing

- 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?

2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}
1024	2048	4096	8192	16384	32768

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.

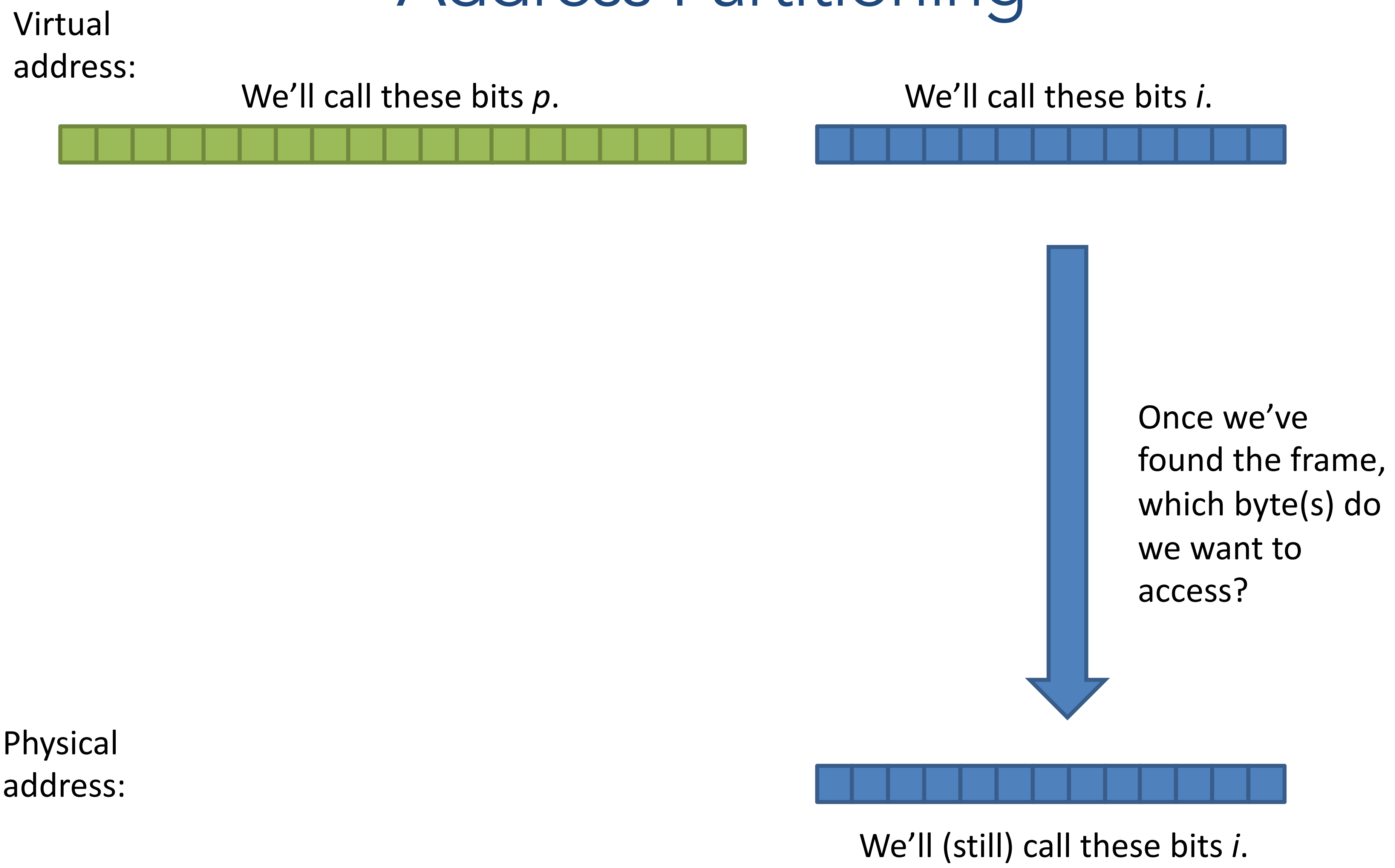
Example: 32-bit virtual addresses

- 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.

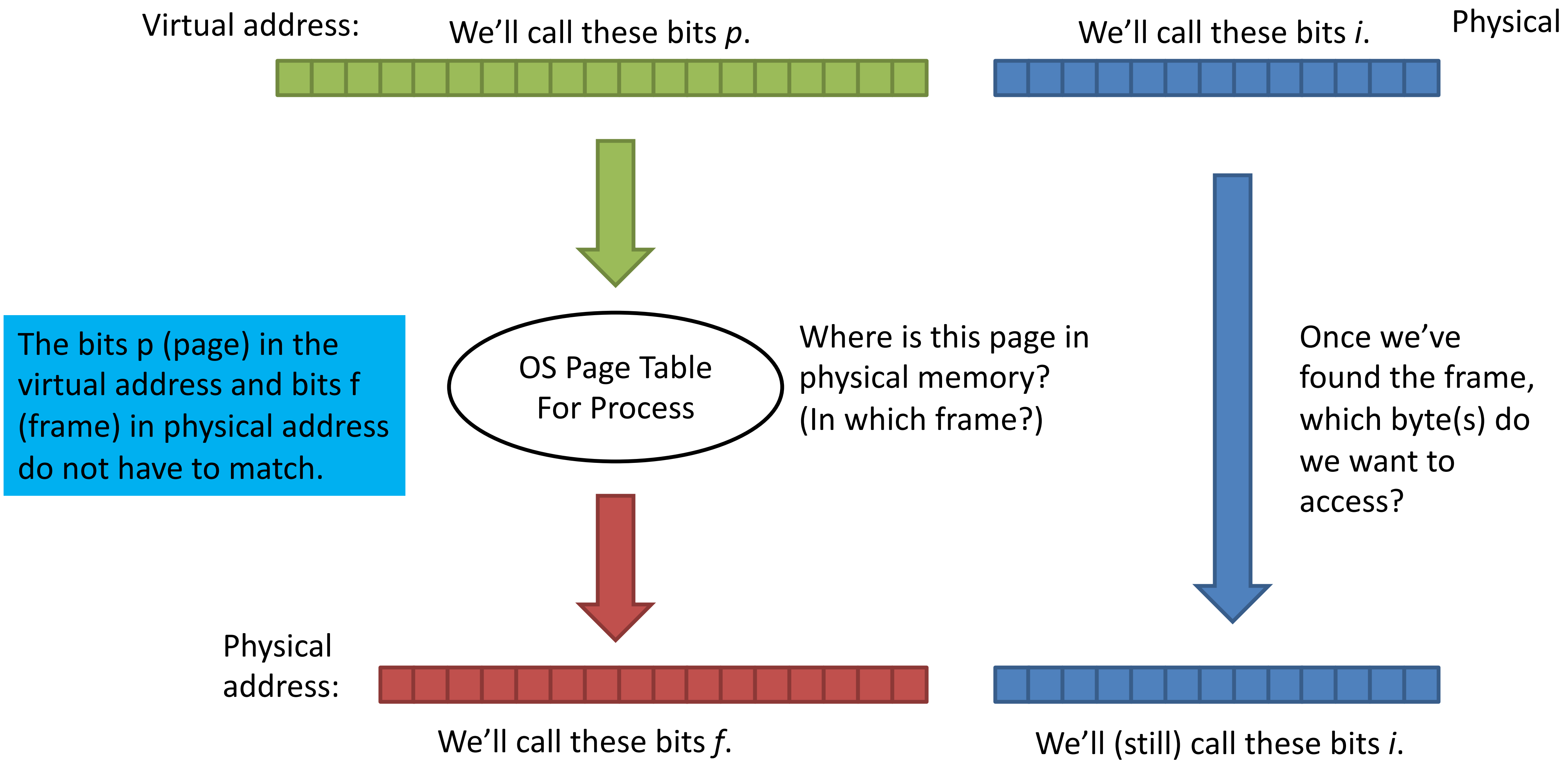
Example: 32-bit virtual addresses

- 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.

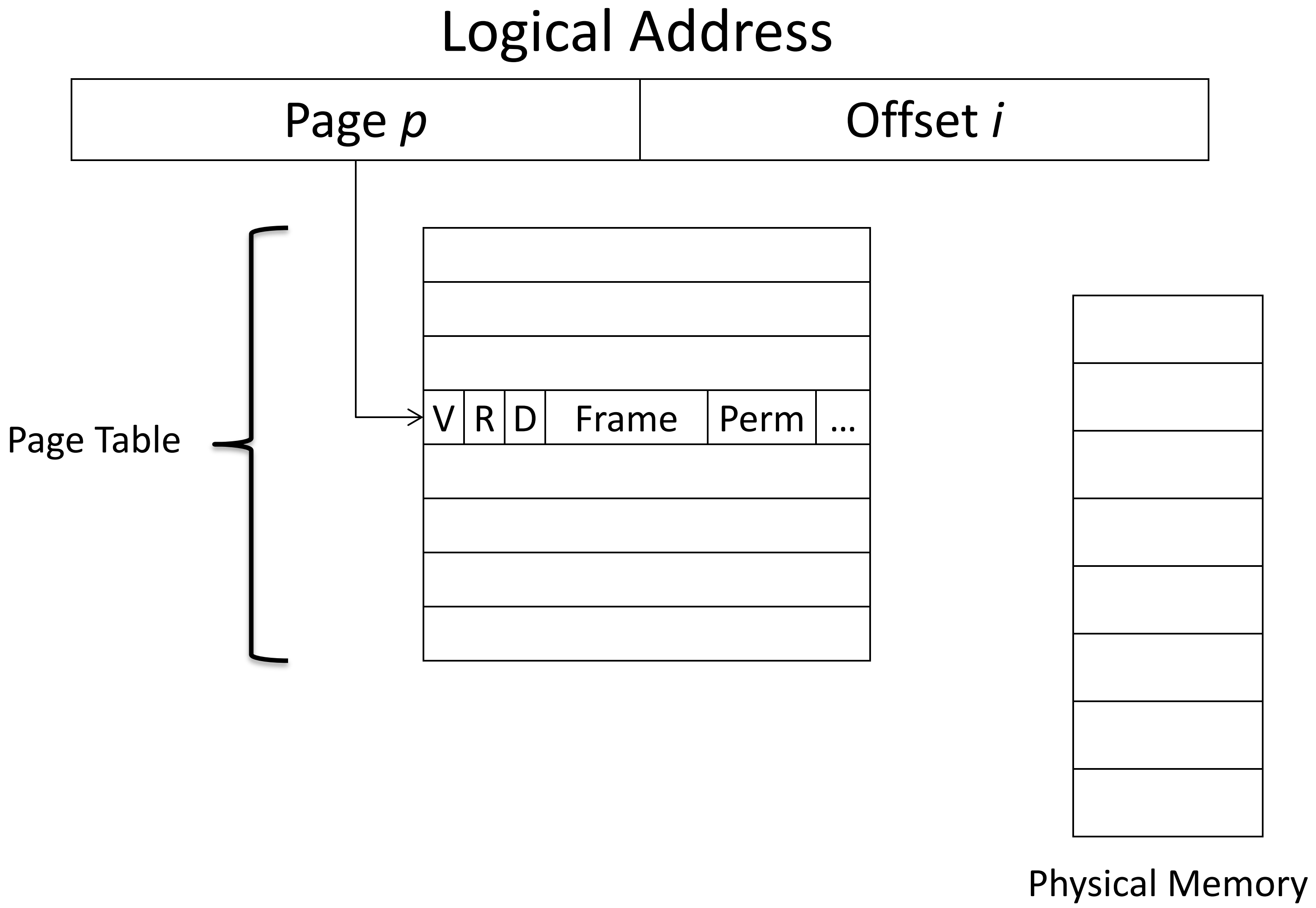
Address Partitioning



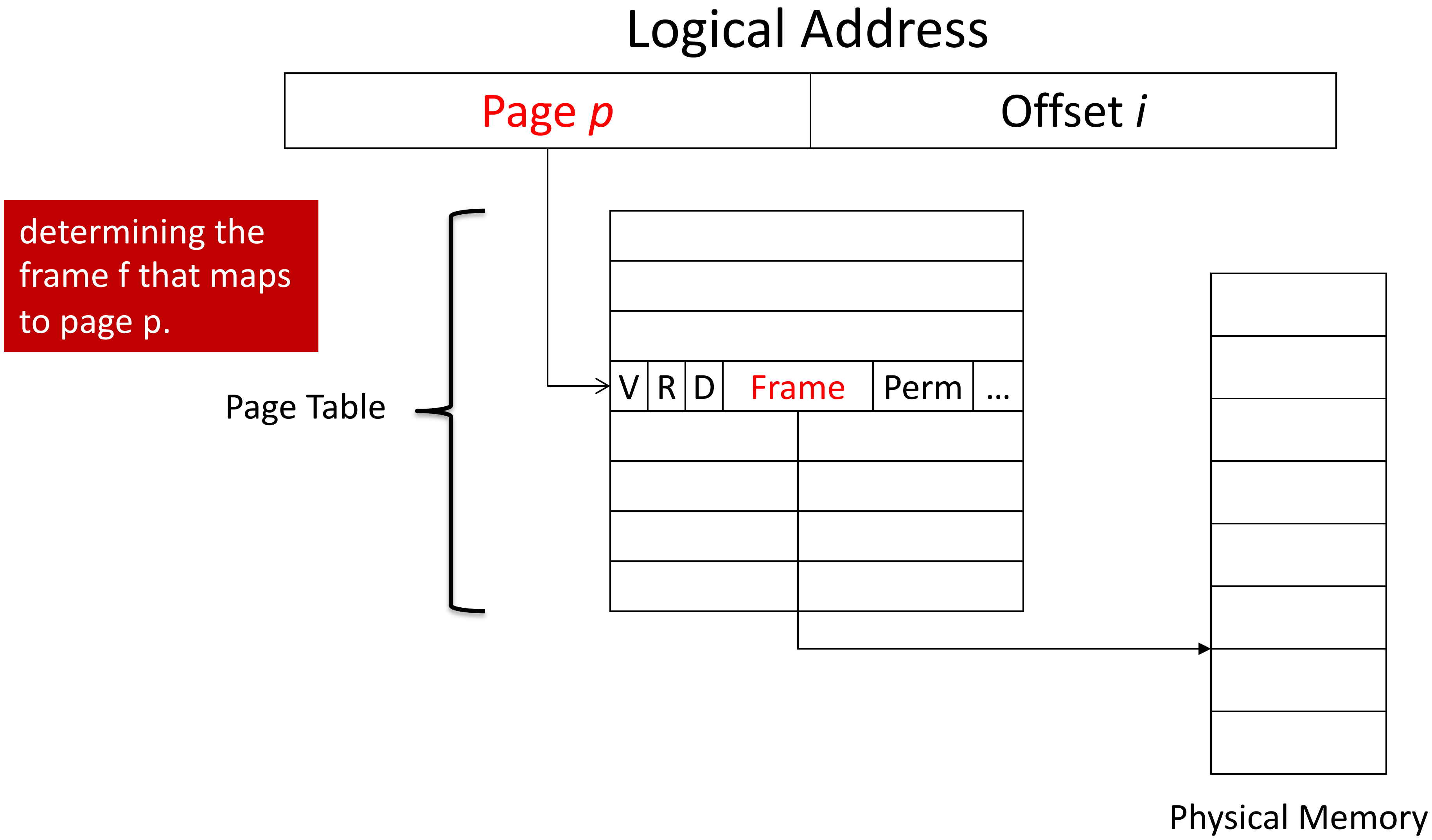
Address Partitioning



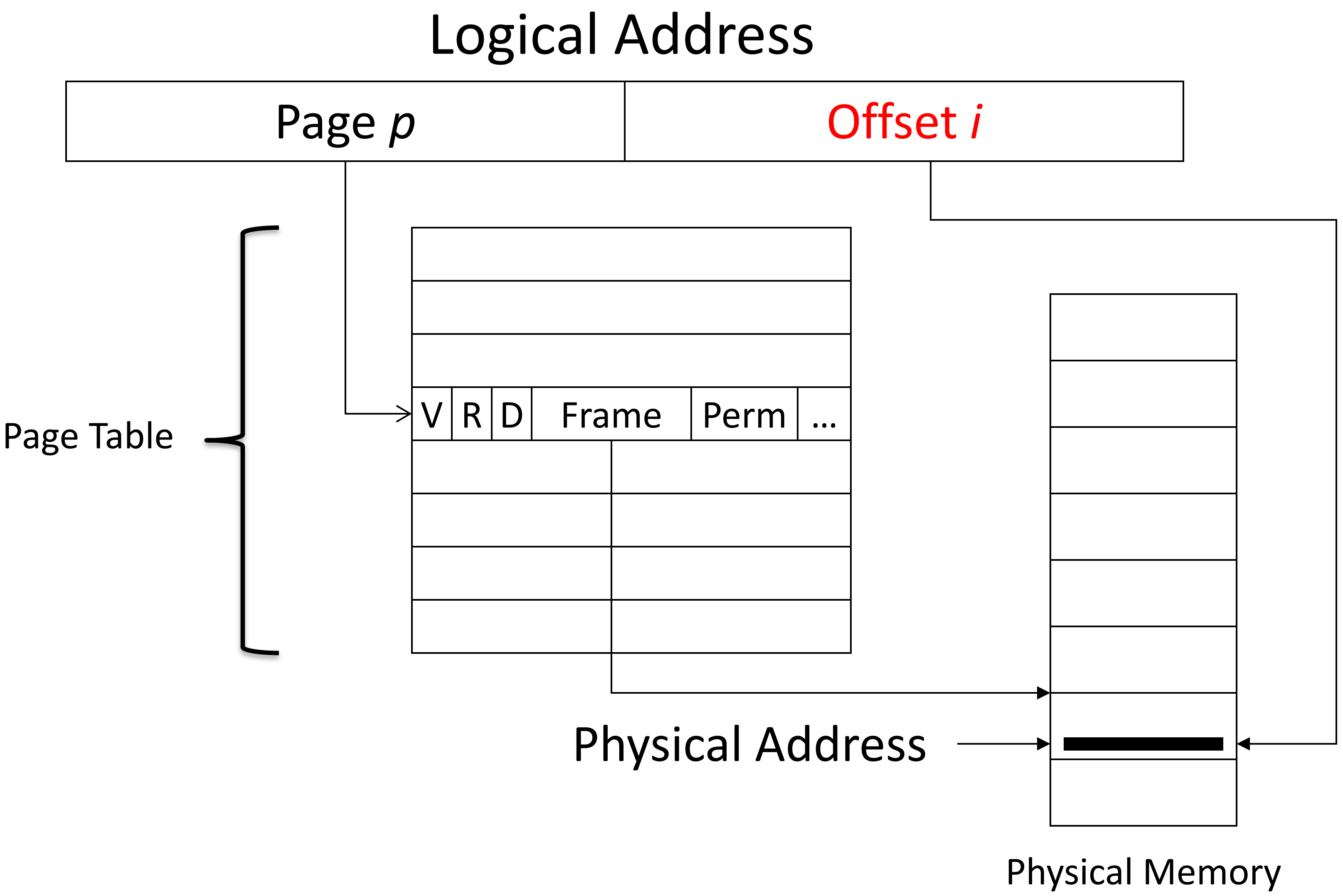
Address Translation



Address Translation



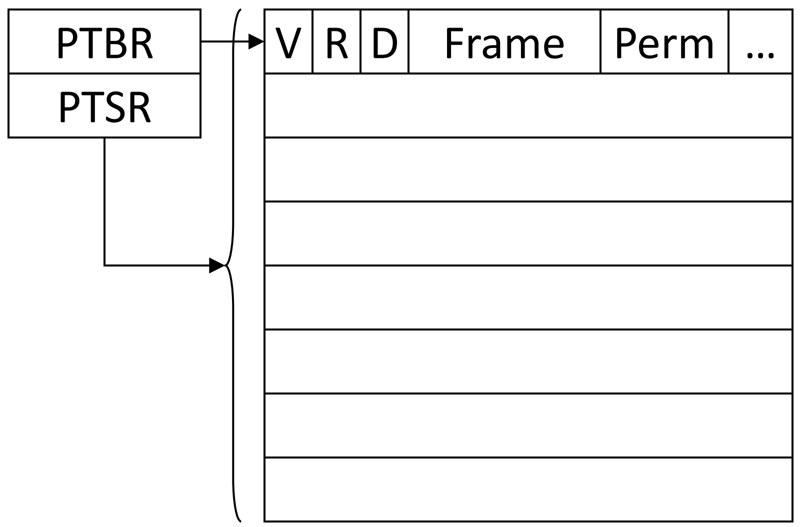
Address Translation



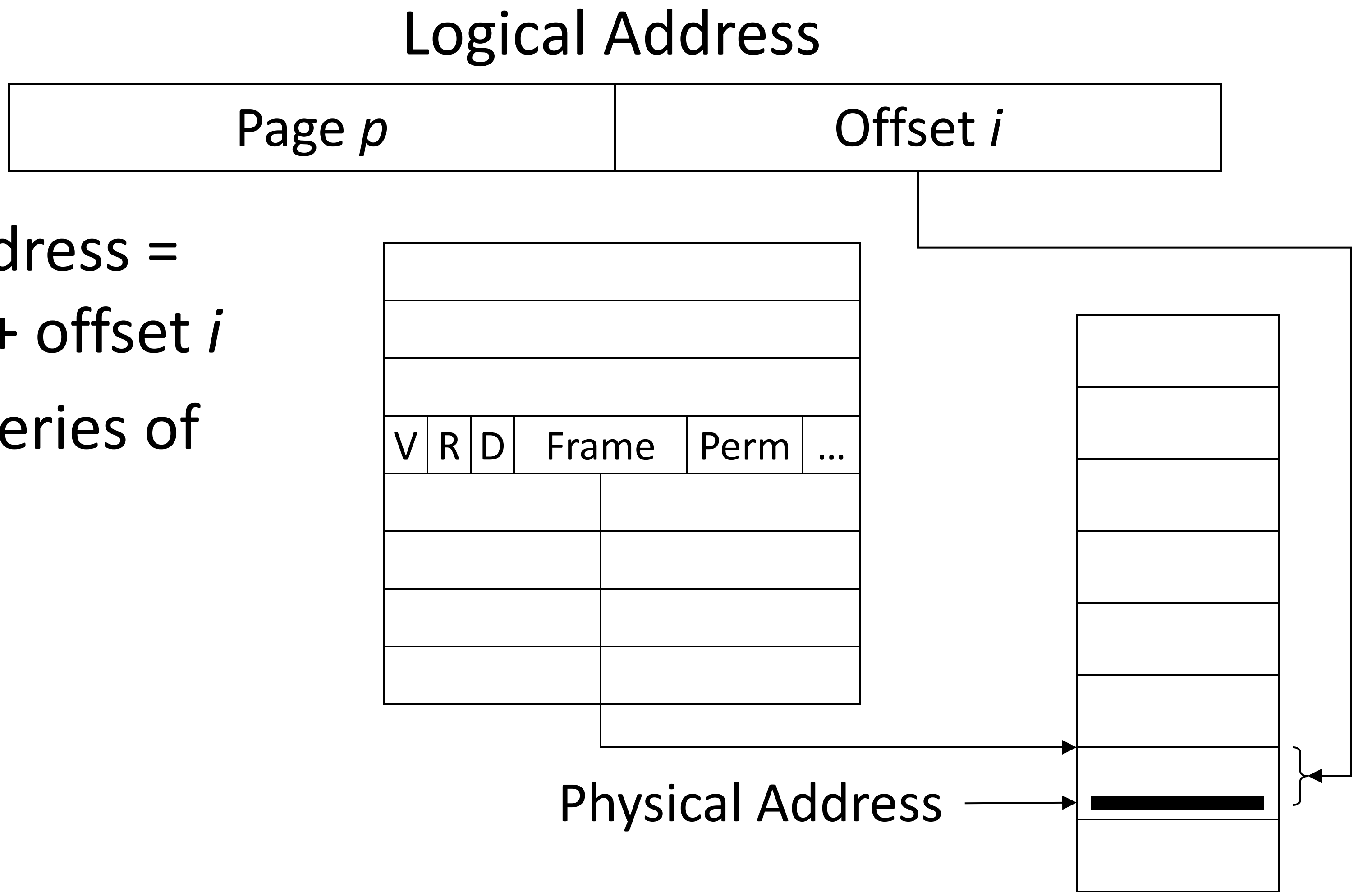
determining the byte offset within the page.

Page Table

- One table per process
- Table entry elements
  - V: valid bit
  - R: referenced bit
    - (how recently have we used this page?)
  - D: dirty bit
  - Frame: location in physical memory
  - Perm: access permissions
- Table parameters in memory
  - Page table base register (start for current process)
  - Page table size register (bound for current process)

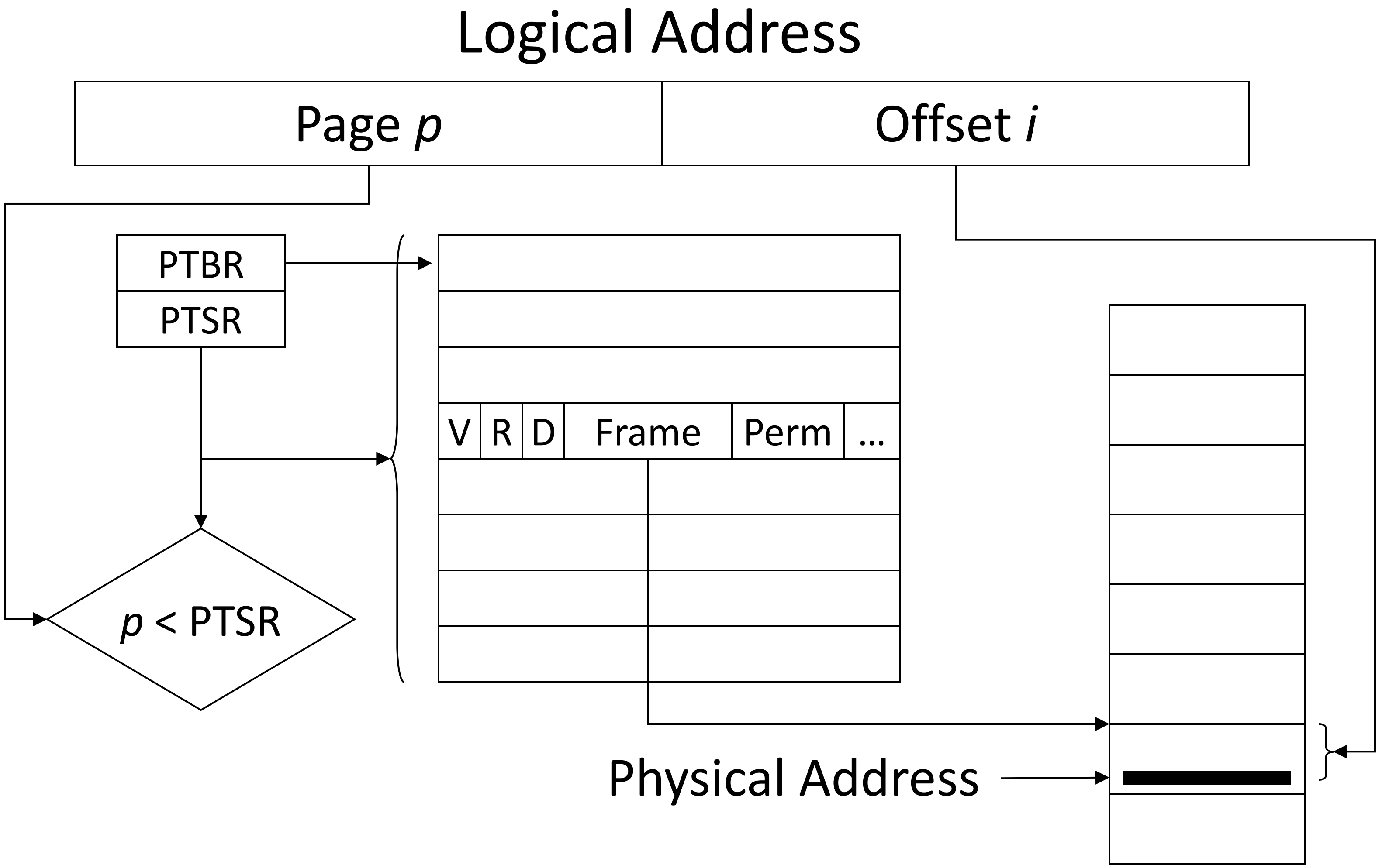


Address Translation

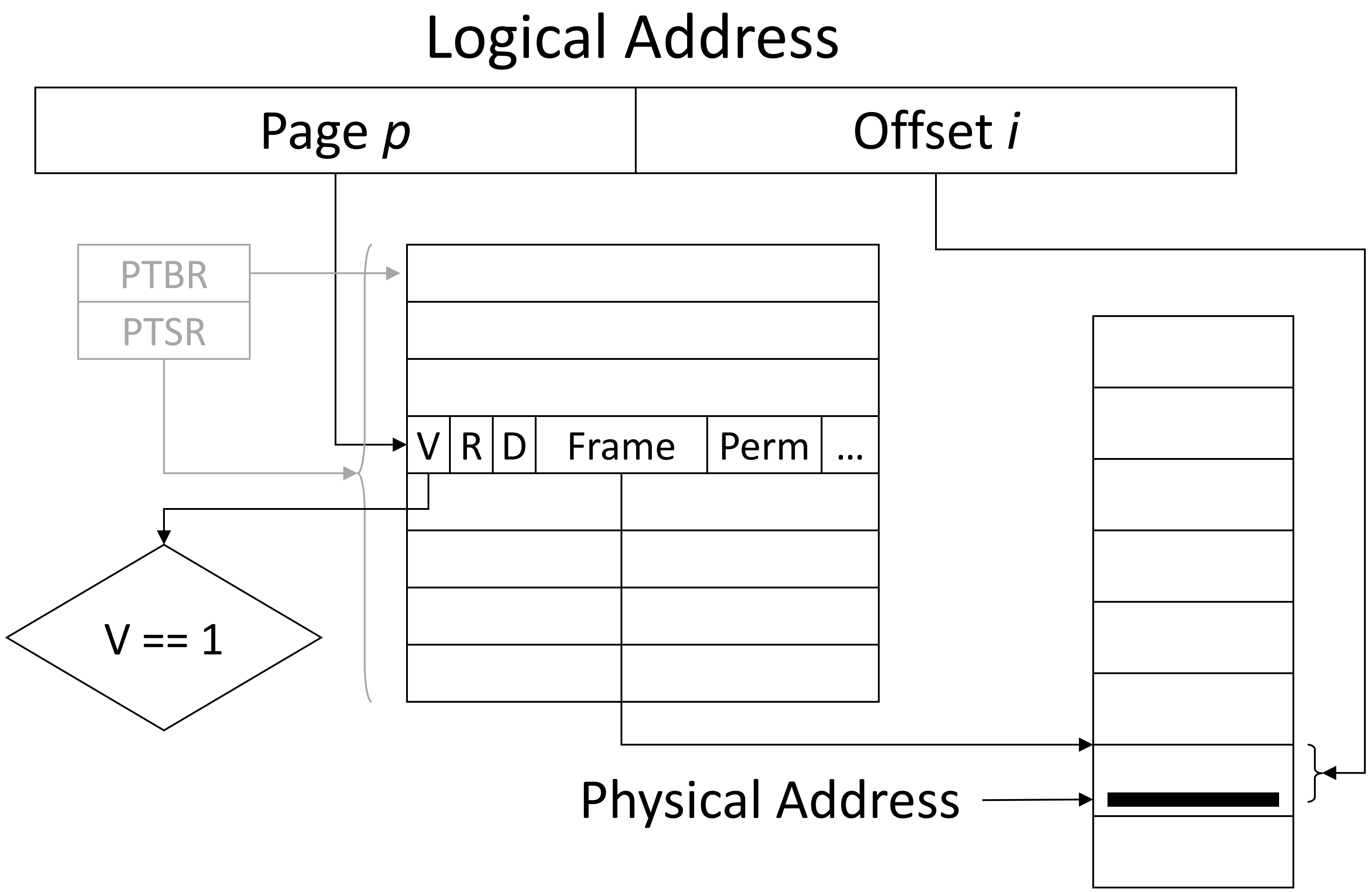


- Physical address = frame of *p* + offset *i*
- First, do a series of checks

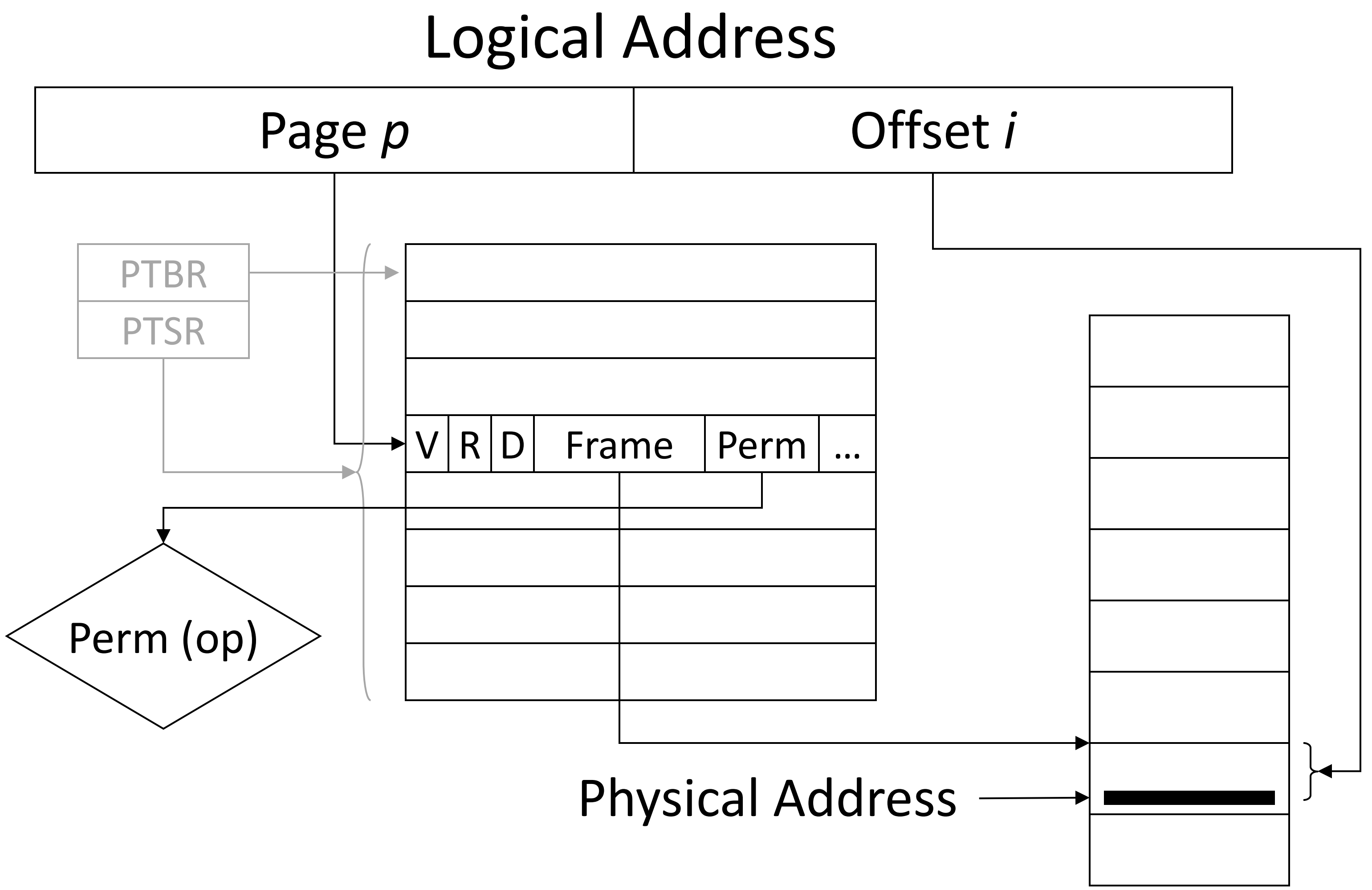
Check if Page *p* is Within Range



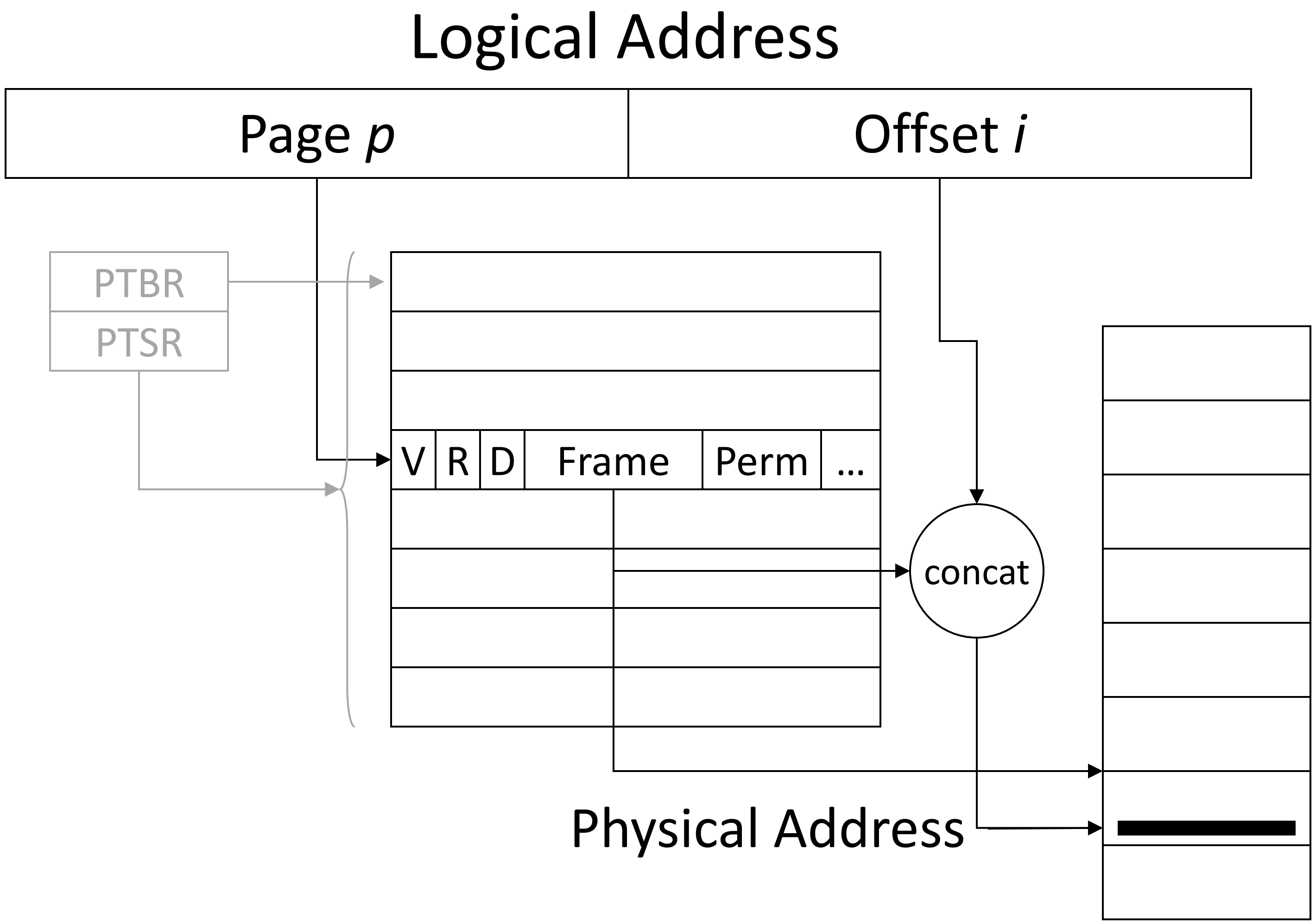
Check if Page Table Entry *p* is Valid



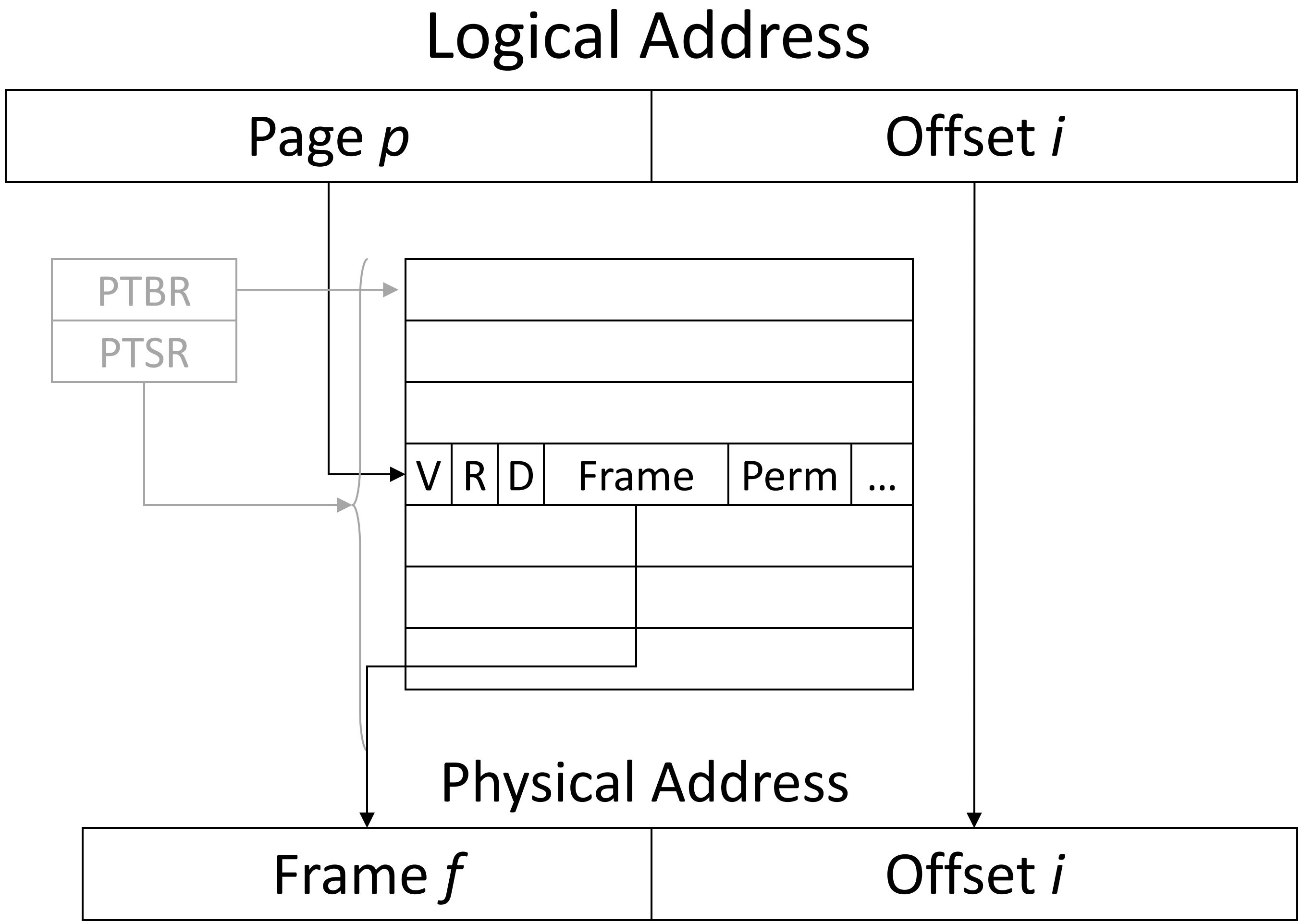
Check if Operation is Permitted



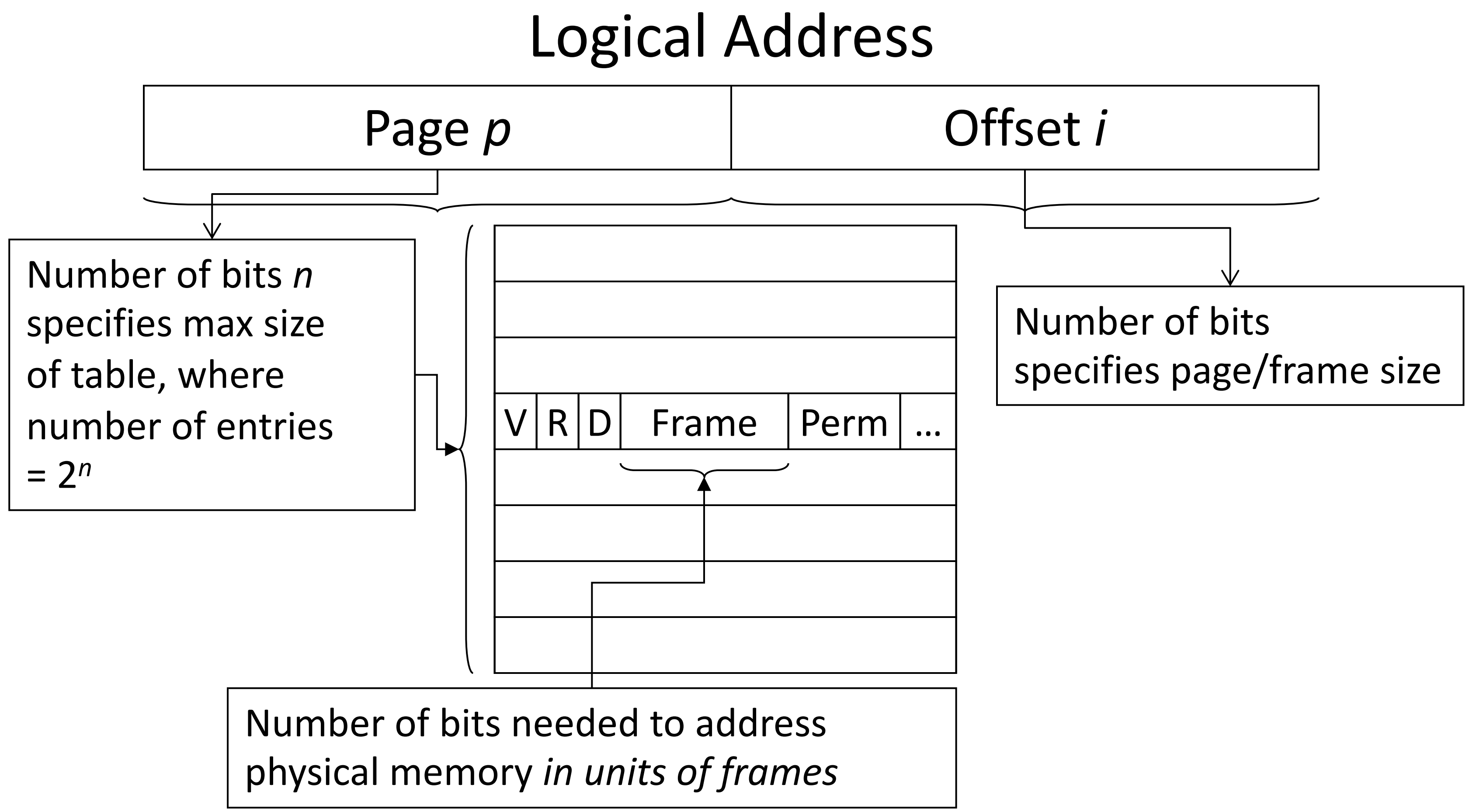
Translate Address



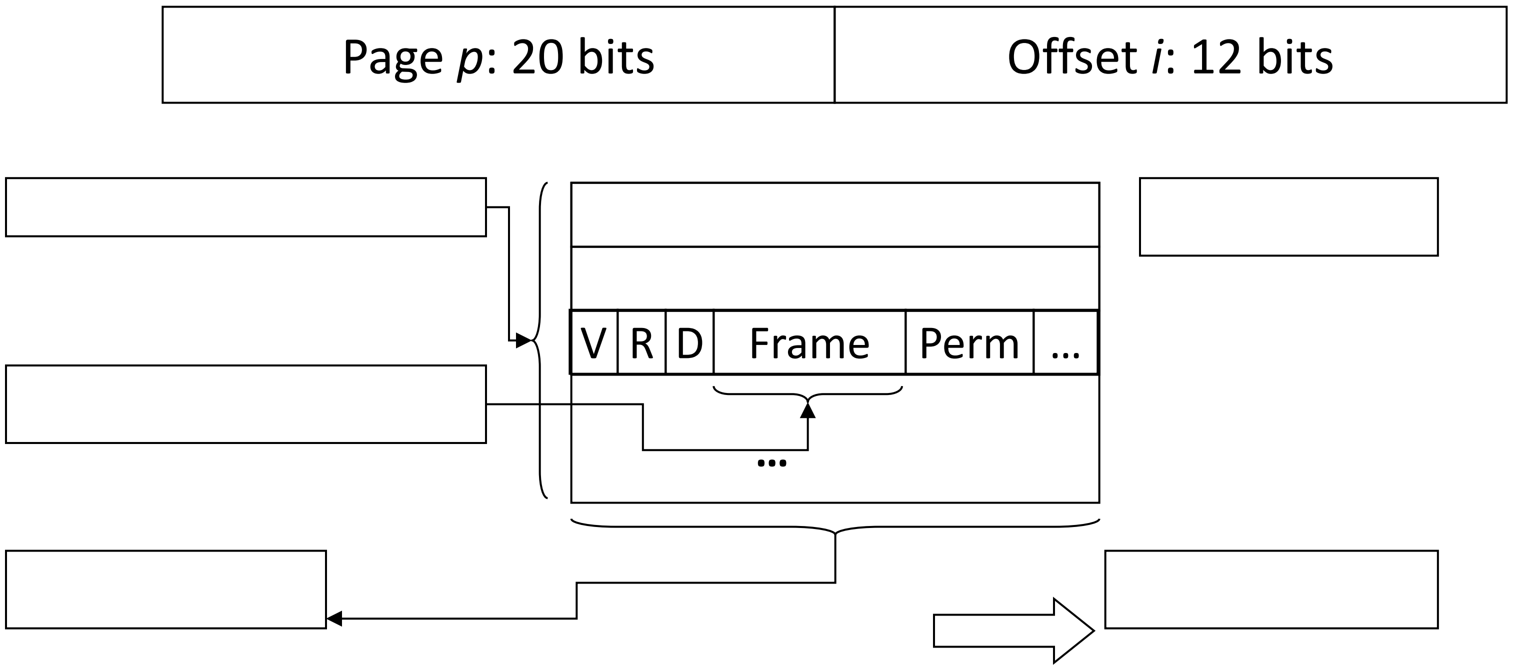
Physical Address by Concatenation



Sizing the Page Table



Example of Sizing the Page Table

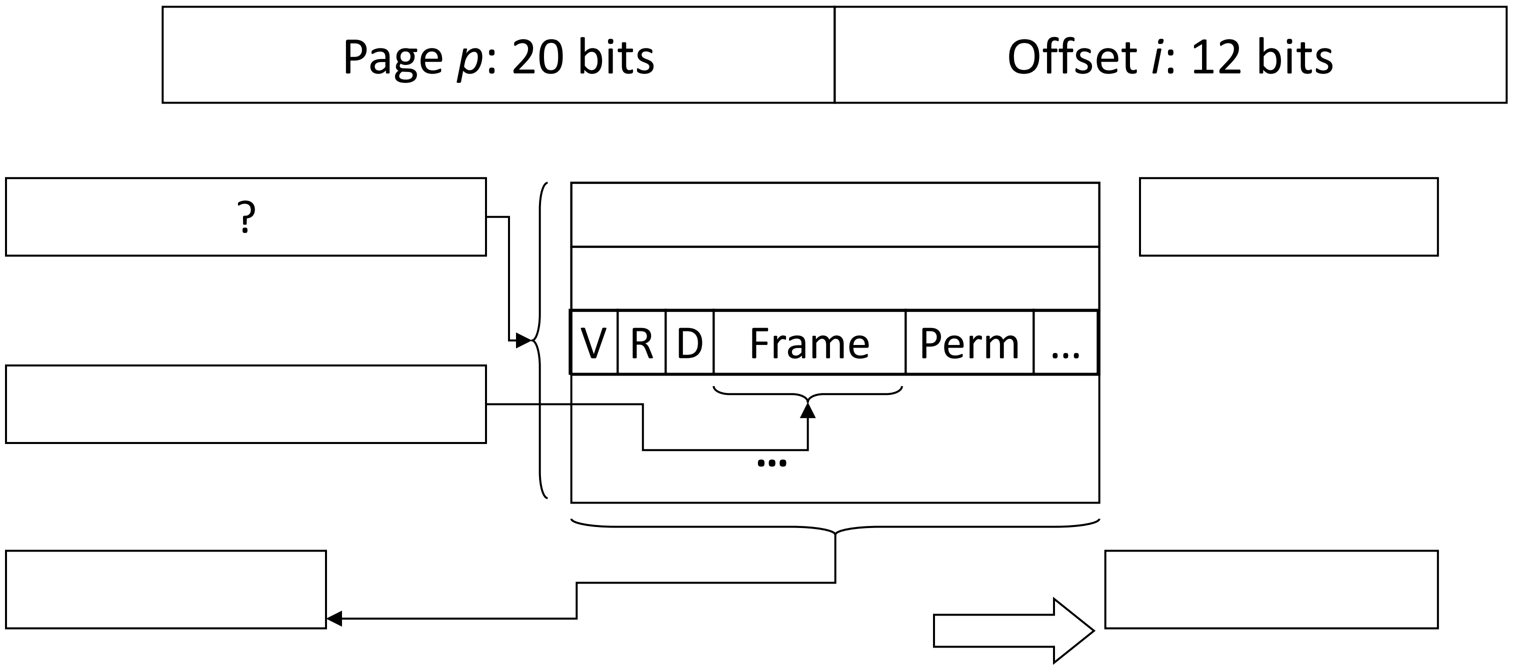


Given: 32 bit virtual addresses, 1 GB physical memory

– Address partition: 20 bit page number, 12 bit offset

2^{10}	2^{20}	2^{30}	2^{40}
1KB	1MB	1GB	1TB

Example of Sizing the Page Table



Given: 32 bit virtual addresses, 1 GB physical memory

– Address partition: 20 bit page number, 12 bit offset

2^{10}	2^{20}	2^{30}	2^{40}
1KB	1MB	1GB	1TB

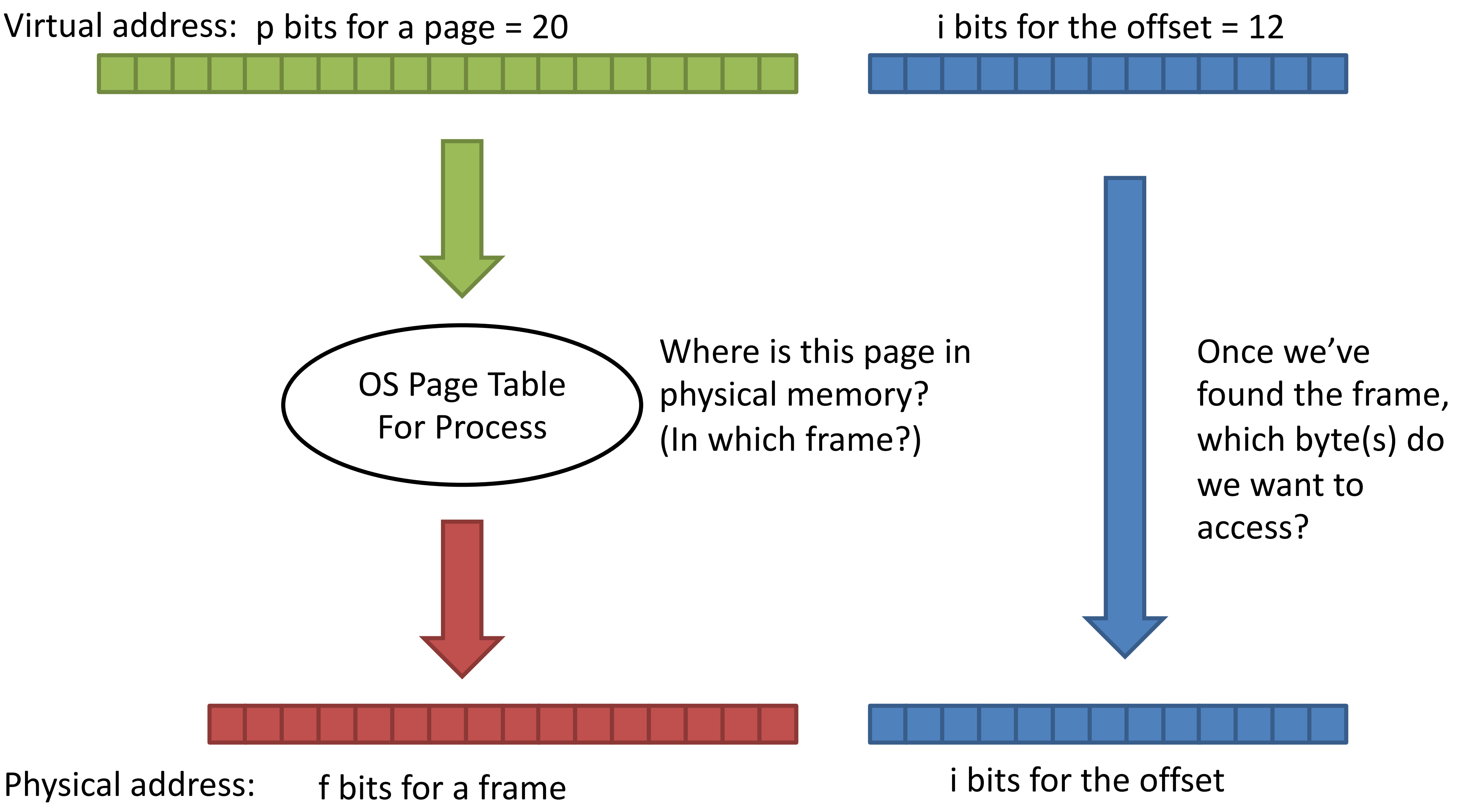
How many entries (rows) will there be in this page table?

A. 2^{12}

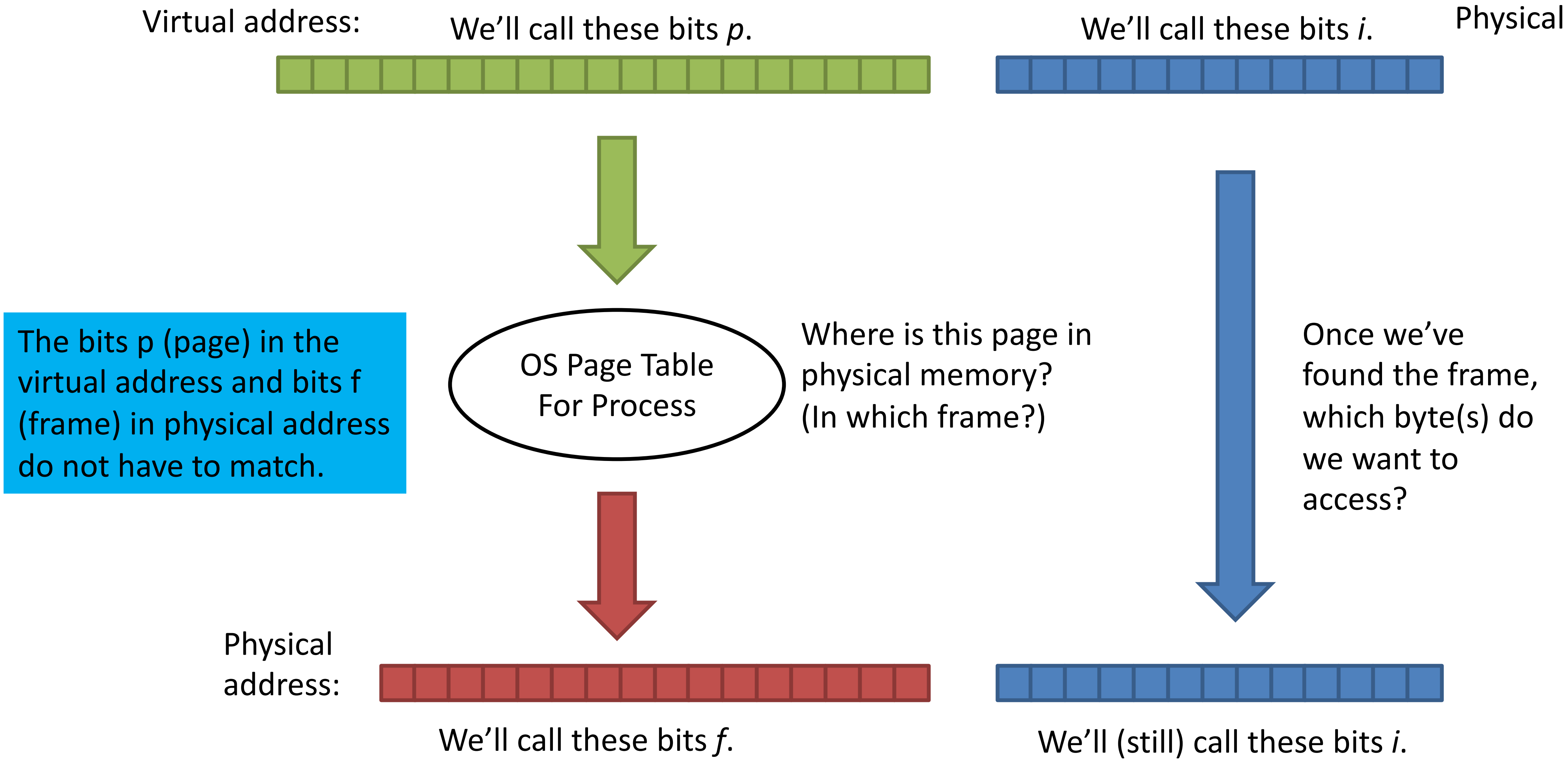
B. 2^{20}

C. 2^{30}

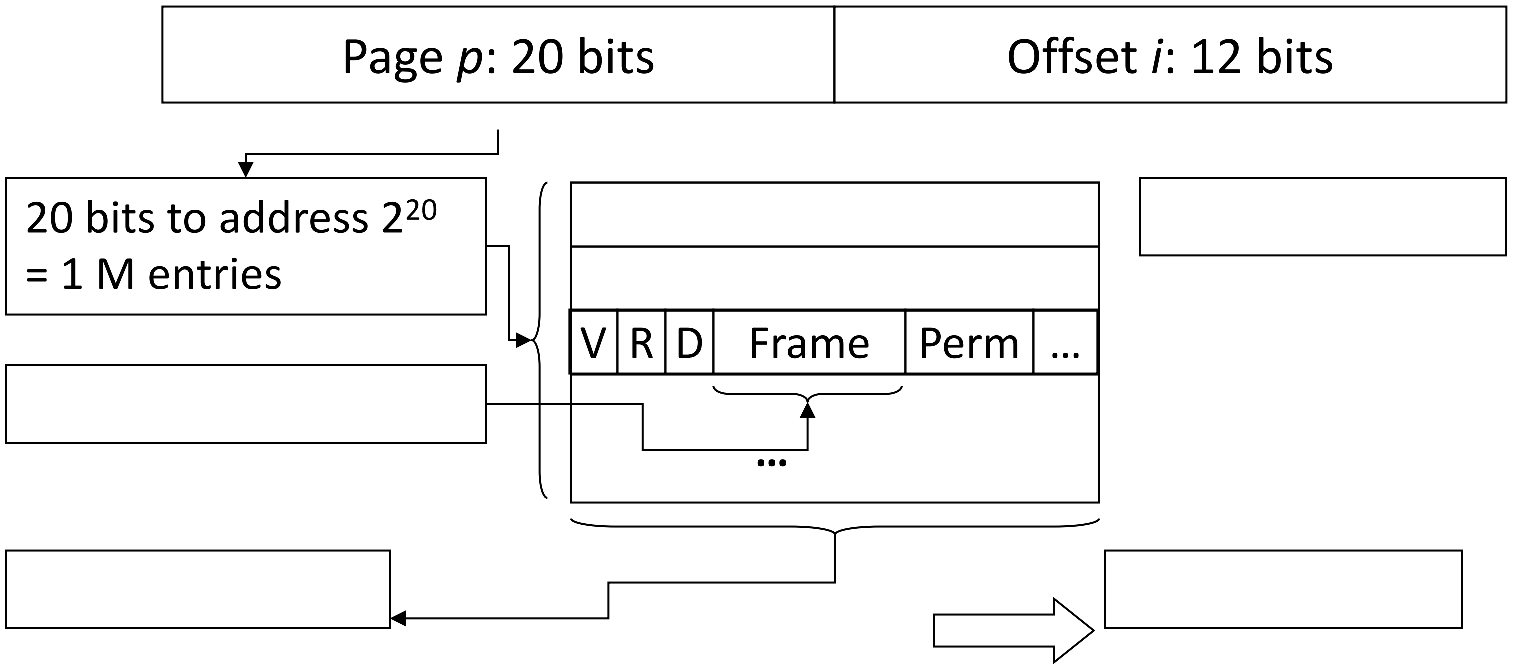
D. 2^{32}



Address Partitioning



Example of Sizing the Page Table

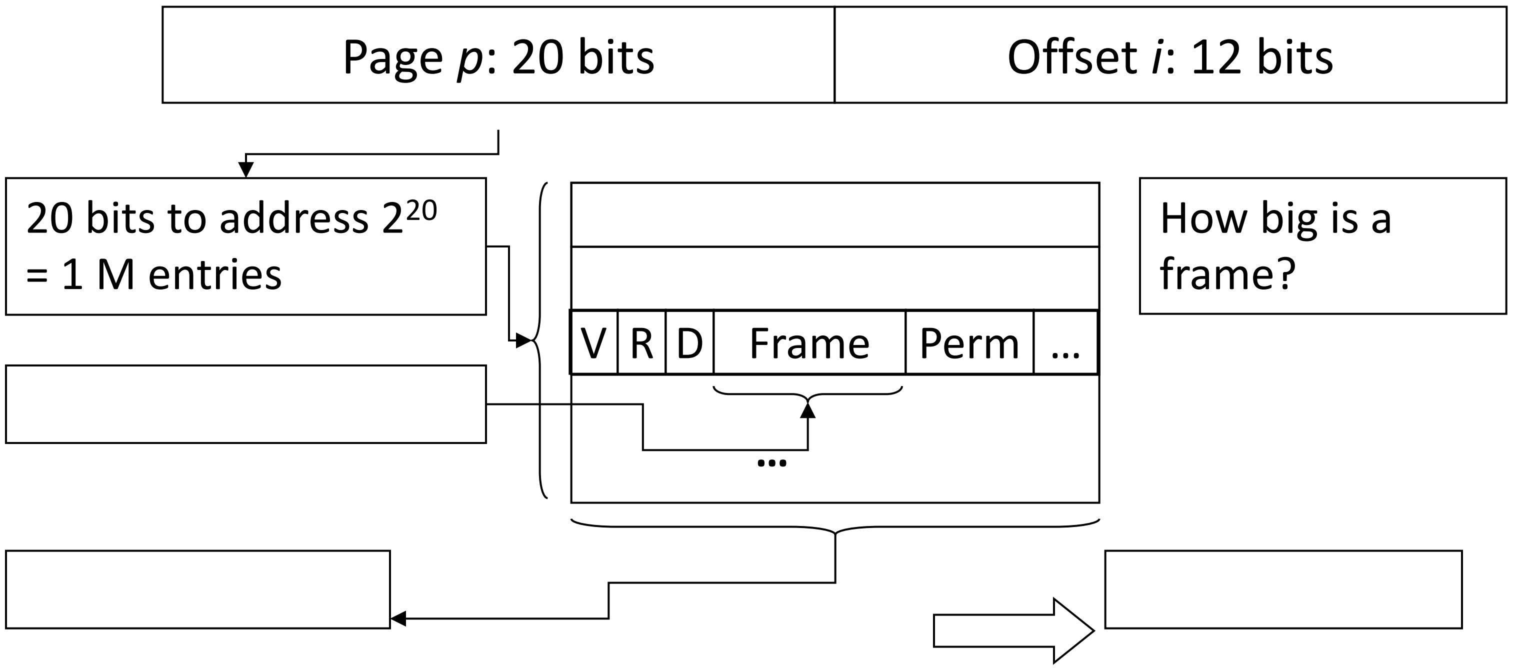


Given: 32 bit virtual addresses, 1 GB physical memory

– Address partition: 20 bit page number, 12 bit offset

2^{10}	2^{20}	2^{30}	2^{40}
1KB	1MB	1GB	1TB

Example of Sizing the Page Table



Given: 32 bit virtual addresses, 1 GB physical memory

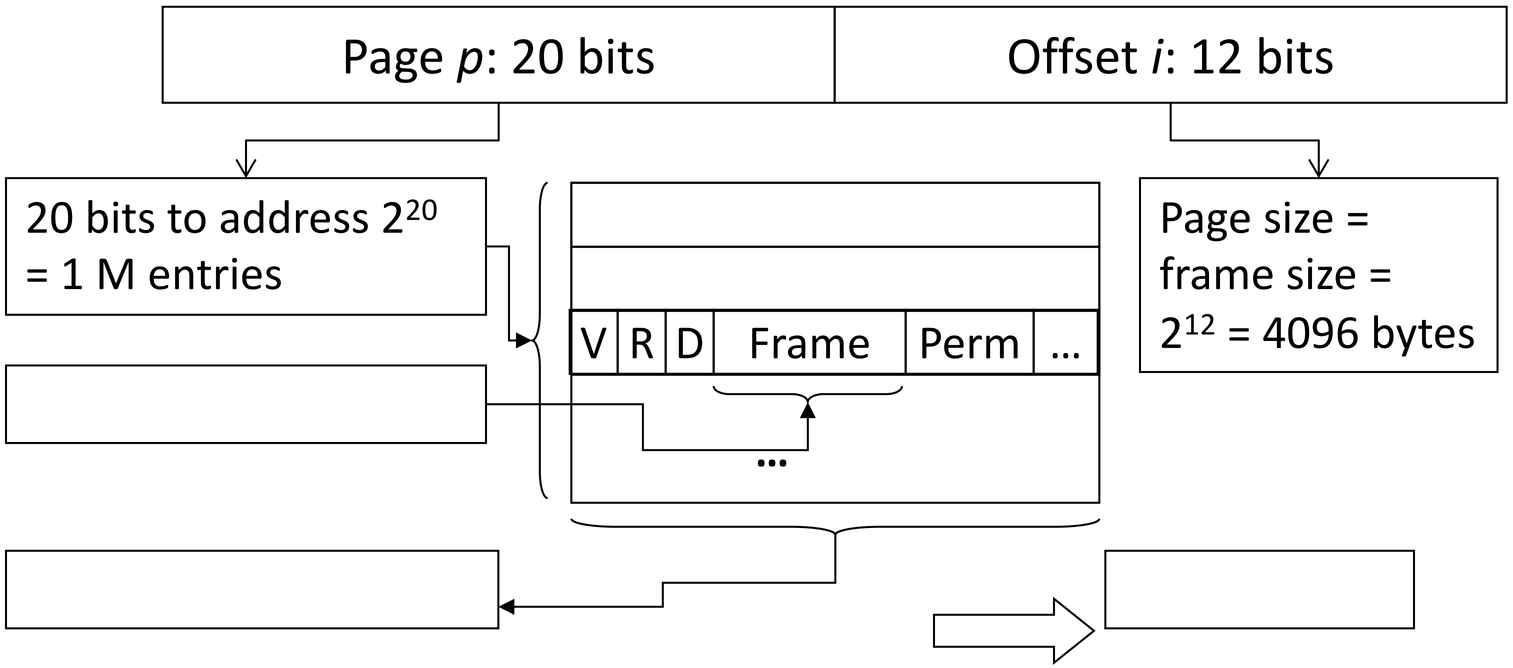
– Address partition: 20 bit page number, 12 bit offset

2^{10}	2^{20}	2^{30}	2^{40}
1KB	1MB	1GB	1TB

What will be the frame size, in bytes?

A. 2^{12}
B. 2^{20}
C. 2^{30}
D. 2^{32}

Example of Sizing the Page Table

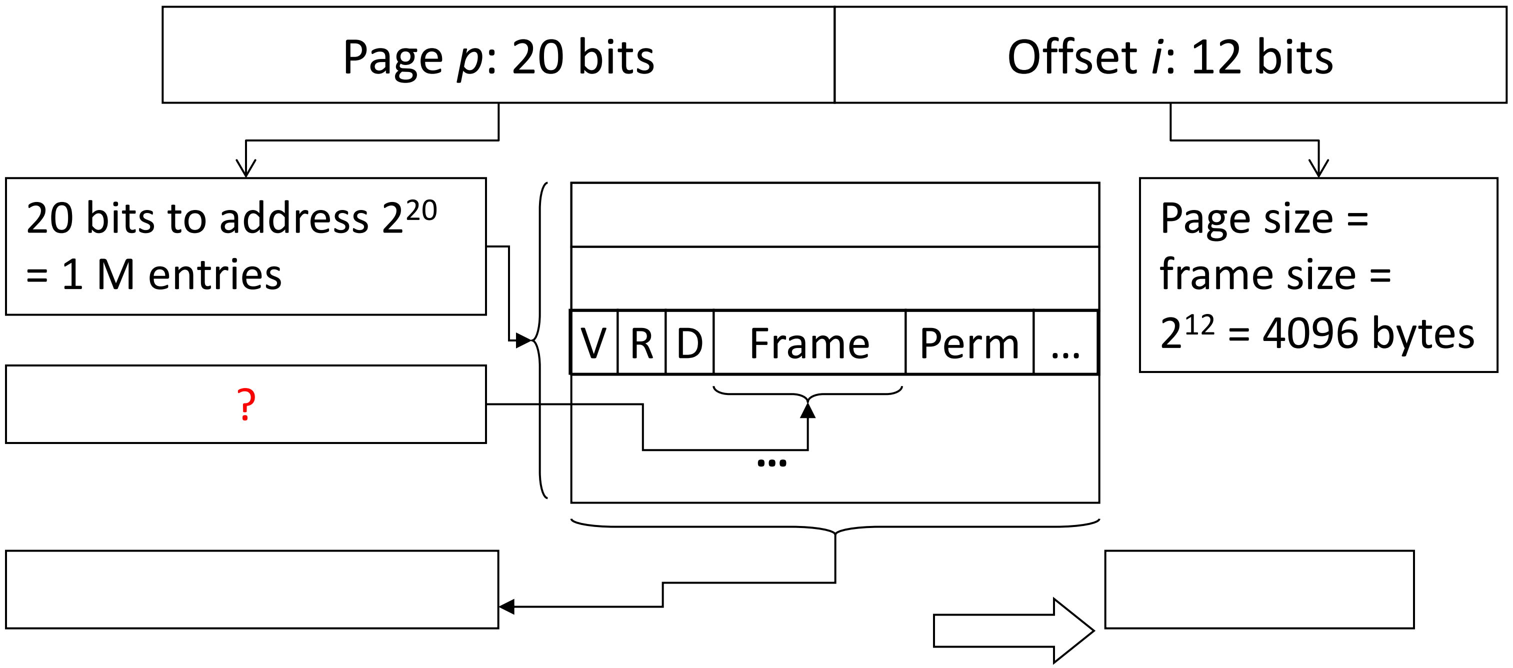


Given: 32 bit virtual addresses, 1 GB physical memory

– Address partition: 20 bit page number, 12 bit offset

2^{10}	2^{20}	2^{30}	2^{40}
1KB	1MB	1GB	1TB

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

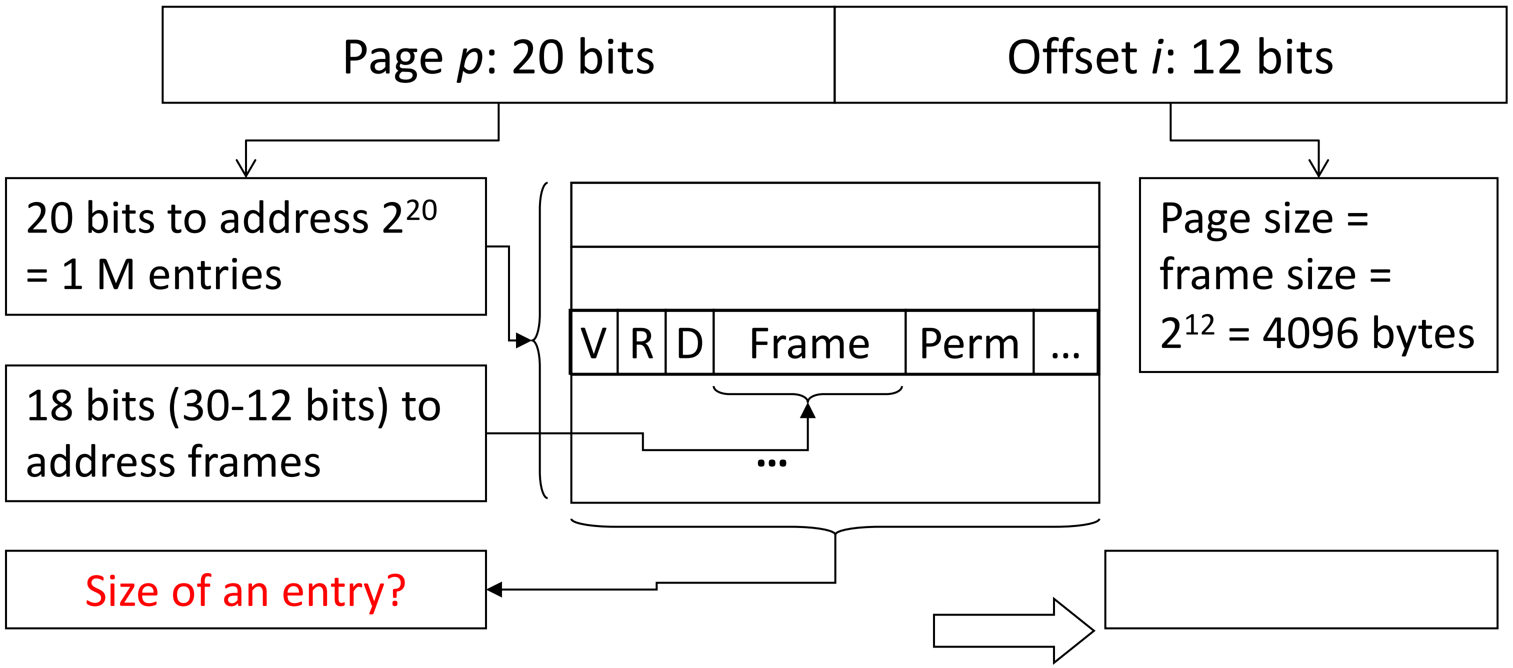


Given: 32 bit virtual addresses, 1 GB physical memory

– Address partition: 20 bit page number, 12 bit offset

2^{10}	2^{20}	2^{30}	2^{40}
1KB	1MB	1GB	1TB

Example of Sizing the Page Table

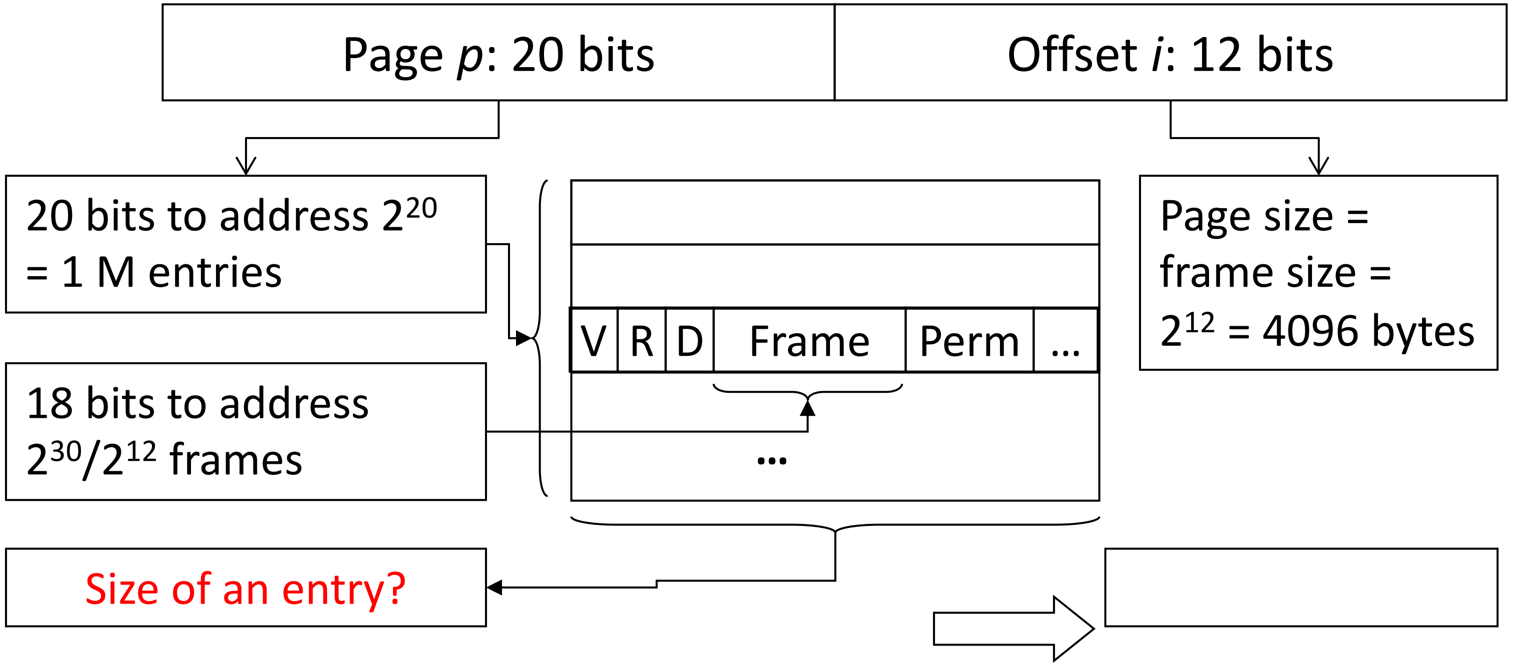


Given: 32 bit virtual addresses, 1 GB physical memory

– Address partition: 20 bit page number, 12 bit offset

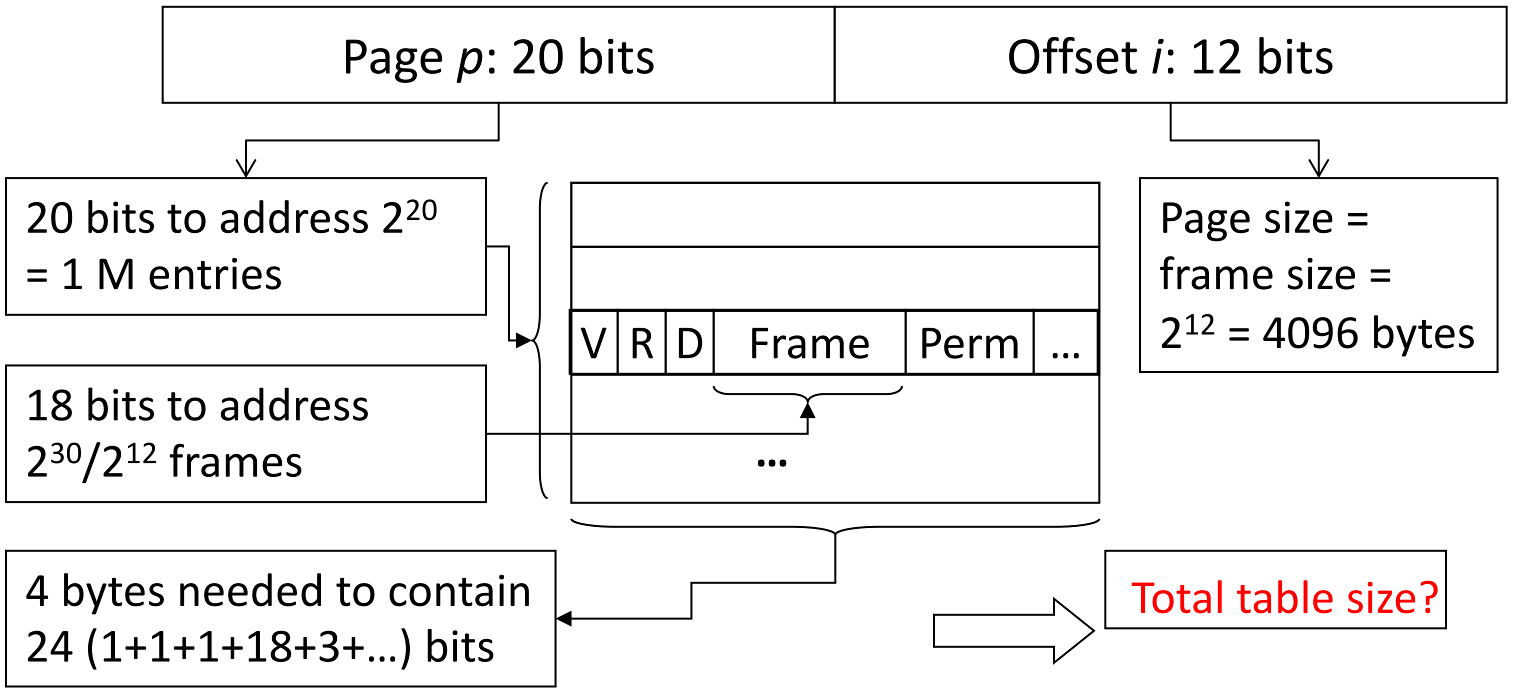
2^{10}	2^{20}	2^{30}	2^{40}
1KB	1MB	1GB	1TB

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



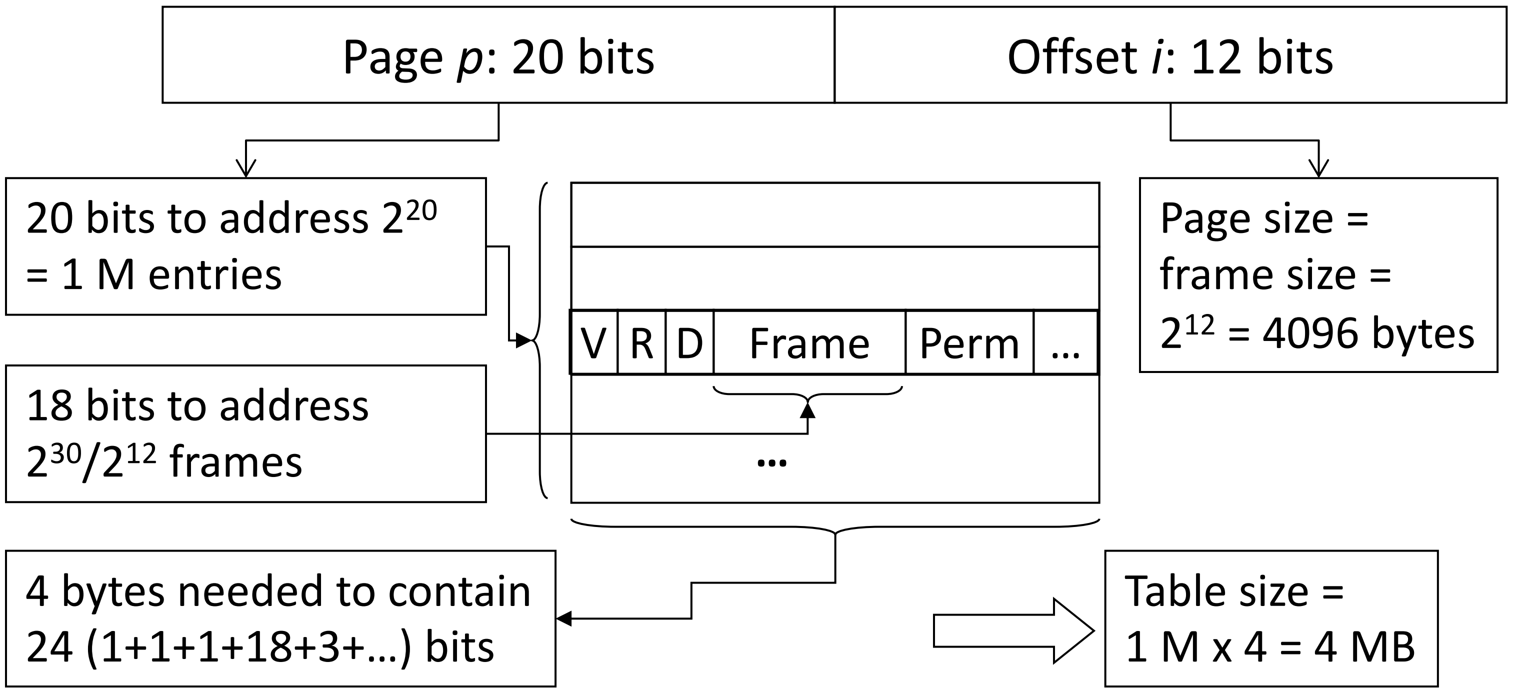
- Given: 32 bit virtual addresses, 1 GB physical memory
  - Address partition: 20 bit page number, 12 bit offset
- A: 1   B: 2   C: 4   D: 8

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

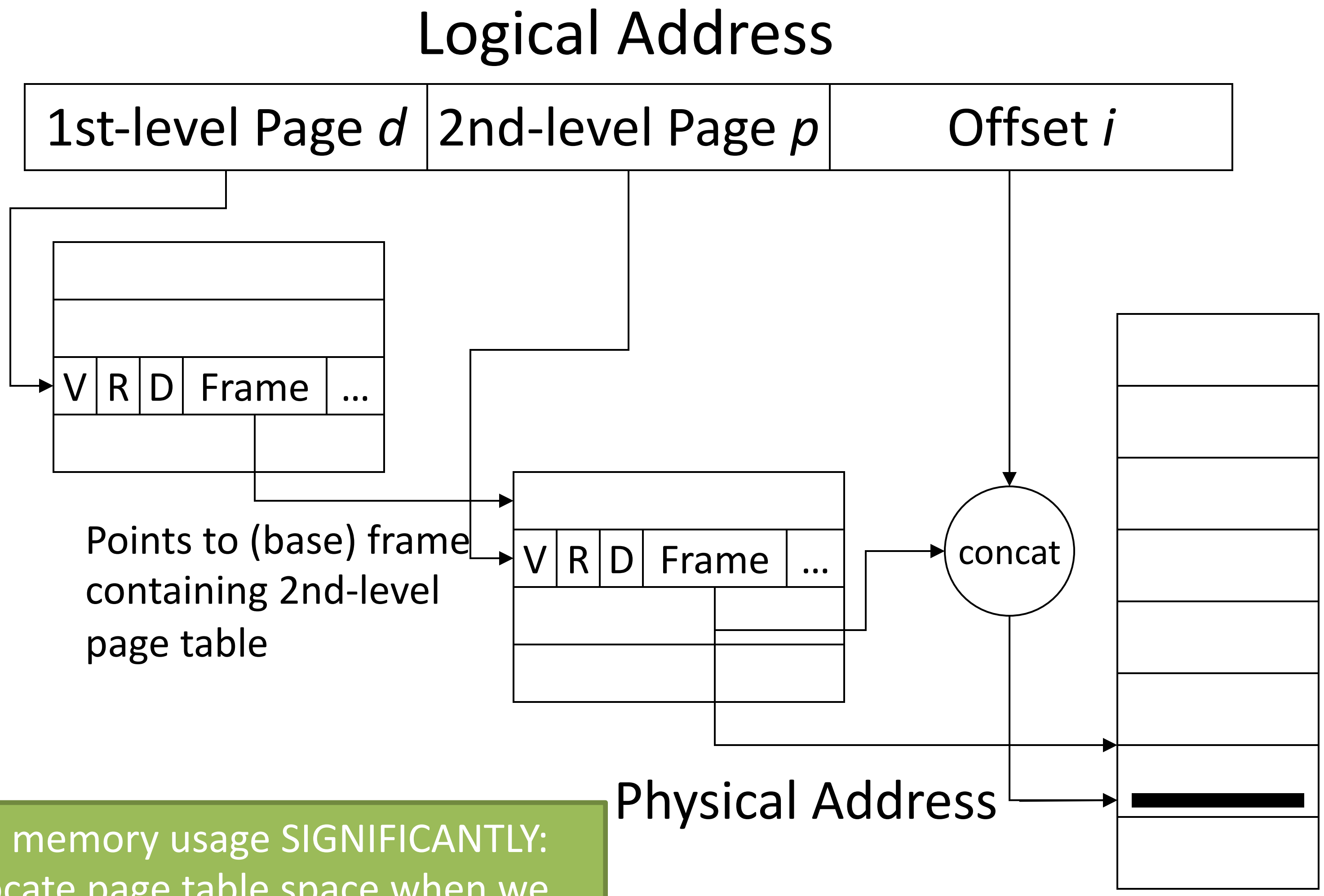
- Great, this page table idea solves a lot of those big problems we identified earlier, but…

1. We're going to need a ton of memory just for page tables…

2. 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.)

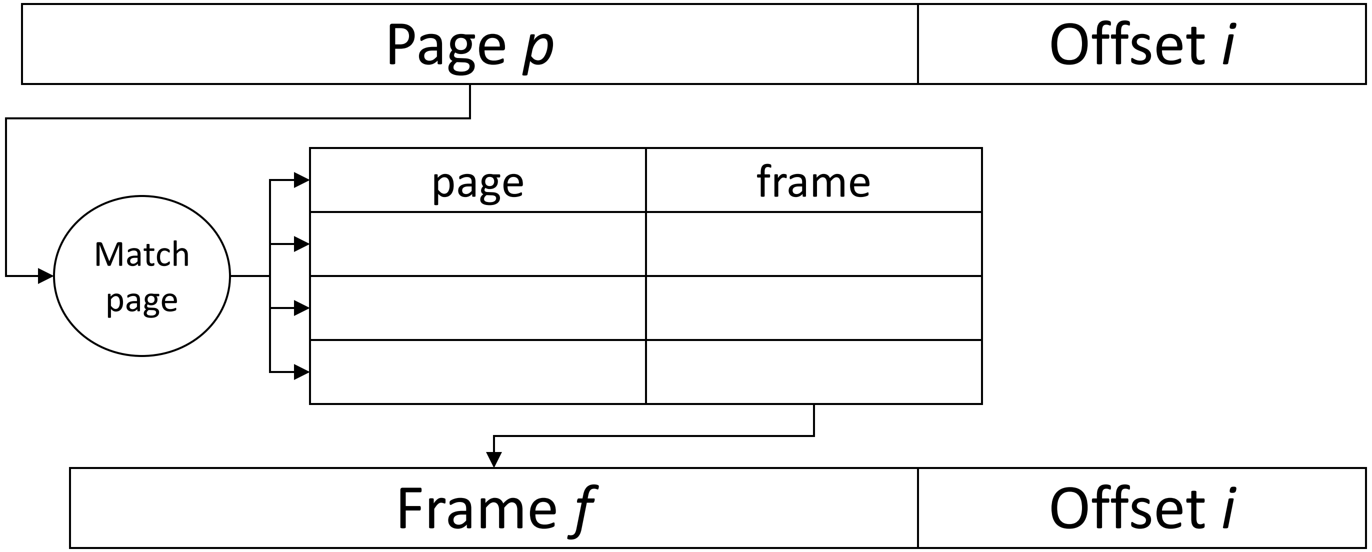


Reduces memory usage SIGNIFICANTLY: only allocate page table space when we need it. More memory accesses though…

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 special fully-associative cache for page translation) called the translation look-aside buffer (TLB)

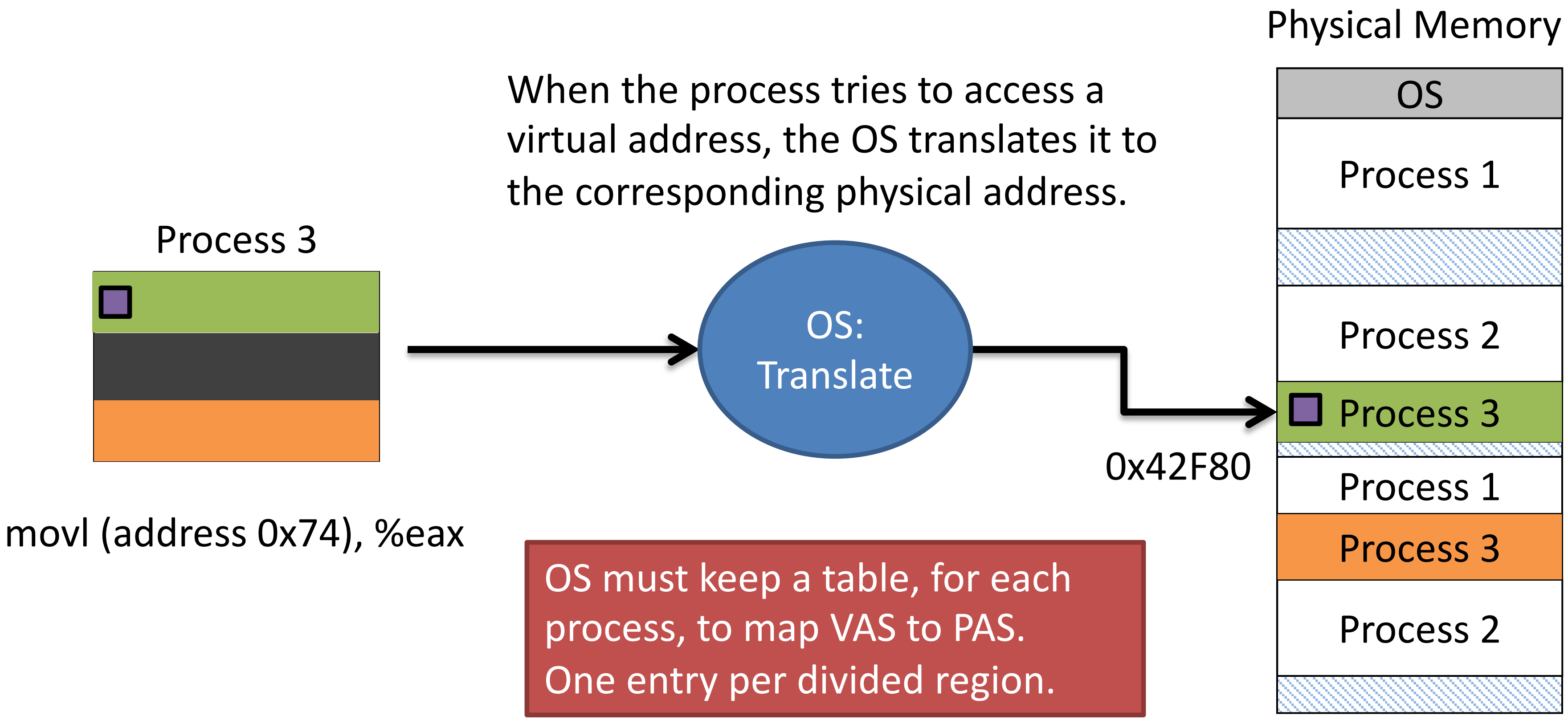
TLB: Translation Look-aside Buffer



- Fast memory keeps most recent translations
  - Fully associative hardware lookup
- If page matches, get frame number
  else wait for normal translation (in parallel)

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

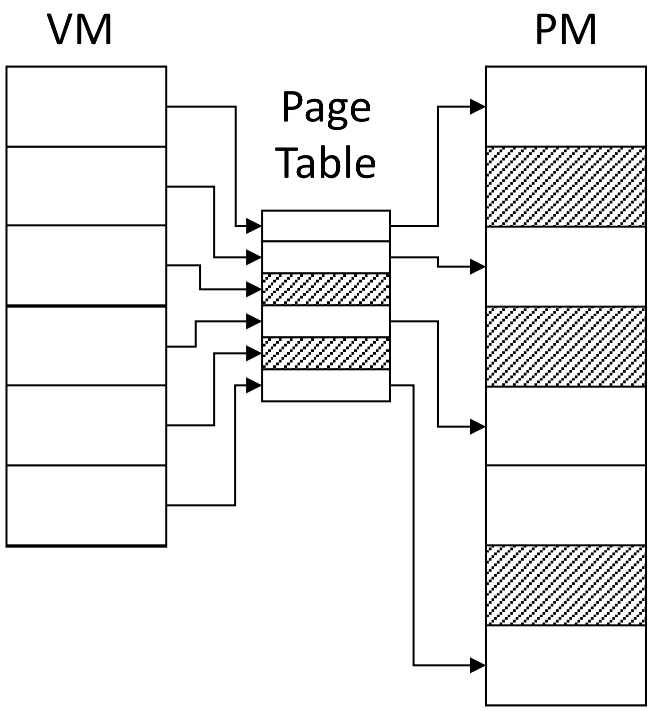
- How do we balance this tradeoff?

Locality to the rescue!

Virtual Memory Implications

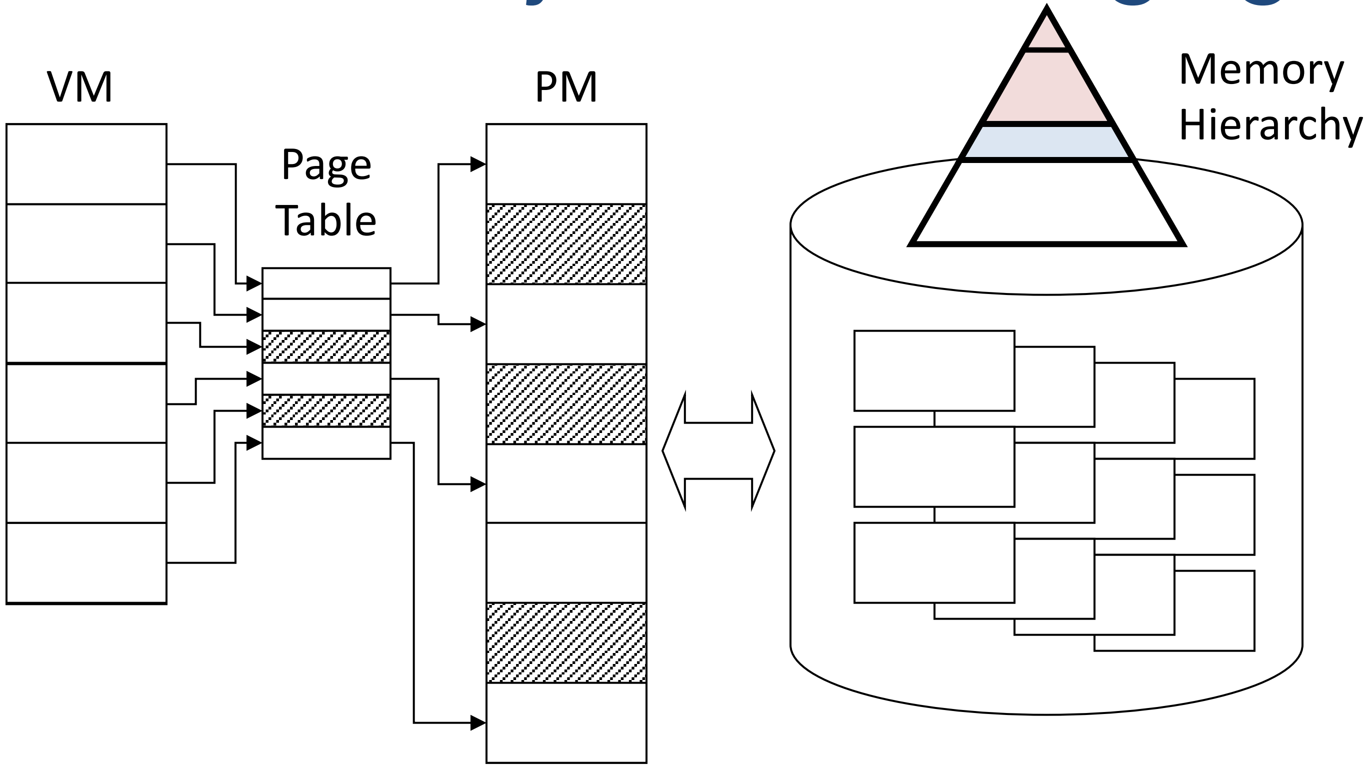
- Not all pieces need to be in memory
  - Need only piece being referenced
  - Other pieces can be on disk
  - Bring pieces in only when needed
- Illusion: there is much more memory
- What’s needed to support this idea?
  - A way to identify whether a piece is in memory
  - A way to bring in pieces (from 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 recently?
- 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 Fault

- A **page fault** occurs when a process tries to access a page, but the page table entry is invalid. That is, the page is not currently mapped to a physical frame.

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)

Adv: The process does not know that this is happening
Disadv: Execution slows down

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!

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 policy 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