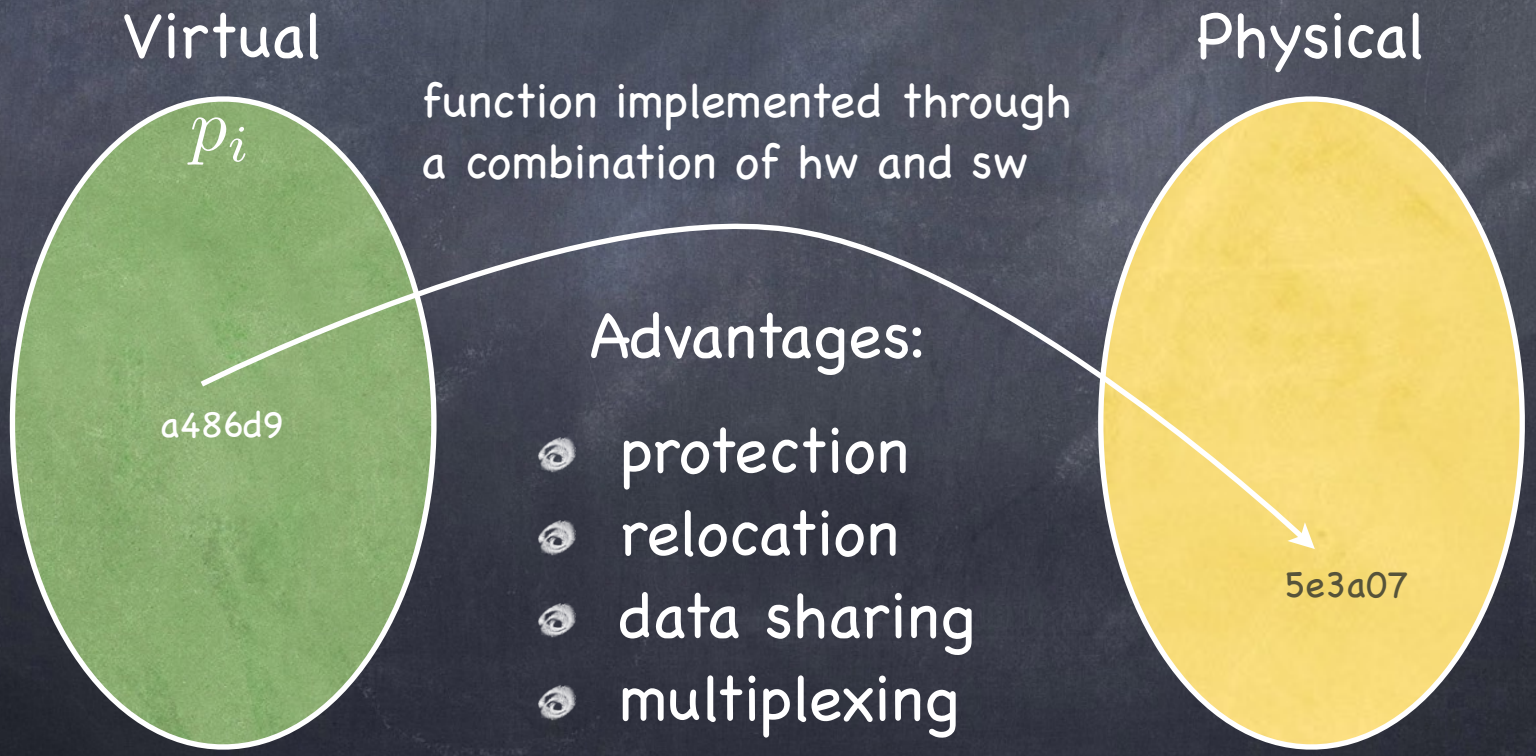
Lecture 14:
Memory Management

Paging, and Page Tables

Recall: Address Translation

- A function that maps $\langle pid, virtual\ address \rangle$ into a corresponding $physical\ address$



Recall: what we care about in address translation

- How to perform the mapping efficiently?
  - So that it can be represented concisely?
  - So that it can be computed quickly?
  - So that it makes efficient use of the limited physical memory?
  - So that multiple processes coexist in physical memory while guaranteeing isolation?
  - So that it decouples the size of the virtual and physical addresses?

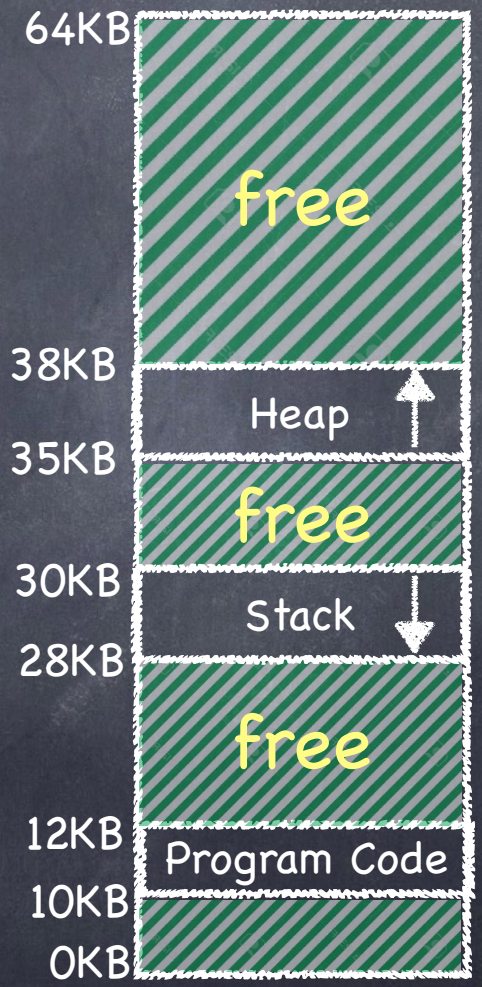
Recall: Technique 1: Base and bound

- Pros:
  - Space-efficient address translation
    - Only need 2 registers per process
  - Fast address translation
    - Simple operations
- Cons:
  - Wastes a lot of space (forces continuity)
  - Does not work if address space > physical memory

Recall: Technique 2: Segments + Base and bound

- Base & Bound registers to each segment
  - each segment is independently mapped to a set of contiguous addresses in physical memory
    - no need to map unused virtual addresses

Segment	Base	Bound
Code	10K	2K
Stack	28	2K
Heap	35K	3K



(not to scale)

Recall: Technique 2: Segments + Base and bound

- Pros: still space efficient and fast
  - segment table: store base and bound registers for the segment
    - stored in memory, at an address pointed to by a Segment Table Base Register (STBR)
    - process' STBR value stored in the PCB
    - 2 registers per segment (fairly space-efficient)
  - Address Translation:
    - first find the segment (using STBR),
    - then use base and bound to perform address translation

Recall: Technique 2: Segments + Base and bound

- Challenge?
  - Contiguous addresses for each segment
  - "Fitting" segments into physical memory
  - Many segments & processes, different sizes
- Many strategies to fit segment into free memory
  - First Fit: first big-enough hole
  - Next Fit: Like First Fit, but starting from where you left off
  - Best Fit: smallest big-enough hole
  - Worst Fit: largest big-enough hole

Recall: Technique 2: Segments + Base and bound

- Challenge?
  - Contiguous addresses for each segment
  - "Fitting" segments into physical memory
  - Many segments & processes, different sizes
- External fragmentation
- Can be avoided using compaction
  - Heavy-weight
  - Does not allow segments to grow

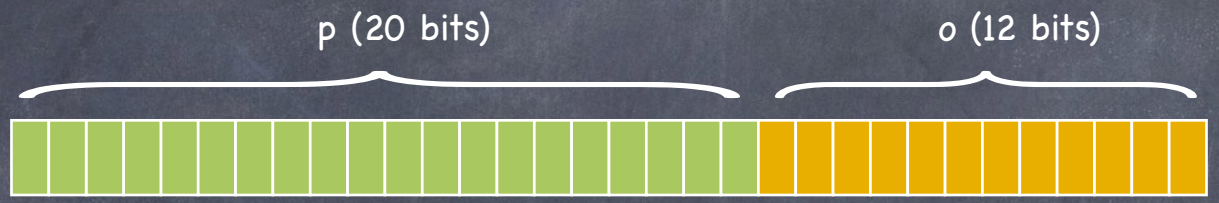
Recall: Technique 2: Segments + Base and bound

- Challenge?
  - Contiguous addresses for each segment
  - "Fitting" segments into physical memory
  - Many segments & processes, different sizes
- External fragmentation
- Can be avoided using compaction
  - Heavy-weight
  - Does not allow segments to grow

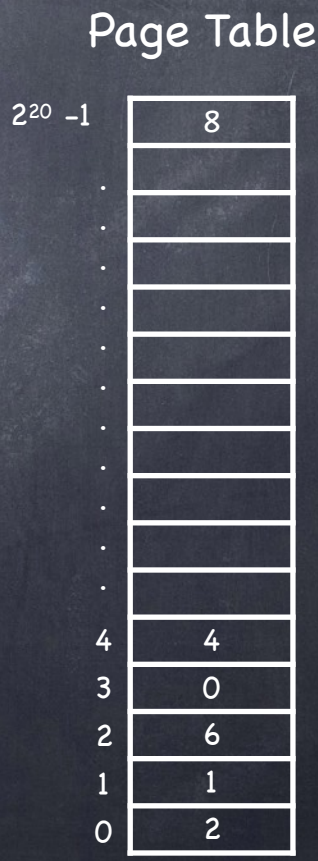
Recall: Paging

- Allocate VA & PA memory in chunks of the same, fixed size (pages and frames, respectively)
- Adjacent pages in VA need not map to contiguous frames in PA!
  - free frames can be tracked using a simple bitmap
    - 001111100111101111000 0 one bit/frame
  - no more external fragmentation!
  - possible internal fragmentation
    - when memory needs are not a multiple of a page
    - typical size of page/frame: 4KB to 16KB

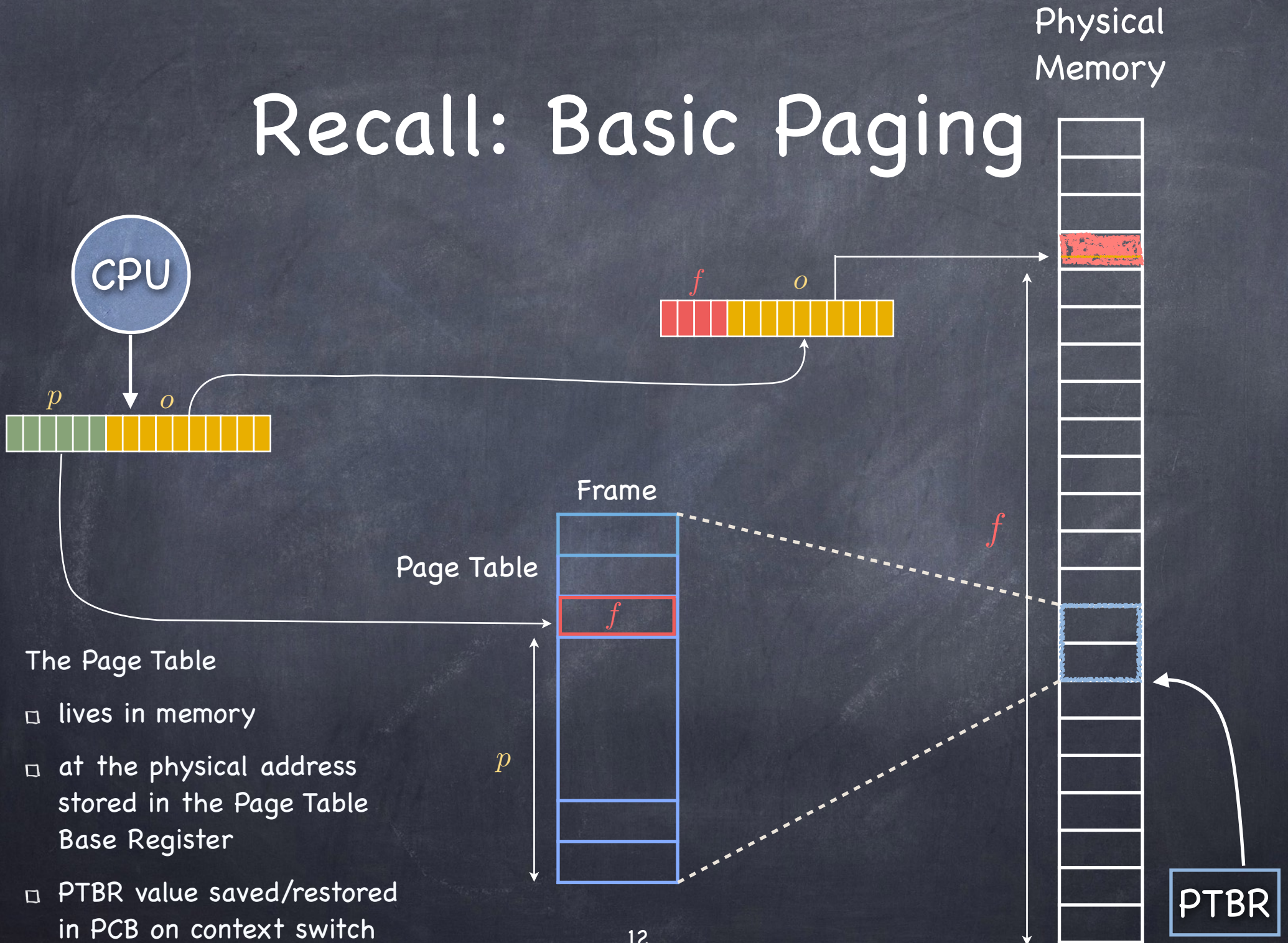
Recall: Paging & Page Tables



- To access a byte
  - extract page number
  - map that page number into a frame number using a page table
    - Note: not all pages may be mapped to frames
  - extract offset
  - access byte at offset in frame



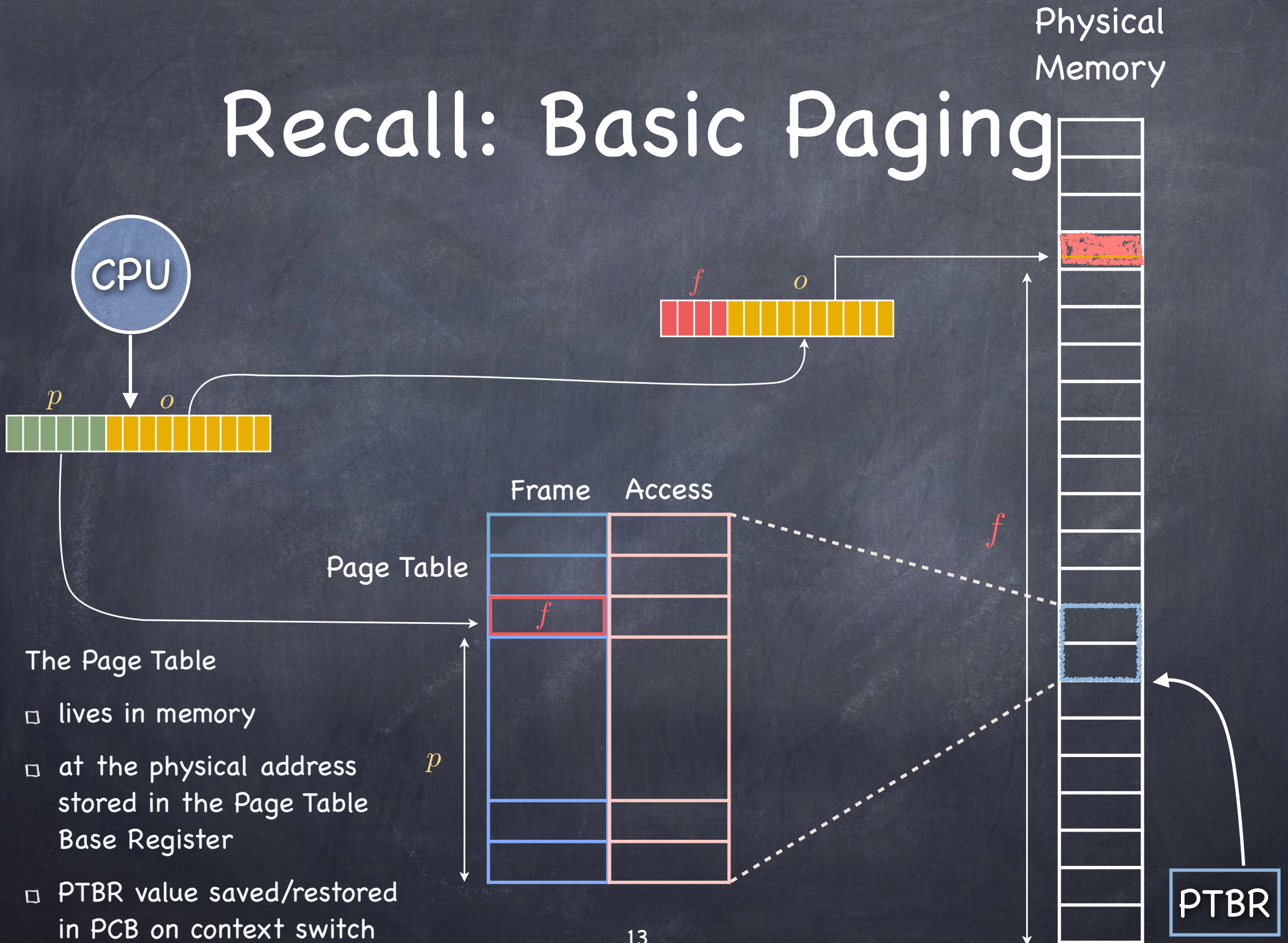
Recall: Basic Paging



The Page Table

- lives in memory
- at the physical address stored in the Page Table Base Register
- PTBR value saved/restored in PCB on context switch

Recall: Basic Paging

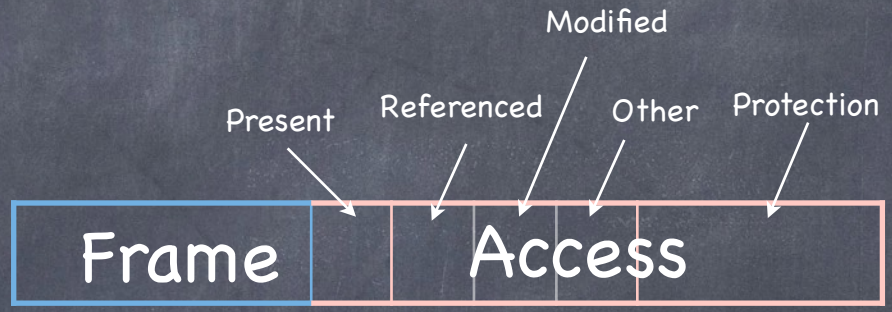


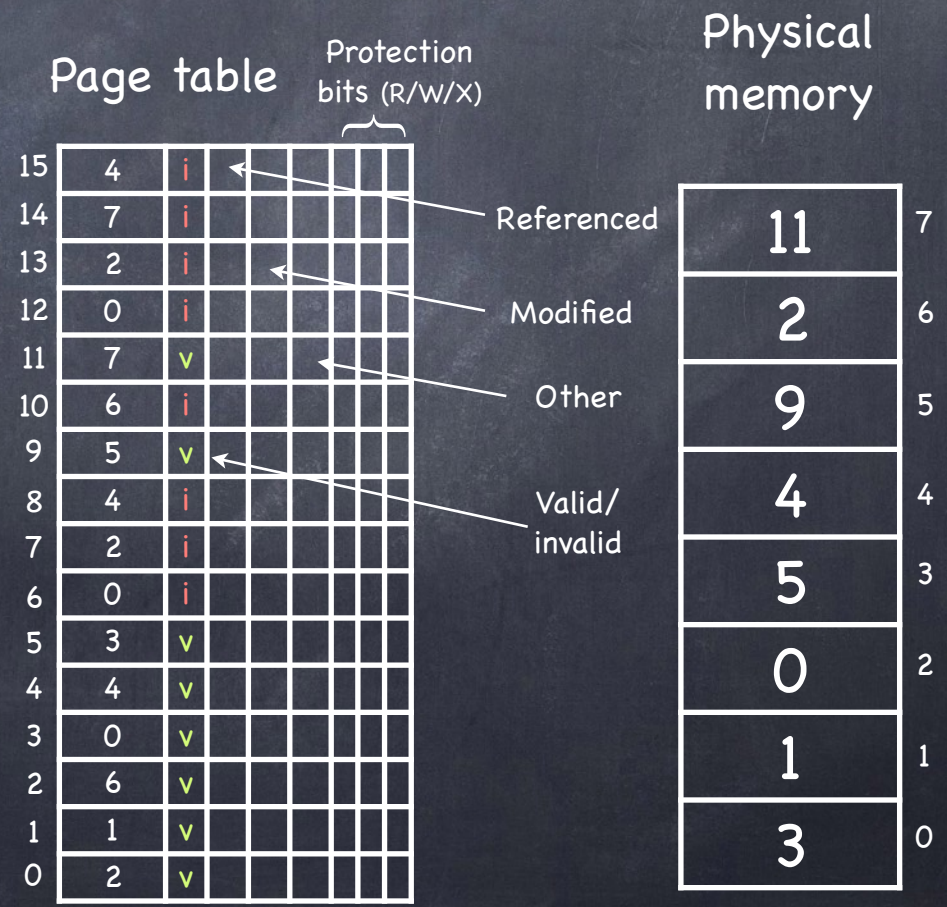
The Page Table

- lives in memory
- at the physical address stored in the Page Table Base Register
- PTBR value saved/restored in PCB on context switch

Recall: Page Table Entries

- Frame number
- Valid/Invalid (Present) bit
  - Set if entry stores a valid mapping. If not, and accessed, page fault
- Referenced bit
  - Set if page has been referenced
- Modified bit
  - Set if page has been modified
- Protection bits (R/W/X)





Questions?

Basic goals in paging

- **Minimize Storage overhead**
  - data structure overhead (the Page Table itself)
  - fragmentation
    - How large should a page be?
- **Fast Address translation**
  - We need "fast" lookups on page table
- **Efficient sharing of physical memory**
  - By multiple processes

Paging—first attempt

- Divide virtual address space into fixed-sized pages (e.g., 4KB)
- Page Table maps each page to a frame
- Storage overheads:
  - Number of entries = size of virtual address space / page size
  - Size of entry
    - enough bits to identify physical page ($\log_2$ (PA_Size / frame size))
    - should include control bits (present, dirty, referenced, etc)
    - usually word or byte aligned
  - 32-bit virtual address space, 4GB physical memory, 4KB pages
    - ($2^{32}/2^{12}$ entries * sizeofEntry)
    - sizeofEntry = 32 bits = 4 bytes
      - $\log_2$ (PA_Size/frame size)+7 control bits + byte aligned = $\log_2$ ($2^{32}/2^{12}$) + 7 +? = 32
    - 4MB

Paging—first attempt

- Divide virtual address space into fixed-sized pages (e.g., 4KB)
- Page Table maps each page to a frame
- Storage overheads:
  - 32-bit virtual address space, 4GB physical memory, 4KB pages
    - ($2^{32}/2^{12}$ entries * sizeofEntry)
    - sizeofEntry = 32 bits = 4 bytes
      - $\log_2$ (PA_Size/frame size)+7 control bits + byte aligned = $\log_2$ ($2^{32}/2^{12}$) + 7 +? = 32
    - 4MB
  - 64-bit virtual address space, 4GB physical memory, 4KB pages
    - ($2^{64}/2^{12}$ entries * sizeofEntry)
    - sizeofEntry = 32 bits = 4 bytes
      - $\log_2$ (PA_Size/frame size)+7 control bits + byte aligned = $\log_2$ ($2^{32}/2^{12}$) + 7 +? = 32
    - $4*2^{52}$ bytes >> 64GB

Paging—first attempt

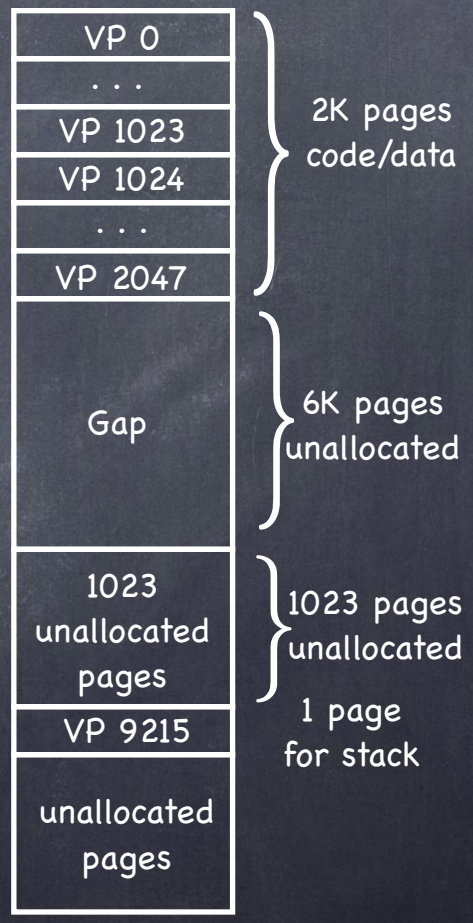
- Divide virtual address space into fixed-sized pages (e.g., 4KB)
- Page Table maps each page to a frame
- Space overhead
  - With a 64-bit address space, size of page table can be huge
  - Insight: page table size dependent on virtual address space
    - wrong design
    - page table size should depend on physical address space
- Time overhead
  - Accessing data now requires two memory accesses
    - One to access page table (no longer fits in cache)
    - Another one to find mapped frame

Reducing the Storage Overhead of Page Tables

- Size of the page table with 64-bit virtual addresses and 4KB page sizes is large
- Good news
  - most of the virtual address space is unused
- Use a better data structure to express the Page Table
  - a tree!

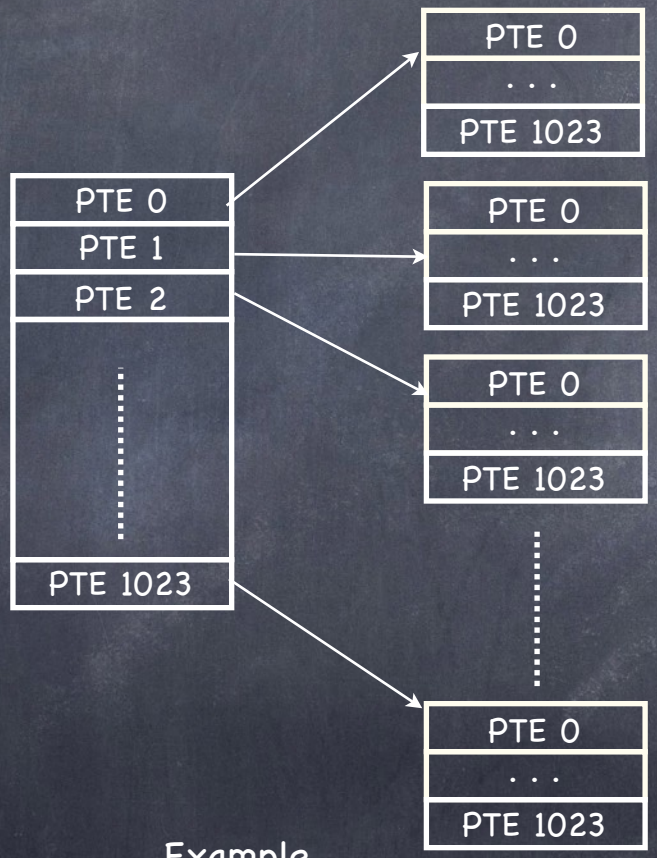
Example

- 32 bit address space
- 4KB pages
- 4 bytes PTE



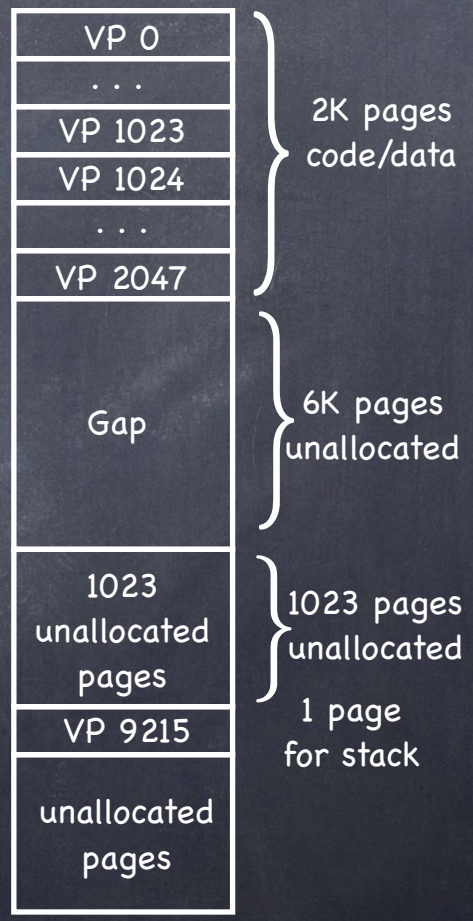
Reducing the Storage Overhead of Page Tables

- Size of the page table with 64-bit virtual addresses and 4KB page sizes is large
- Good news
  - most of the virtual address space is unused
- Use a better data structure to express the Page Table
  - a tree!



Example

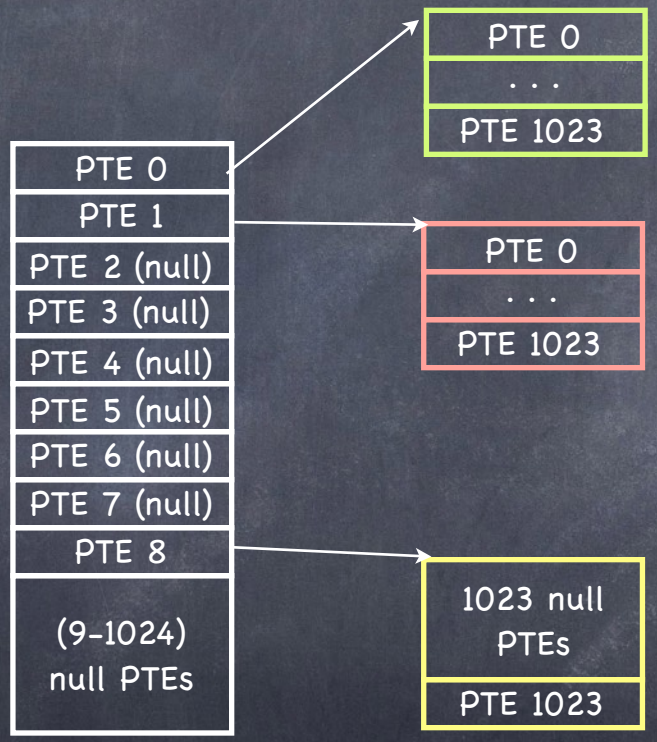
- 32 bit address space
- 4Kb pages
- 4 bytes PTE



Page Table

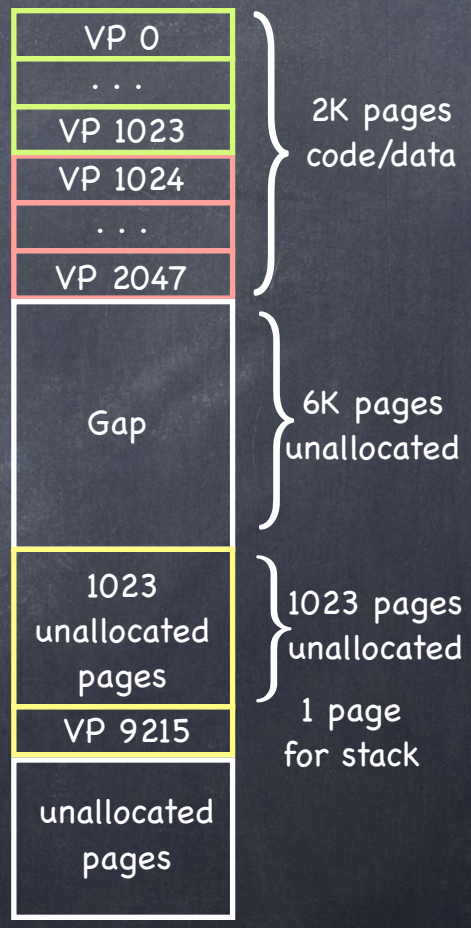
Reducing the Storage Overhead of Page Tables

- Size of the page table with 64-bit virtual addresses and 4KB page sizes is large
- Good news
  - most of the virtual address space is unused
- Use a better data structure to express the Page Table
  - a tree!



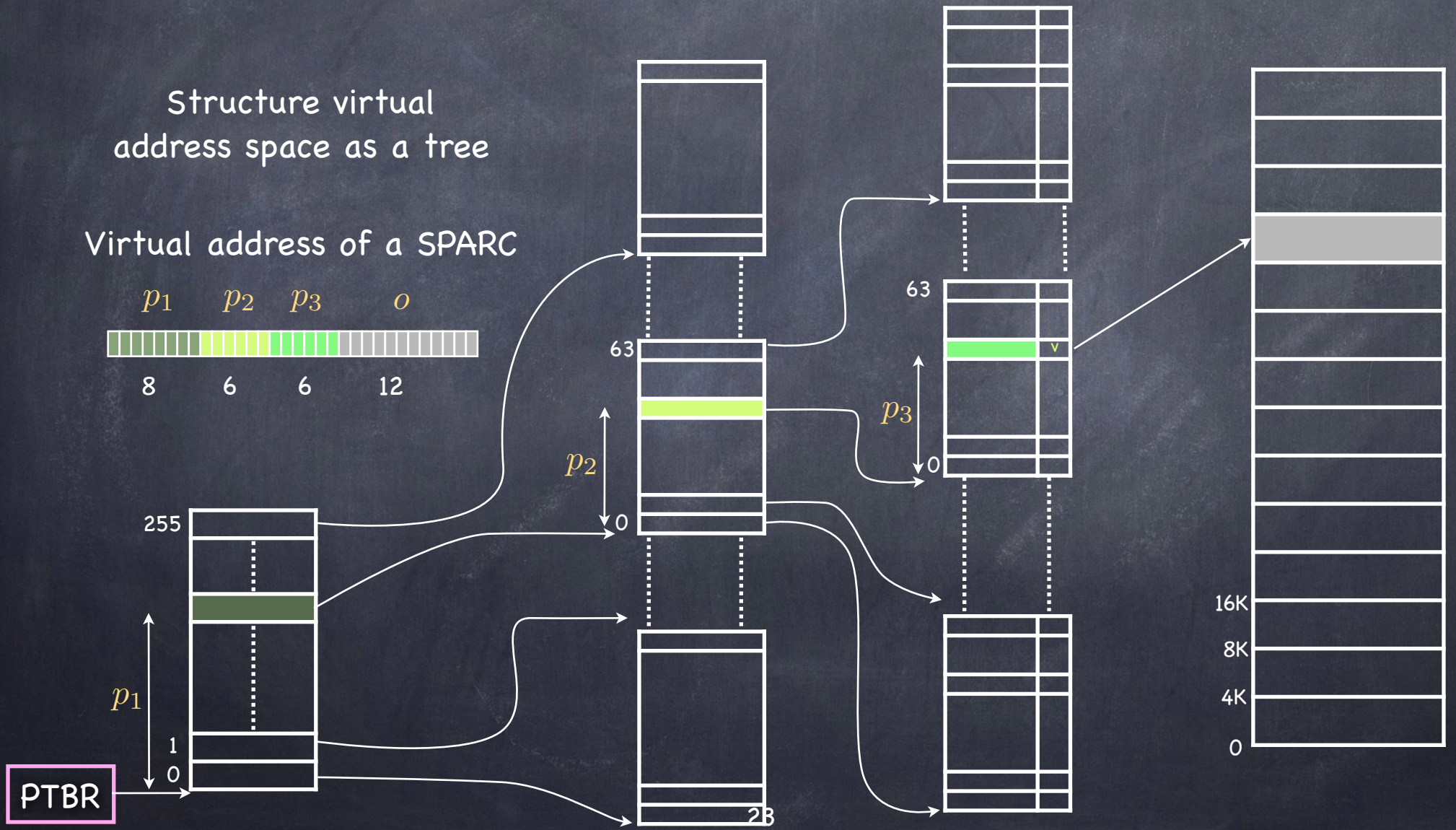
Example

- 32 bit address space
- 4Kb pages
- 4 bytes PTE

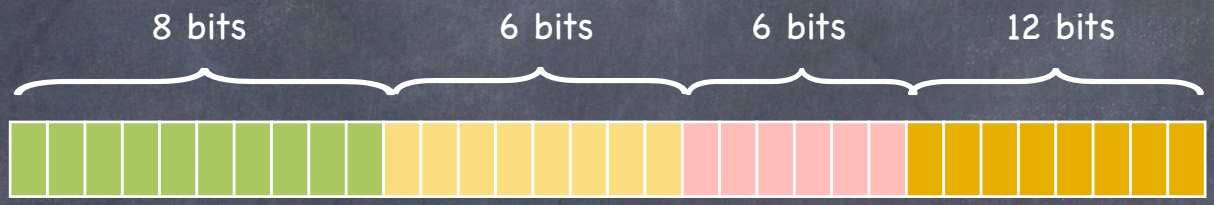


Page Table

Multi-level Paging

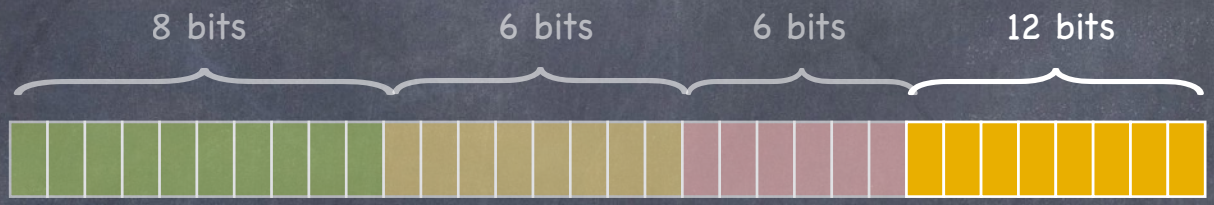


32-bit, 4GB, Example



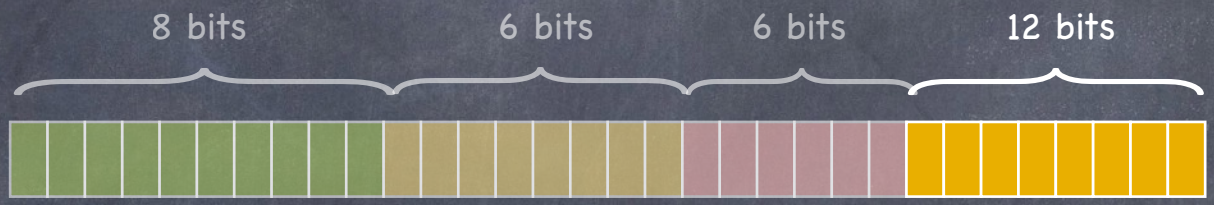
- What is the page size?

32-bit, 4GB, Example



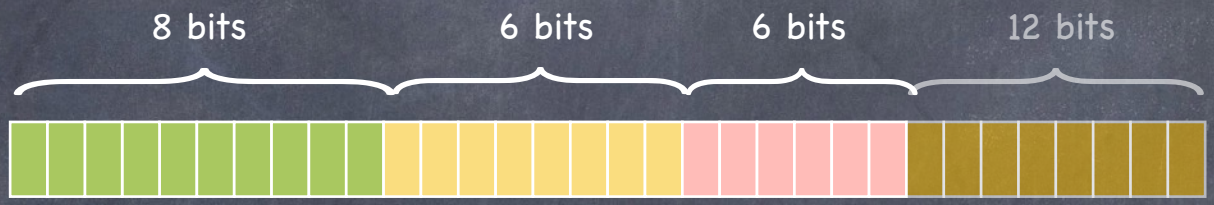
- What is the page size? Page size is 4KB (2^{12})
- What is the Page Table size for a process that uses contiguous 4KB of its VAS starting at address 0? [Assume each PTE is 4 bytes]
  - if we used a linear representation of the page table:

32-bit, 4GB, Example



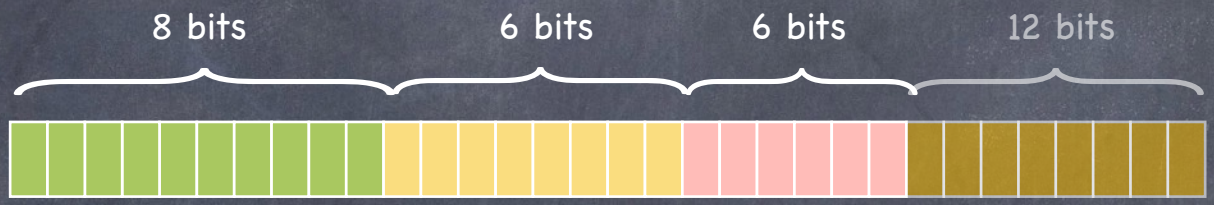
- What is the page size? Page size is 4KB (2^{12})
- What is the Page Table size for a process that uses contiguous 4KB of its VAS starting at address 0? [Assume each PTE is 4 bytes]
  - if we used a linear representation of the page table:
    - Page Table has 2^{20} entries

32-bit, 4GB, Example

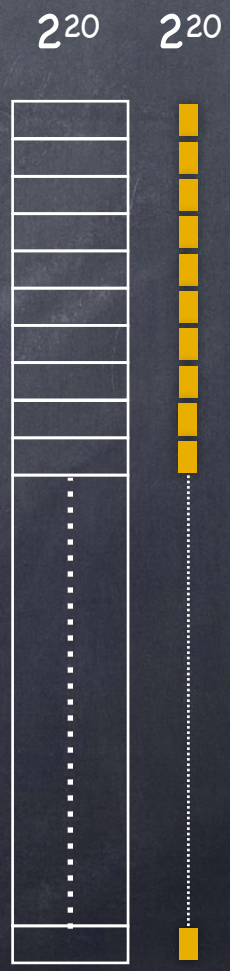


- What is the page size? Page size is 4KB (2^{12})
- What is the Page Table size for a process that uses contiguous 4KB of its VAS starting at address 0? [Assume each PTE is 4 bytes]
  - if we used a linear representation of the page table:
    - Page Table has 2^{20} entries
    - PT Size: 2^{20} x 4 bytes = 2^{22} bytes = 4MB

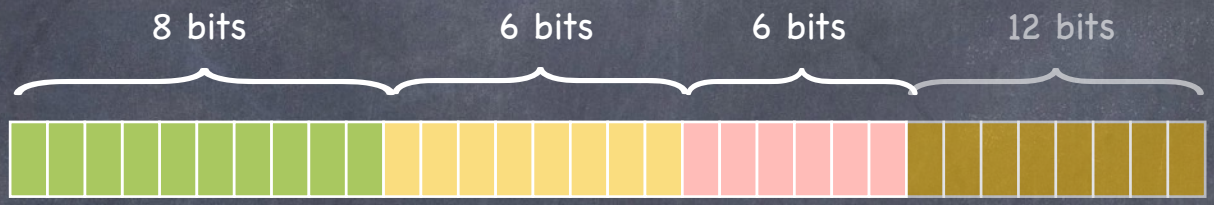
32-bit, 4GB, Example



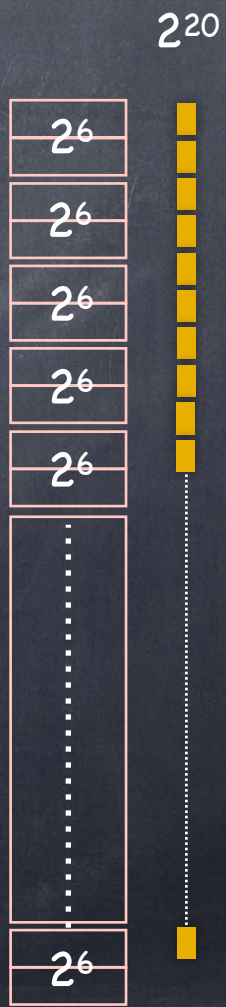
- What is we use a tree?
  - We still need to account for 2^{20} pages...
  - ...but we are going to partition the PT in a sequence of chunks, each with 2^6 entries



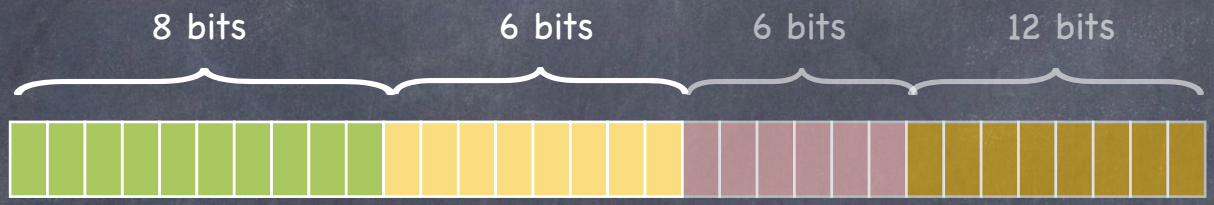
32-bit, 4GB, Example



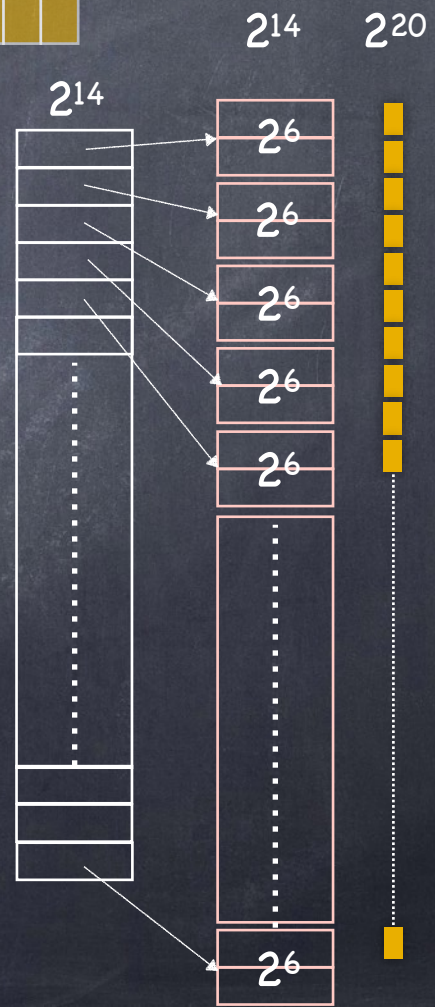
- What is we use a tree?
  - We still need to account for 2^{20} pages...
  - ...but we are going to partition the PT in a sequence of chunks, each with 2^6 entries



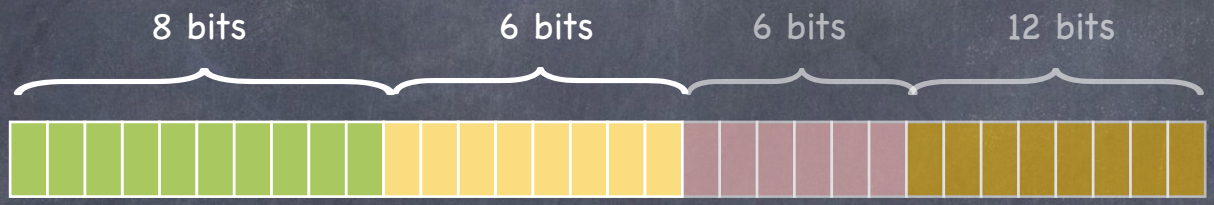
32-bit, 4GB, Example



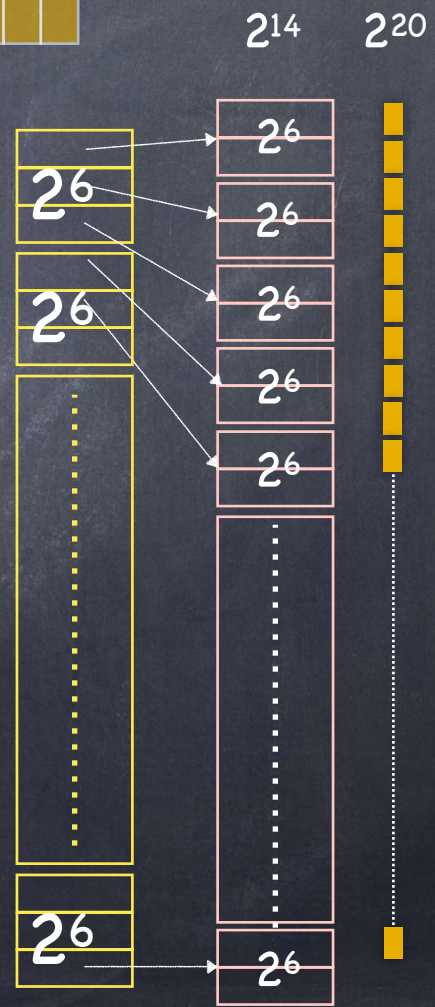
- What is we use a tree?
  - We still need to account for 2^{20} pages...
  - ...but we are going to partition the PT in a sequence of chunks, each with 2^6 entries
  - we'll need an index with 2^{14} entries...
  - ...which we'll partition in chunks of 2^6 entries



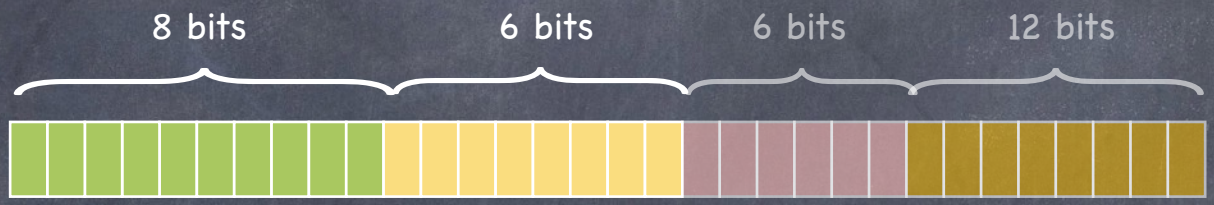
32-bit, 4GB, Example



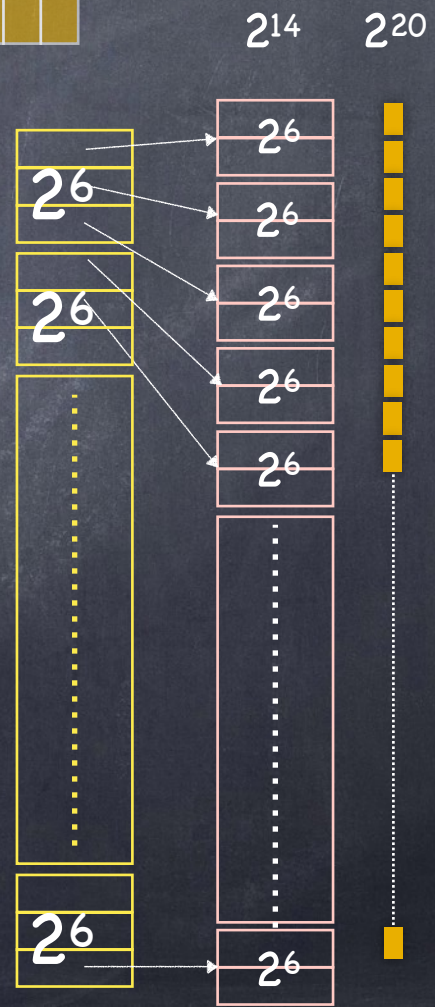
- What is we use a tree?
  - We still need to account for 2^{20} pages...
  - ...but we are going to partition the PT in a sequence of chunks, each with 2^6 entries
  - we'll need an index with 2^{14} entries...
  - ...which we'll partition in chunks of 2^6 entries



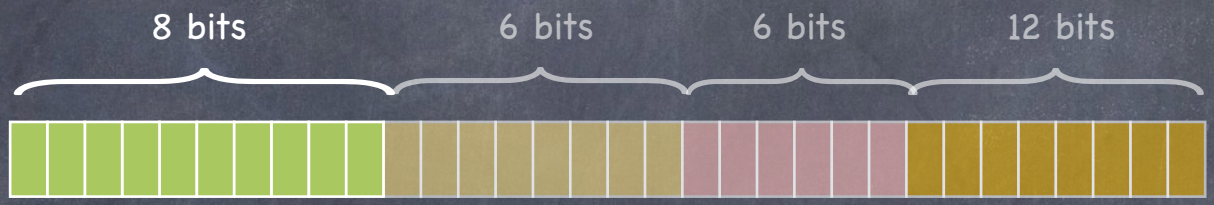
32-bit, 4GB, Example



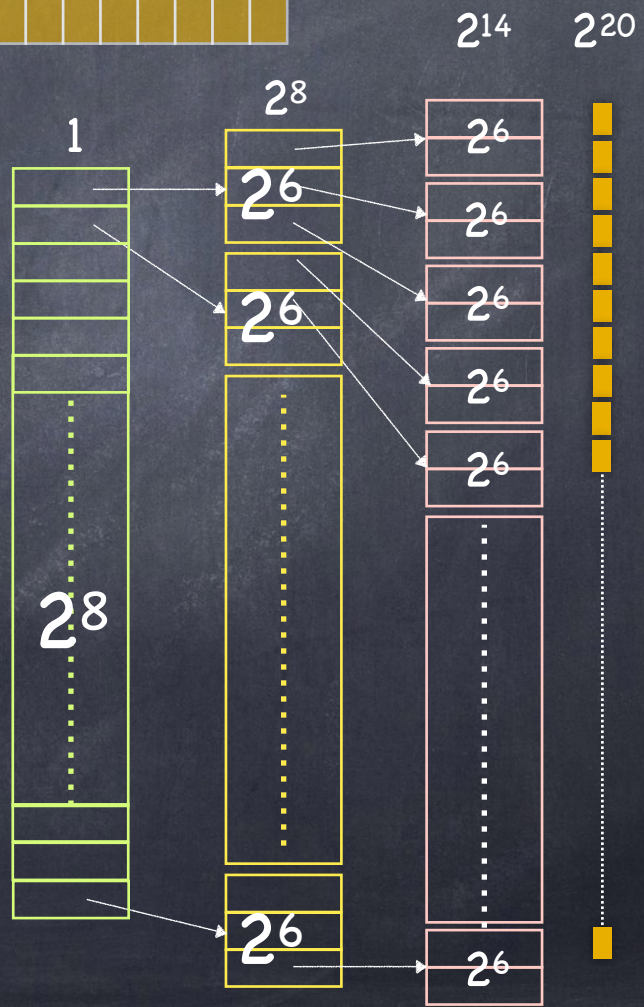
- What is we use a tree?
  - We still need to account for 2^{20} pages...
  - ...but we are going to partition the PT in a sequence of chunks, each with 2^6 entries
  - we'll need an index with 2^{14} entries...
  - ...which we'll partition in chunks of 2^6 entries
  - We'll need an index of 2^8



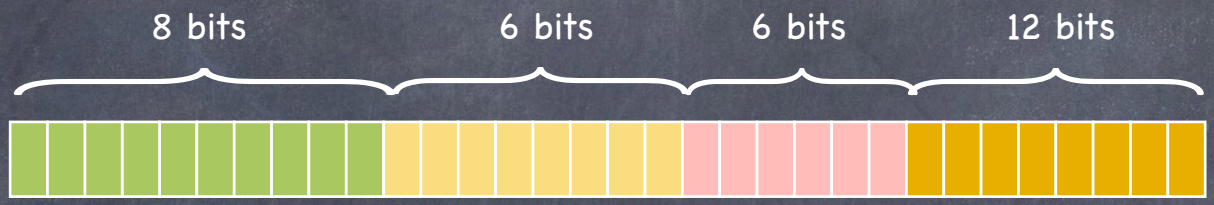
32-bit, 4GB, Example



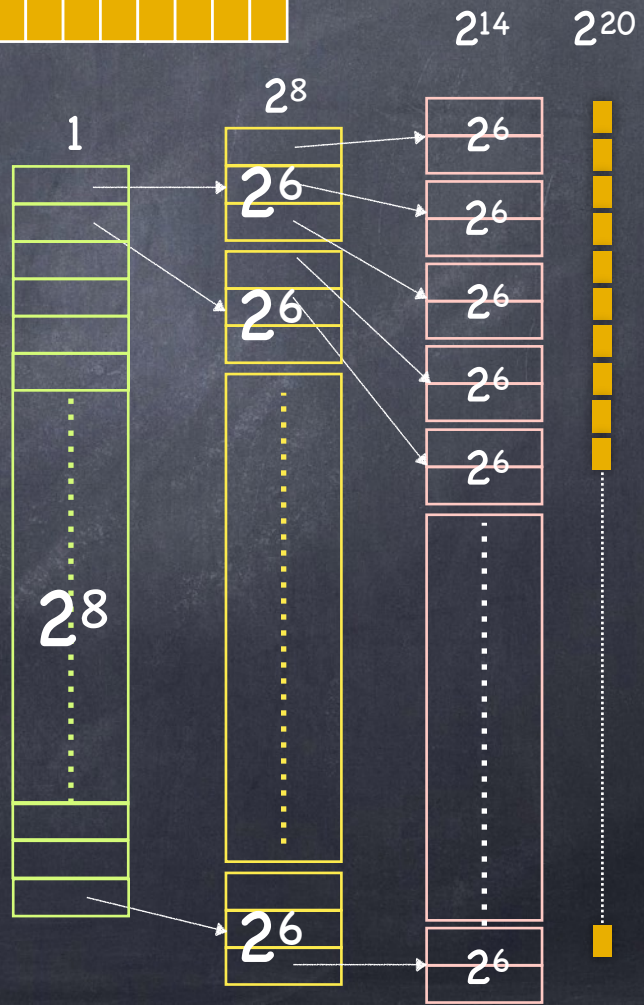
- What is we use a tree?
  - We still need to account for 2^{20} pages...
  - ...but we are going to partition the PT in a sequence of chunks, each with 2^6 entries
  - we'll need an index with 2^{14} entries...
  - ...which we'll partition in chunks of 2^6 entries
  - We'll need an index of 2^8



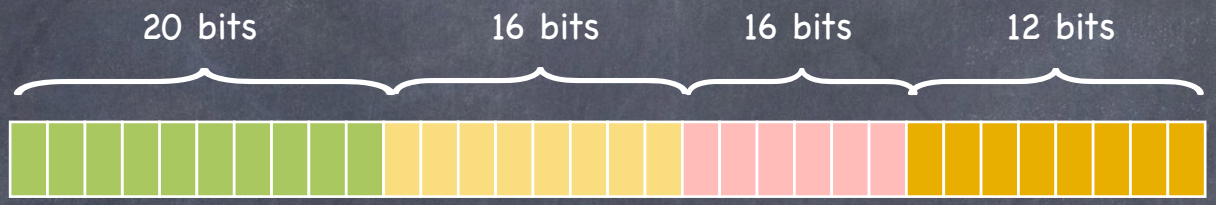
32-bit, 4GB, Example



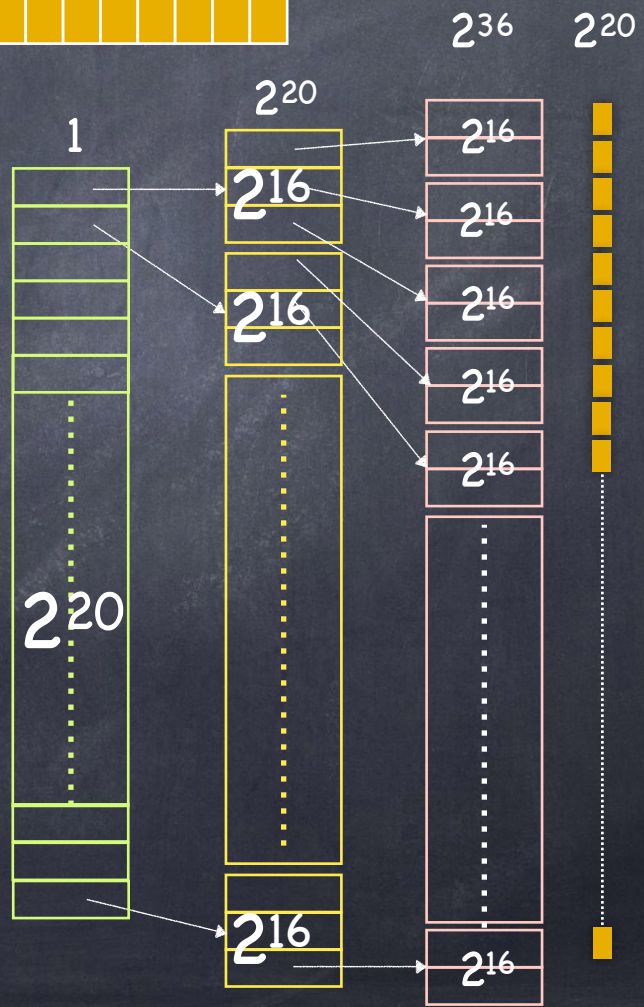
- Are we better off?
  - The number of PT entries now is $(2^6 \times 2^{14}) + (2^8 \times 2^6) + 2^8 > 2^{20}$!!
  - Slightly larger than 4MB
    - What we needed for a single-level page table
  - But….
    - We can now exploit "sparsity" in virtual address space



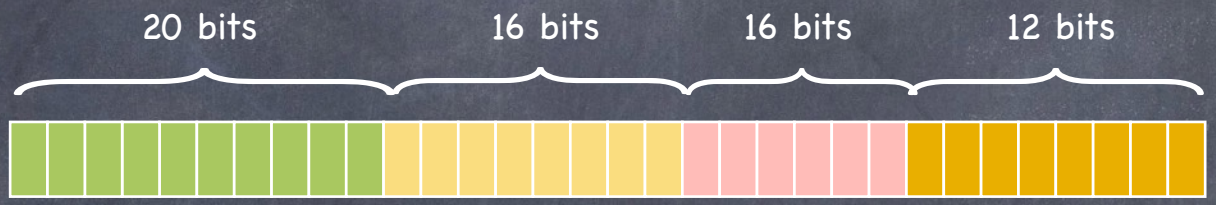
64-bit, 4GB, Example



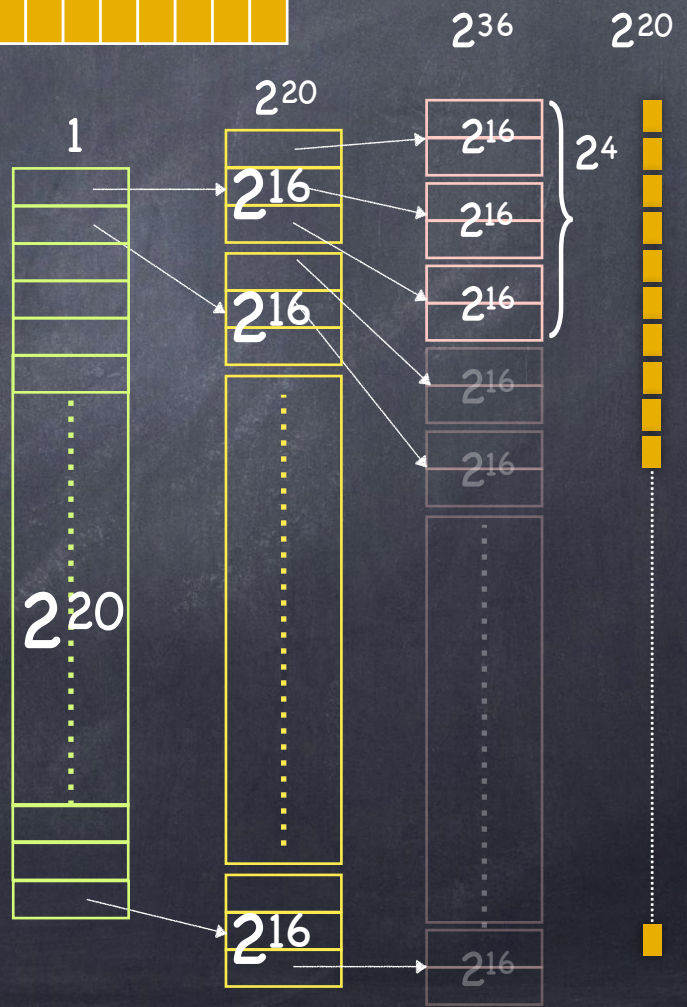
- Naïvely storing page table:
  - The number of PT entries now is $(2^{16} \times 2^{36}) + (2^{16} \times 2^{20}) + 2^{20} >> 2^{20}$!!
- We don't need all the entries
  - Store only that part of the tree that is needed



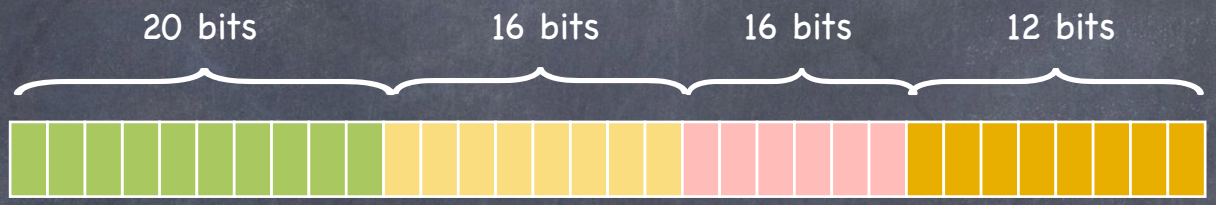
64-bit, 4GB, Example



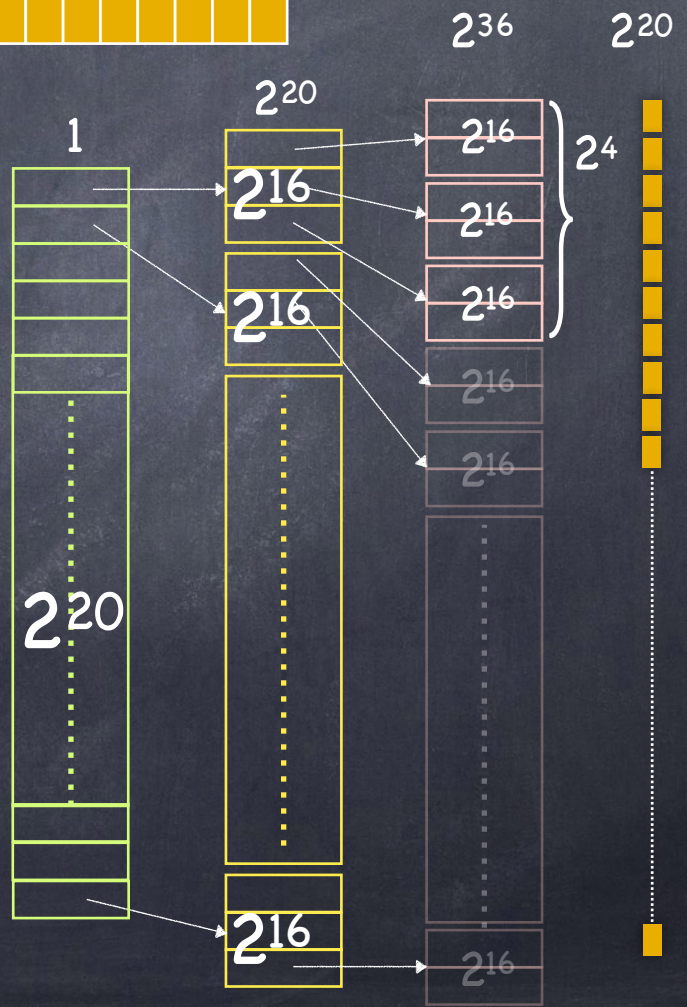
- How many chunks of size 2^{16} are needed to hold 2^{20} PTEs of frames starting at 0?
  - $2^{20}/2^{16} = 2^4 = 16$



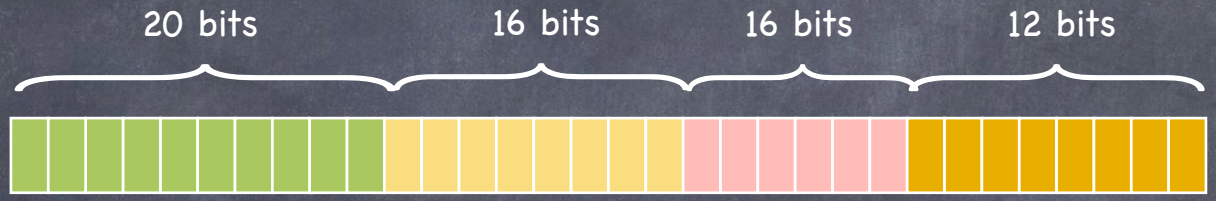
64-bit, 4GB, Example



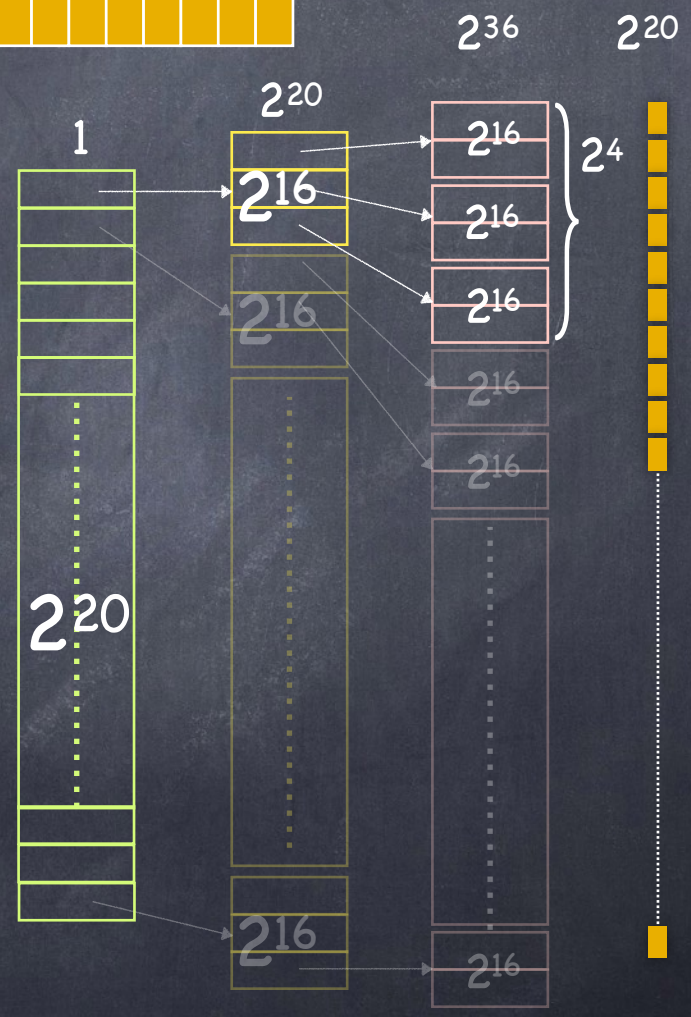
- How many chunks of size 2^{16} are needed to hold 2^{20} PTEs of frames starting at 0?
  - $2^{20}/2^{16} = 2^4 = 16$
- How many chunks of size 2^{16} are needed to hold pointers to 16 pink chunks?
  - 1



64-bit, 4GB, Example



- How many chunks of size 2^{16} are needed to hold 2^{20} PTEs of frames starting at 0?
  - $2^{20}/2^{16} = 2^4 = 16$
- How many chunks of size 2^{16} are needed to hold pointers to 16 pink chunks?
  - 1



- So, if each entry is 4 bytes, the PT takes
  - $4 * (1 \times 2^{20} + 1 \times 2^{16} + 16 \times 2^{20}) < 68.25\text{MB}$
  - Can be further reduced a bit

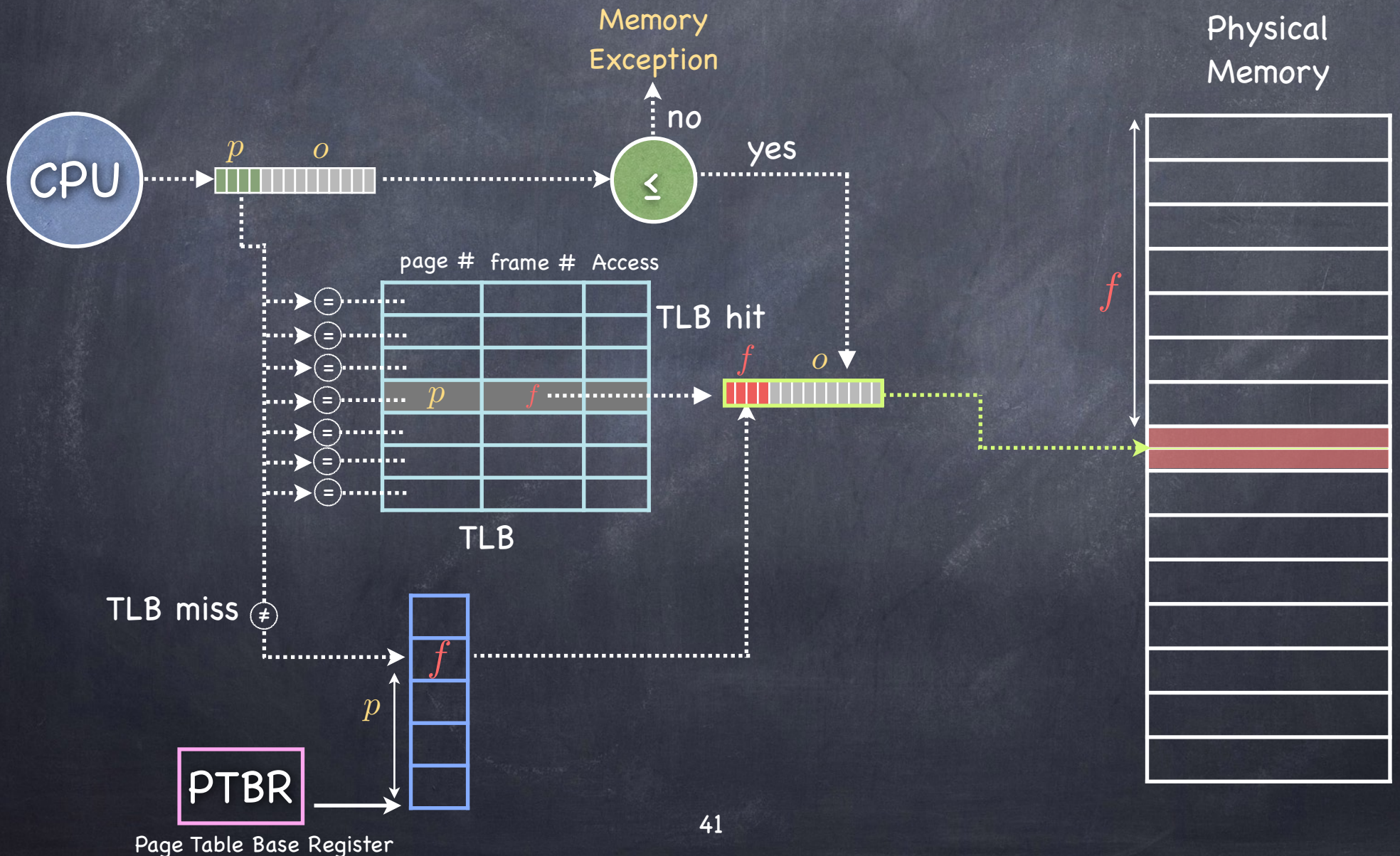
Where are we?

- Storage overheads
  - Minimized! Using multi-level page tables.
- How about address translation time?
  - Every new level of paging
    - reduces the memory overhead for computing the mapping function...
    - ... but increases the time necessary to perform the mapping function

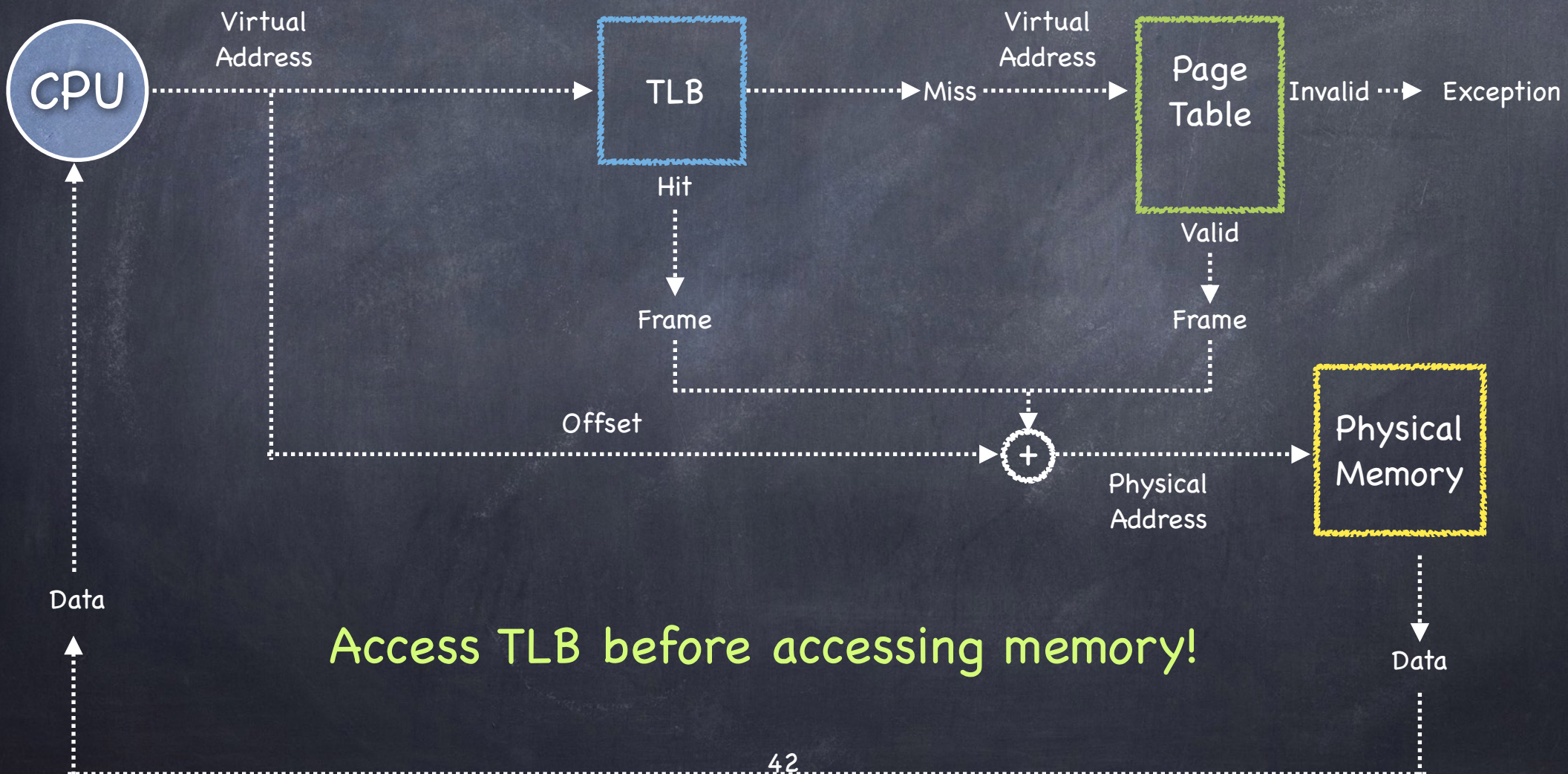
Caching!

- Keep the results of recent virtual address to physical address translations in a structure called Translation Lookaside Buffer (TLB)

Speeding things up: The TLB



Address Translation with TLB



Access TLB before accessing memory!

Hit and Miss

- The TLB is small; it cannot hold all PTEs
  - it can be fast only if it is small!
  - Some translations will inevitably miss the TLB
  - Must access memory to find the appropriate PTE
    - called walking the page table
    - incurs large performance penalty

Handling TLB Misses

- Hardware-managed (e.g., x86)
  - The hardware does the page walk
  - Hardware fetches PTE and inserts it in TLB
    - If TLB is full, must replace another TLB entry
  - Done transparently to system software
- Software-managed (e.g., MIPS)
  - Hardware raises an exception
  - OS does the page walk, fetches PTE, and inserts evicts entries in TLB

Tradeoffs, Tradeoffs...

- Hardware-managed TLB
  - + No exception on TLB miss. Instruction just stalls
  - + No extra instruction/data brought into the cache
  - – OS has no flexibility in deciding Page Table organization
  - – OS has no flexibility in TLB entry replacement policy
- Software-managed TLB
  - + OS can define Page Table organization
  - + More flexible TLB entry replacement policies
  - – Slower: exception causes to flush pipeline; execute handler; pollute cache

TLB Consistency - I

- On context switch
  - VAs of old process should no longer be valid
  - Change PTBR — but what about the TLB?

TLB Consistency - I

- On context switch
  - VAs of old process should no longer be valid
  - Change PTBR — but what about the TLB?
    - Option 1: Flush the TLB
    - Option 2: Add pid tag to each TLB entry

	PID	VirtualPage	PageFrame	Access
TLB Entry	1	0x0053	0x0012	R/W

Ignore entries with wrong PIDs

TLB Consistency - II

- What if OS changes permissions on page?
  - If permissions are reduced, OS must ensure affected TLB entries are purged
  - If permissions are expanded, no problem
    - new permissions will cause an exception and OS will restore consistency