Today

- Memory Management
  - Segmentation, Paging
- Improving memory performance
  - MMU
  - Translation Lookaside Buffer

Review

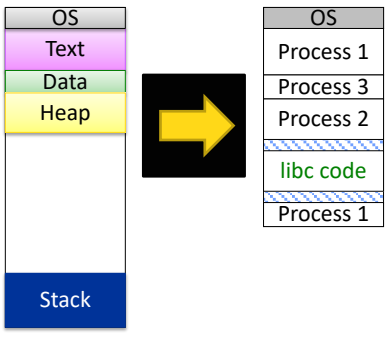
- What abstraction does virtual memory provide?
- What requirements do we have for the VM, from the various stakeholders?
- What is paging? Segmentation?
  - What are they used for?
  - Compare and contrast them
- How does the OS translate from the virtual address to the physical address?

Cody Watson, William & Mary
"An Introduction to Deep Learning and Its Applications"
Talk at 4 p.m.

The Big Picture: Virtual Memory

How can the OS build the abstraction of a private, potentially large address space for multiple running processes (all sharing memory) on top of a single, physical memory?

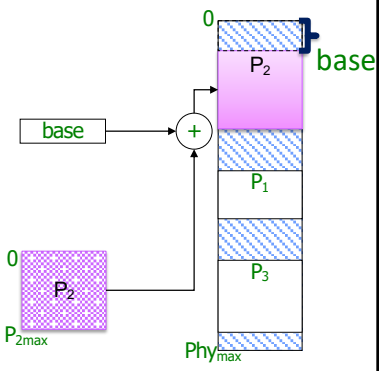
Review: Address Translation: Wish List



- 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

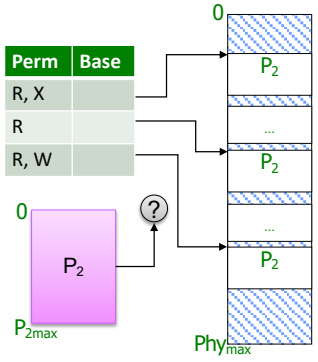
Review: (Unrealistic) Translation Example

- Process P_2's virtual addresses don't align with physical memory's addresses
- Consider: P_2 wants to access address 0x1000
- Determine offset from *physical address* 0 to start of P_2
  - store in *base*



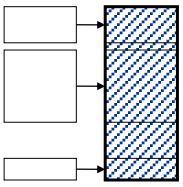
Review: Generalizing

- Problem: process may not fit in one contiguous region
- Solution: keep a table (one per 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 table entry we should use?

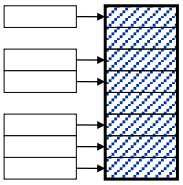


Review: Defining Regions

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



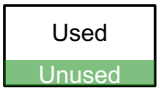
- Paging:
  - Partition address space and memory into pages
  - Pages are a constant, *fixed* size



Review: Fragmentation

Internal

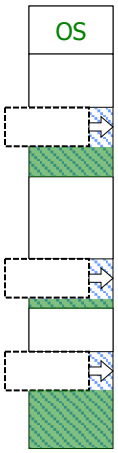
- Process asks for memory, doesn't use it all
- Possible reasons:
  - Process was wrong about needs
  - OS gave it more than it asked for
- ***internal***: within an allocation



Memory allocated to process

External

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



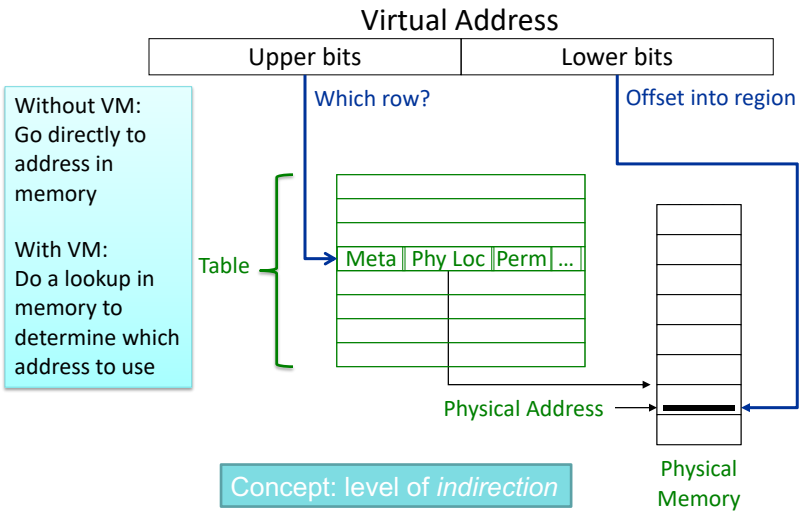
Review: Segmentation vs. Paging

- 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

Review: For **Both** Segmentation and Paging…

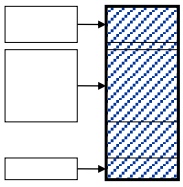
- Each process has a table to track memory address translations
- When a process attempts to read/write to memory:
  - use *high order* bits of virtual address to determine which row to look at in the table
  - use *low order* bits of virtual address to determine an offset within the physical region

Review: Performance Implications

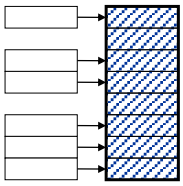


Defining Regions - Two Approaches

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

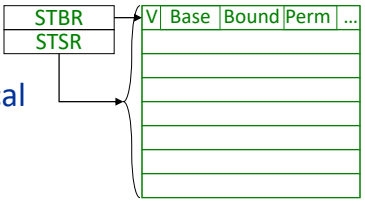


- Paging:
  - Partition address space and memory into pages
  - Pages are a constant, *fixed* size

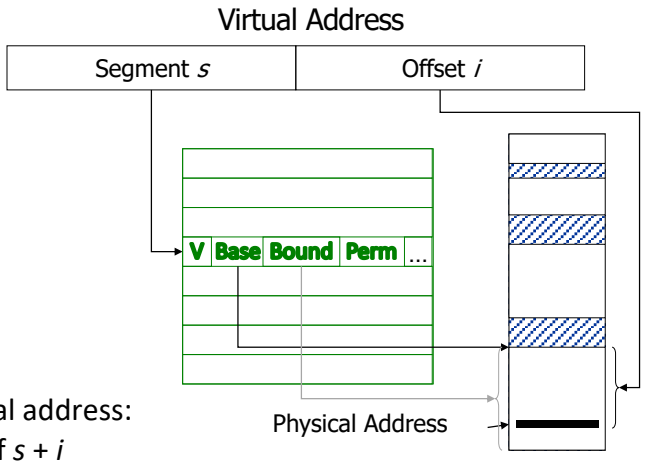


Segment Table

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

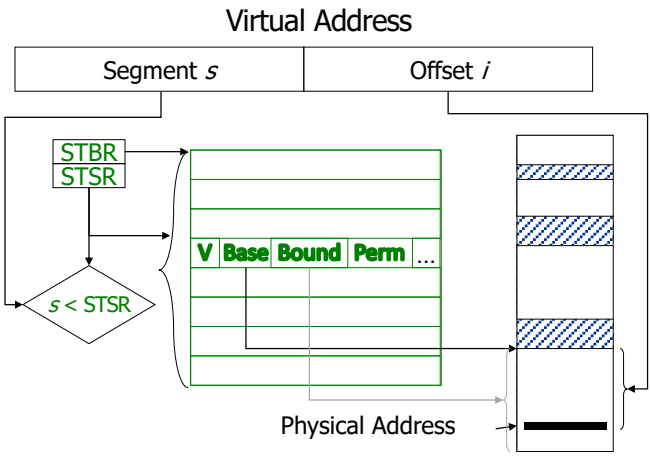


Segment Address Translation

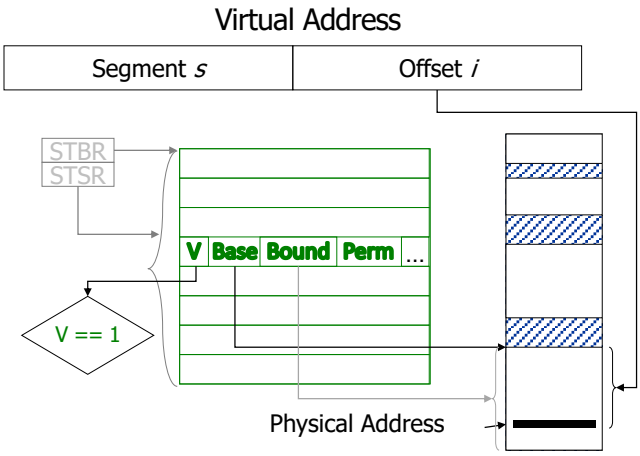


- Physical address: base of *s* + *i*

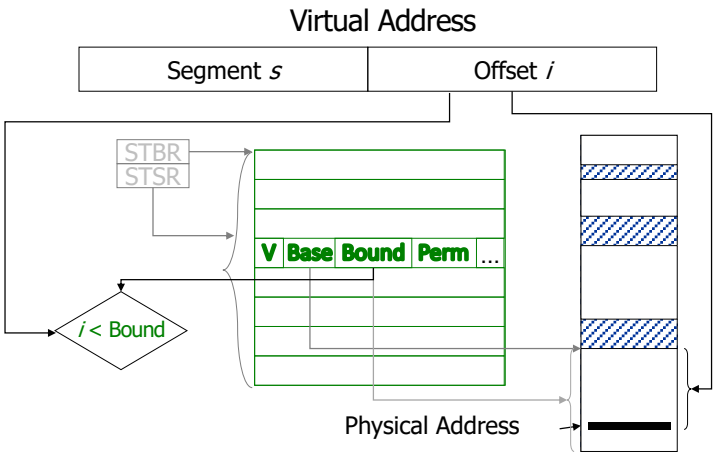
Check if Segment *s* is within Range



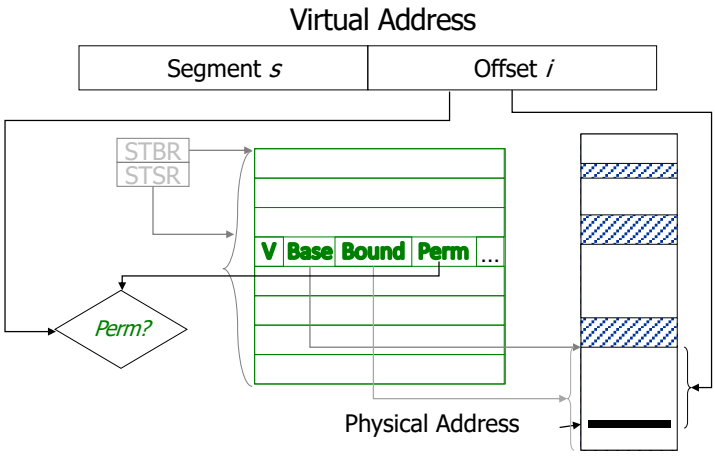
Check if Segment Entry *s* is Valid



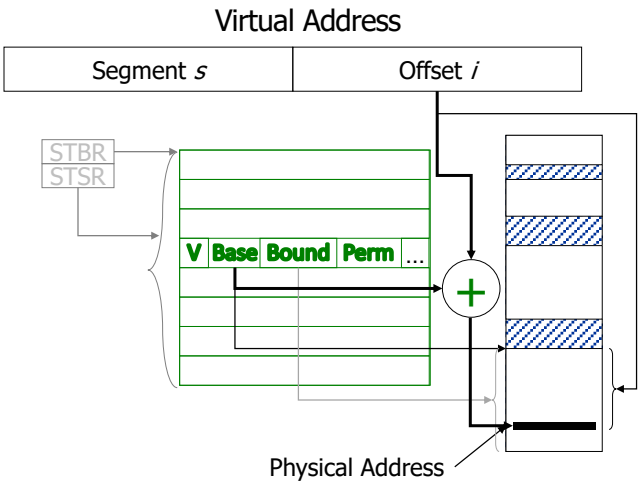
Check if Offset *i* is within Bounds



Check Permissions



Translate Address



Pros and Cons of Segmentation

Pros

Cons

Pros and Cons of Segmentation

Pros

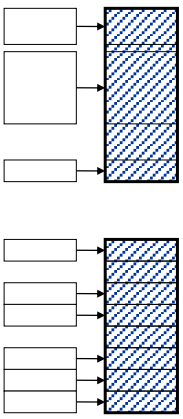
- Each segment can be
  - located independently
  - separately protected
  - grown/shrunk independently
- Small segment table size
  - ~256 Bytes → 1GB memory

Cons

- Variable-size allocation
  - Difficult to find holes in physical memory
  - External fragmentation

Defining Regions - Two Approaches

- Segmentation:
  - Partition address space and memory into logical segments
  - Segments have *varying* sizes
- Paging:
  - Partition address space and memory into pages
  - Pages are a constant, *fixed* size

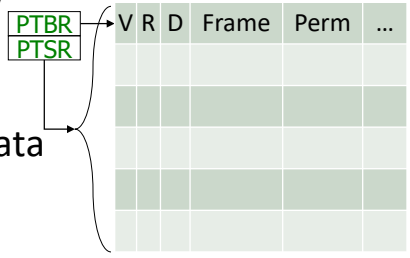


Paging Terminology

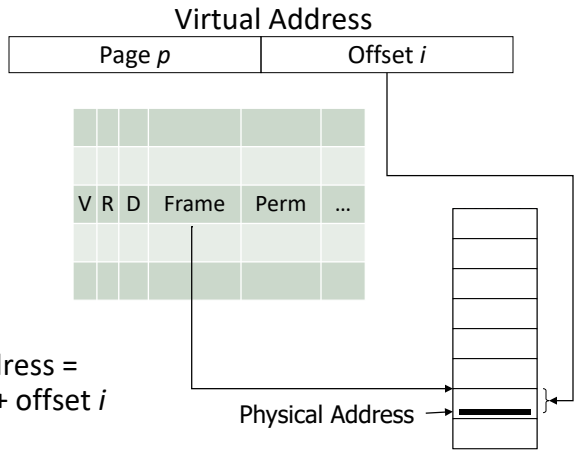
- For each process, the ***virtual*** address space is divided into *fixed-size **pages***
- For the system, the ***physical*** memory is divided into *fixed-size **frames***
- The size of a page is equal to that of a frame
  - Often 4 KB in practice
  - Some CPUs allow for small and large pages at the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

Page Table

- One table per process
- Table parameters in memory
  - Page table base register
  - Page table size register
- Table elements: Page metadata
  - V: valid bit
  - R: referenced bit
  - D: dirty bit
    - If page has been modified
  - Frame: location in physical memory
  - Perm: access permissions



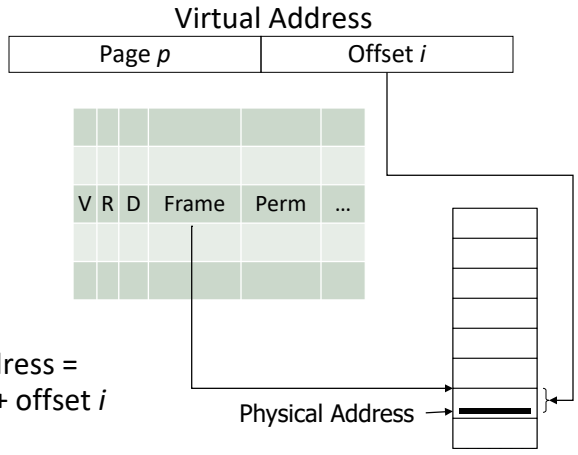
Paging Address Translation



- Physical address = frame of *p* + offset *i*

Why do we just need the frame number, rather than the location?

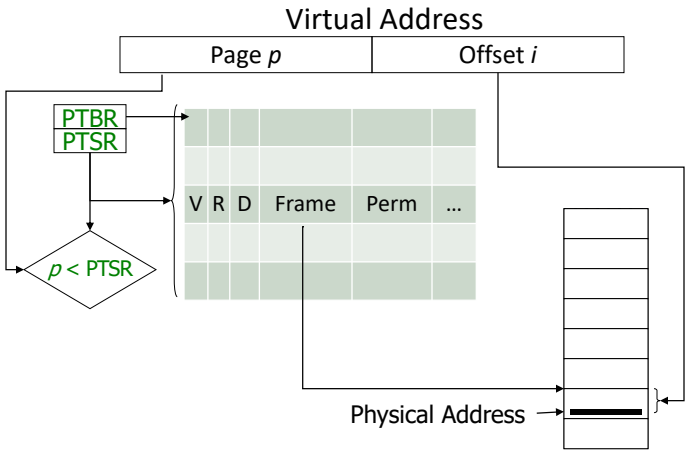
Paging Address Translation



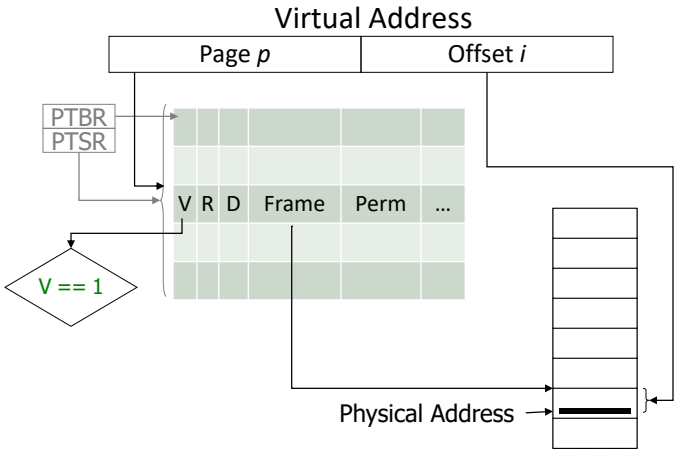
- Physical address = frame of *p* + offset *i*

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

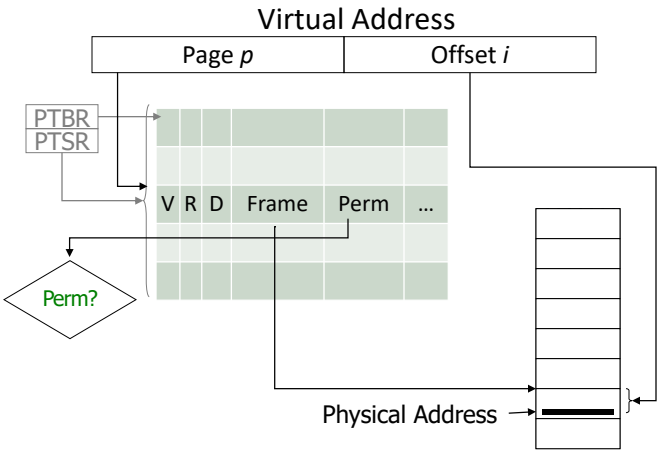
Check if Page *p* is Within Range



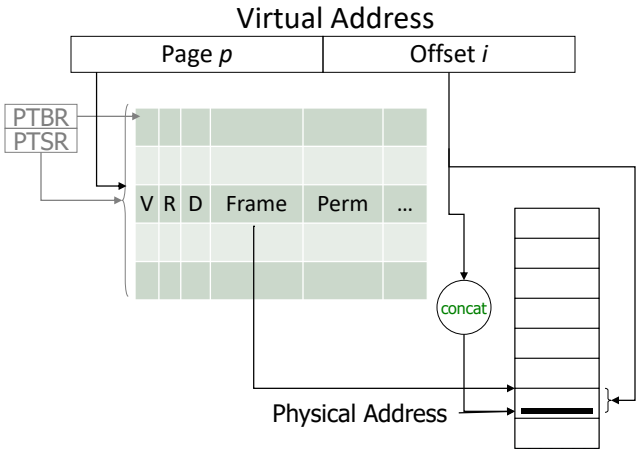
Check if Page Table Entry *p* is Valid



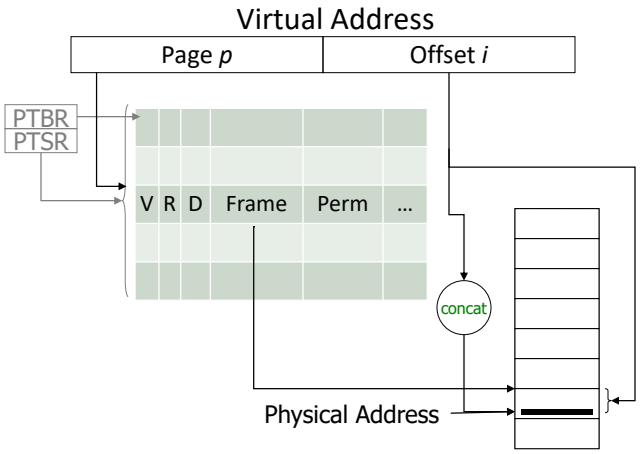
Check if Operation is Permitted



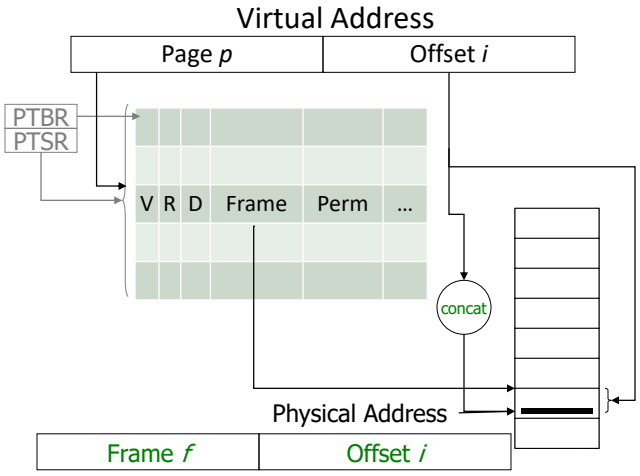
Translate Address



Physical Address by Concatenation



Physical Address by Concatenation



Pros and Cons of Paging

Pros

Cons

Pros and Cons of Paging

Pros

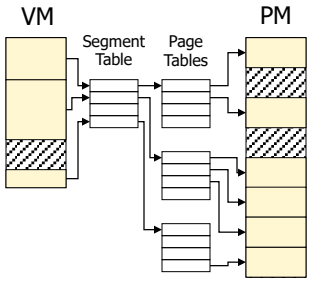
- Each page can be
  - located independently
  - separately protected
- Fixed-size pages and frames
  - No external fragmentation
  - No difficult placement decisions

Cons

- Large table size
  - ~4MB for 1GB of memory
    - That's for each process!
- *maybe* internal fragmentation

Hybrid Approach: Paged Segmentation – x86

- 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

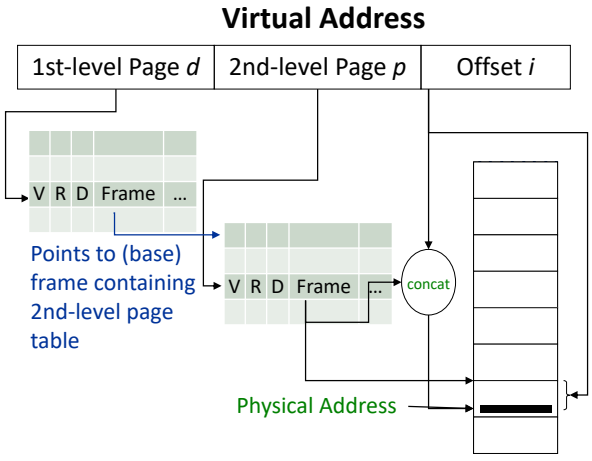
1. Page tables are way too big
   - Most processes don't need that many pages
   - Can't justify a huge table
2. Adding indirection hurts performance
   - Accessing memory to access memory…

Challenge: Large Page Tables

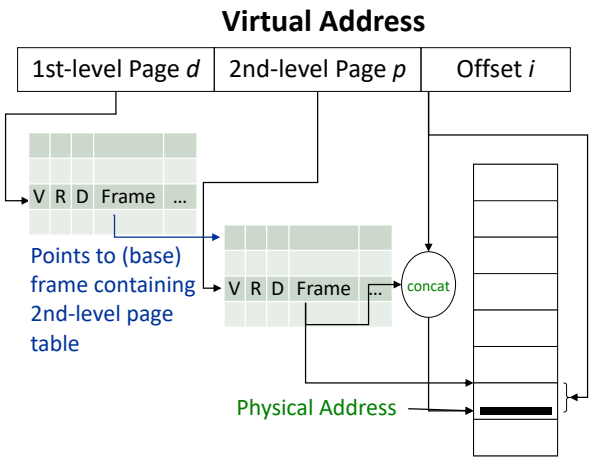
- Most processes don't need that many pages
  - Can't justify a huge table for every process
- What can we do so that our page table scales with the amount of memory we need?
  - What problem does this sound like?

Solution:
MORE indirection!

Multi-Level Page Tables



Multi-Level Page Tables

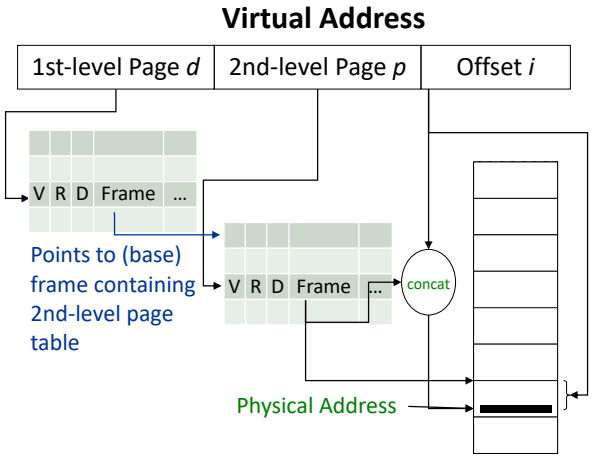


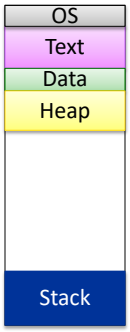
Insight: VAS is typically sparsely populated

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





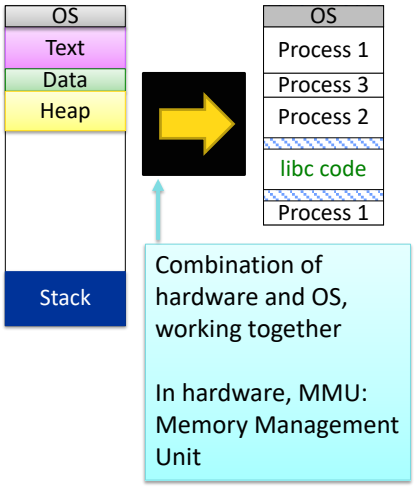
Multi-Level Page Tables

- 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

Challenge: Translation Cost

- Each application [logical] memory access now requires multiple memory accesses!
- Suppose a memory access takes 100 ns
  - 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

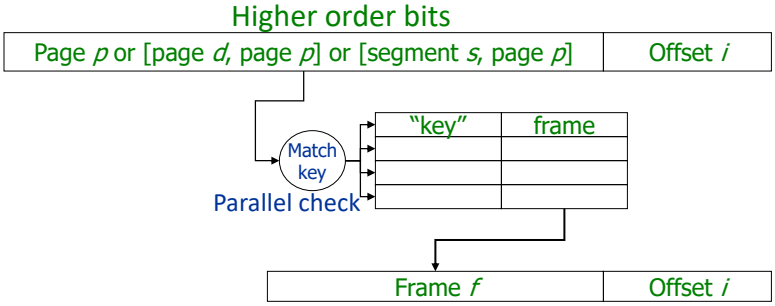
Memory Management Unit (MMU)



- 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

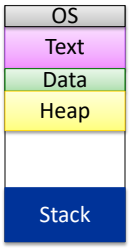
Translation Look-aside Buffer (TLB)

- Fast memory mapping cache inside MMU keeps most recent translations
  - If key matches, get frame number quickly
  - Otherwise, wait for normal translation
    - Add to TLB



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

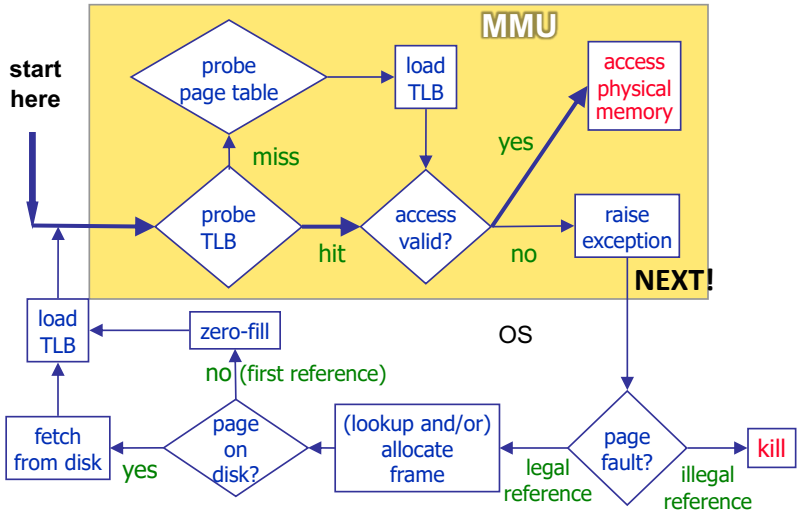
- Cost is determined by
  - Speed of memory: ~100 nsec
  - Speed of TLB: ~10 nsec
  - Hit ratio: fraction of memory references satisfied by TLB, ~95%
- Speed to access memory with address translation (2-level paging):
  - TLB miss: 300 nsec (200% slowdown)
  - TLB hit: 110 nsec (10% slowdown)
  - Average: 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

Virtual Addressing: Under the Hood



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

Looking Ahead

- Project 5 due Friday