Lecture 13: Memory Management

Swap and Paging

Recall: address space

- What I have
  - A certain amount of physical memory
  - Multiple programs I would like to run
    - together, they may need more than the available physical memory
- What I want: an Address Space
  - Each program has as much memory as the machine's architecture will allow to name
  - All for itself

Recall: Virtual Address Space: An Abstraction for Memory

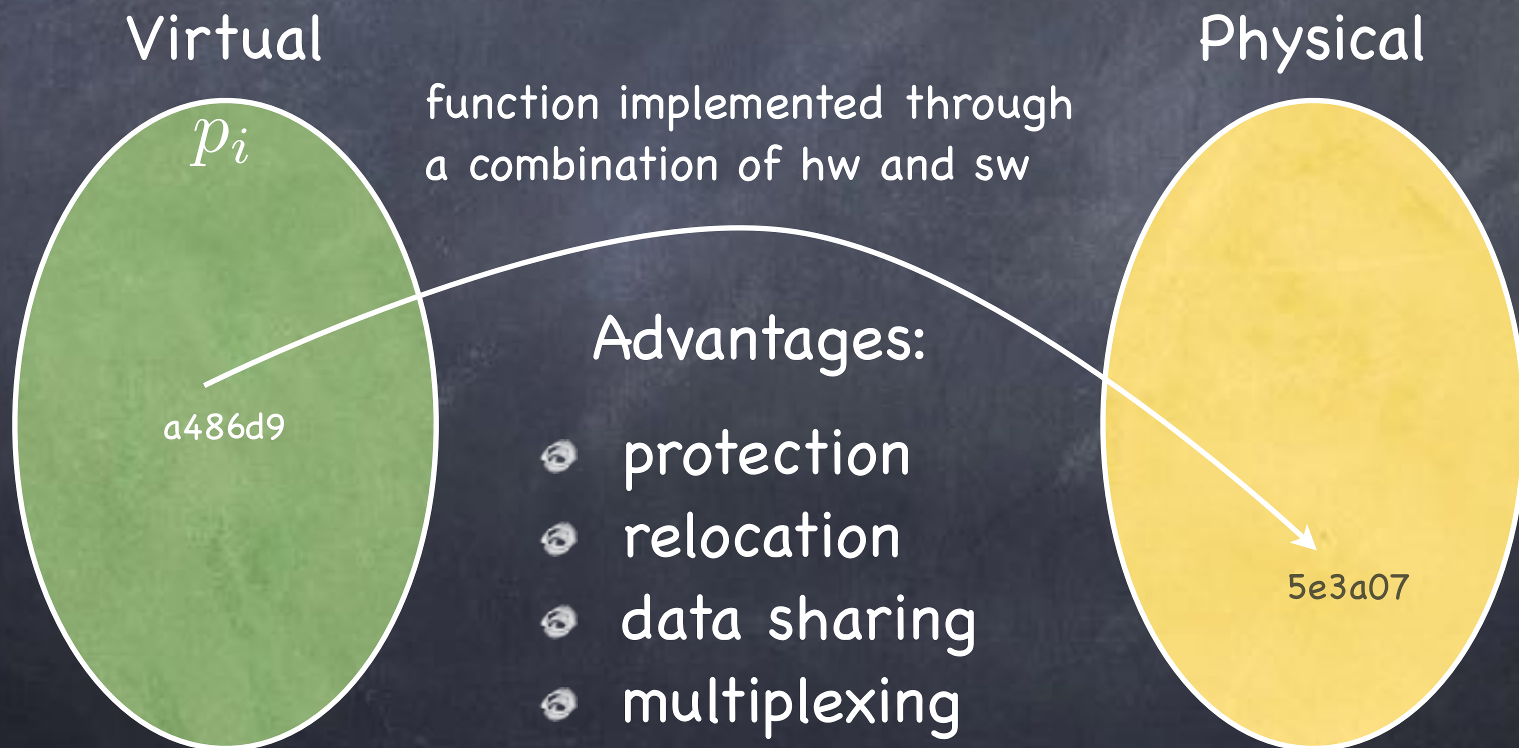
- Virtual addresses start at 0
- Heap and stack can be placed far away from each other, so they can nicely grow
- Addresses are all contiguous
- Size is independent of physical memory on the machine

Recall: Physical Address Space: How memory actually looks

- Processes loaded in memory at some memory location
  - virtual address 0 is not loaded at physical address 0
- Multiple processes may be loaded in memory at the same time, and yet...
- ...physical memory may be too small to hold even a single virtual address space in its entirety
  - e.g., 64-bit

Recall: Address Translation

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



Recall: Memory Management Unit

- Hardware device
  - Maps virtual addresses to physical addresses

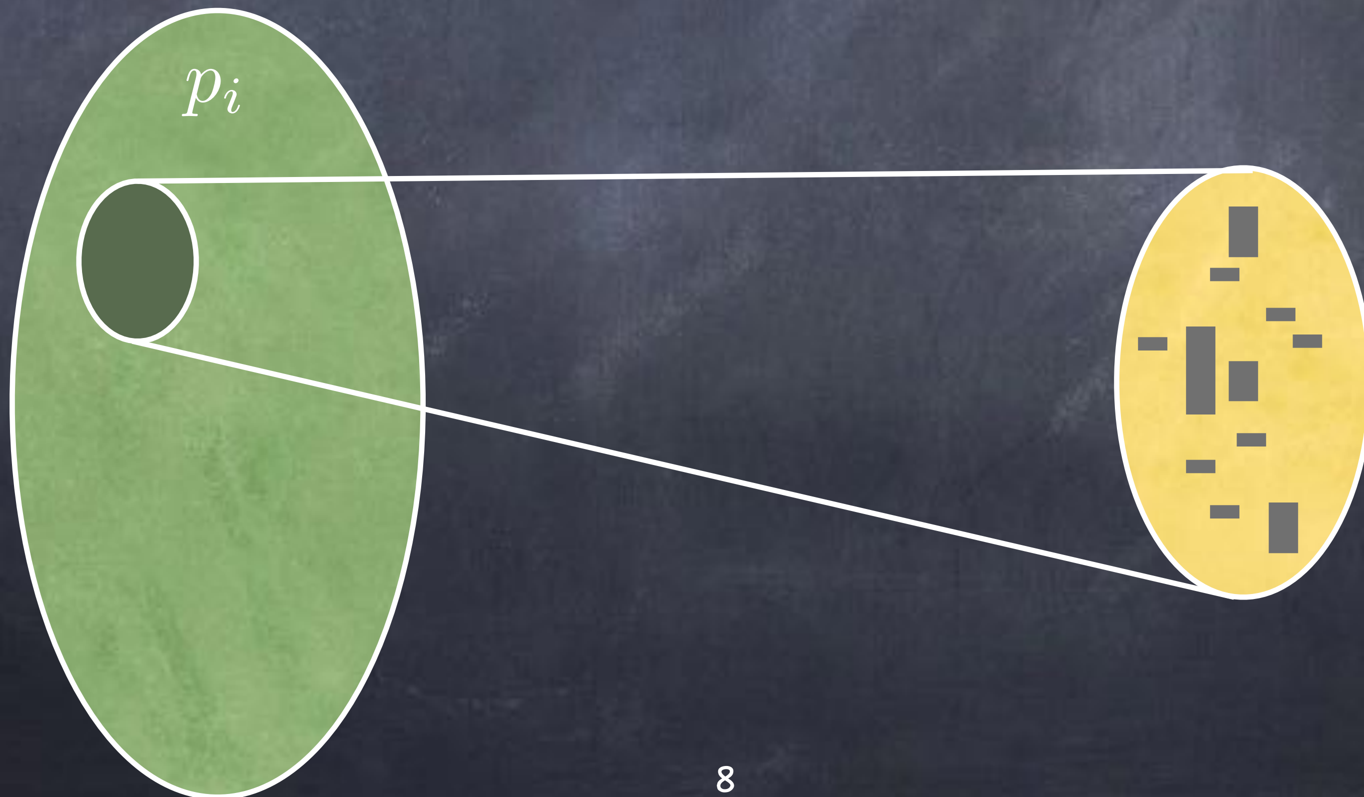
- User process
  - deals with virtual addresses
  - never sees the physical address
- Physical memory
  - deals with physical addresses
  - never sees the virtual address

Recall: virtual address space enables ...

- Protection
- Relocation
- Multiplexing
- Data sharing
- (Non)Contiguity

Recall: Non-Contiguity

- Contiguous virtual addresses need not map to contiguous physical addresses



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: Base & Bound

- Goal: let multiple processes coexist in memory while guaranteeing isolation
- Needed hardware
  - two registers: Base and Bound (a.k.a. Limit)
  - Stored in the PCB
- Mapping
  - pa = va + Base
    - as long as $0 \le va \le$ Bound
  - On context switch, change B&B (privileged instruction)

Recall: Base & Bound: Pros and Cons

- Pros: Contiguous Allocation
  - contiguous virtual addresses are mapped to contiguous physical addresses
- Cons: mapping entire address space to physical memory
  - is wasteful
    - lots of free space between heap and stack…
    - makes sharing hard
  - does not work if the address space is larger than physical memory
    - think 64-bit registers…

Recall: Segments

- An address space comprises multiple segments
  - contiguous sets of virtual addresses, logically connected
    - heap, code, stack, (and also globals, libraries...)
  - each segment can be of a different size

Recall: Segmentation: Generalizing Base & 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

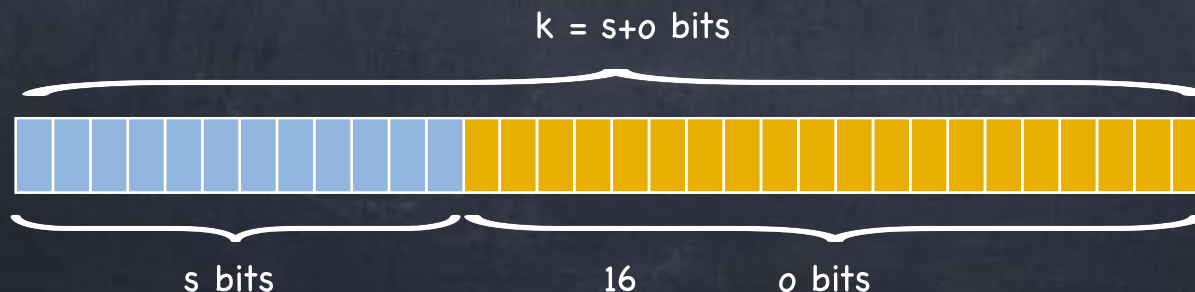
Questions?

Segmentation

- Goal: Supporting large address spaces (while allowing multiple processes to coexist in memory)
- Needed hardware
  - two registers (Base and Bound) per segment
    - values stored in the PCB
  - if many segments, a segment table, stored in memory, at an address pointed to by a Segment Table Register (STBR)
    - process' STBR value stored in the PCB

Segmentation: Mapping

- How do we map a virtual address to the appropriate segment?
  - Read VA as having two components
    - s most significant bits identify the segment
      - at most 2^s segments
    - o remaining bits identify offset within segment
      - each segment's size can be at most 2° bytes



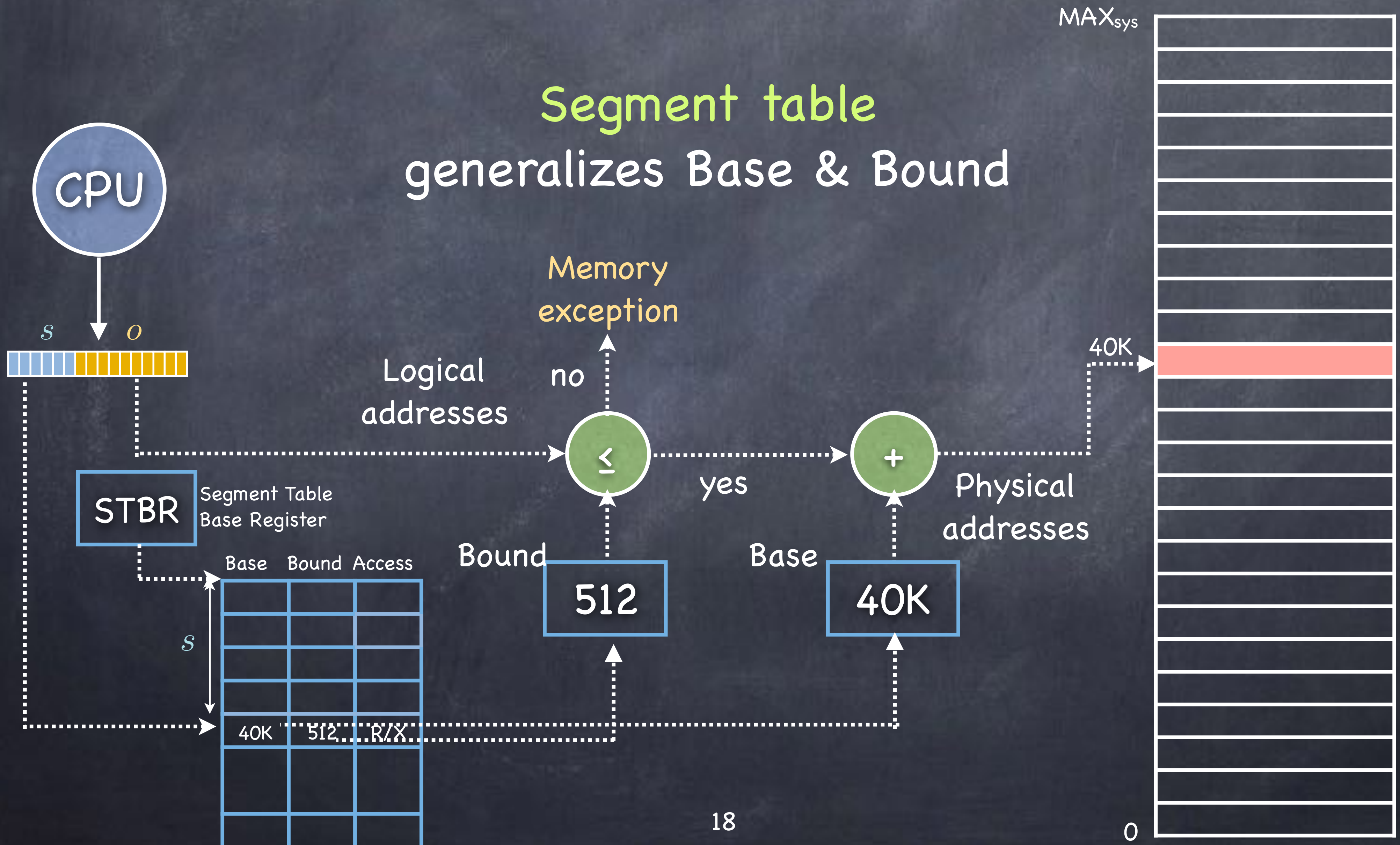
Segment Table

- Use s bits to index to the appropriate row of the segment table

	Base	Bound (Max 4k)	Access
$Code_{00}$	32K	2K	Read/Execute
$Heap_{01}$	34K	3K	Read/Write
$Stack_{10}$	28K	3K	Read/Write

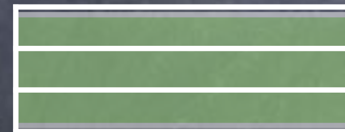
- Segments can be shared by different processes
  - use protection bits to determine if shared Read only (maintaining isolation) or Read/Write (if shared, no isolation)
    - processes can share code segment while keeping data private

Implementing Segmentation



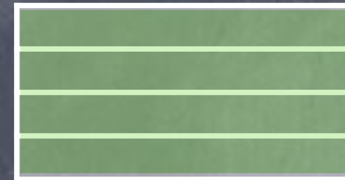
Managing Free space

- Many segments, different processes, different sizes
- OS tracks free memory blocks ("holes")
  - Initially, one big hole
- 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



External Fragmentation

- Over time, memory can become full of small holes
  - Hard to fit more segments
  - Hard to expand existing ones
- Compaction
  - Relocate segments to coalesce holes



External Fragmentation

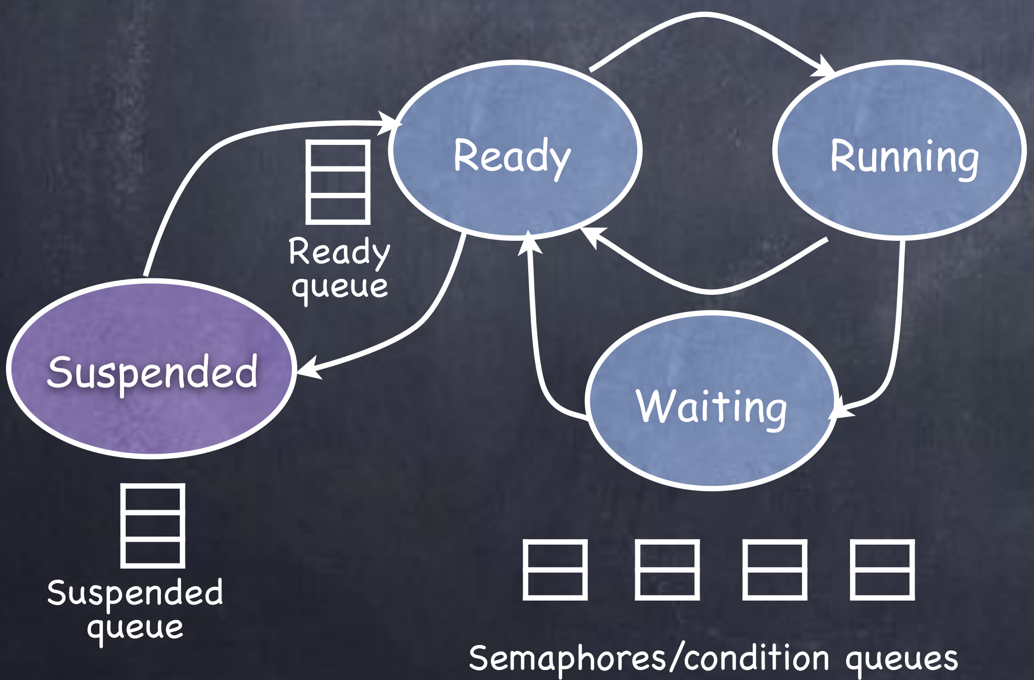
- Over time, memory can become full of small holes
  - Hard to fit more segments
  - Hard to expand existing ones
- Compaction
  - Relocate segments to coalesce holes

External Fragmentation

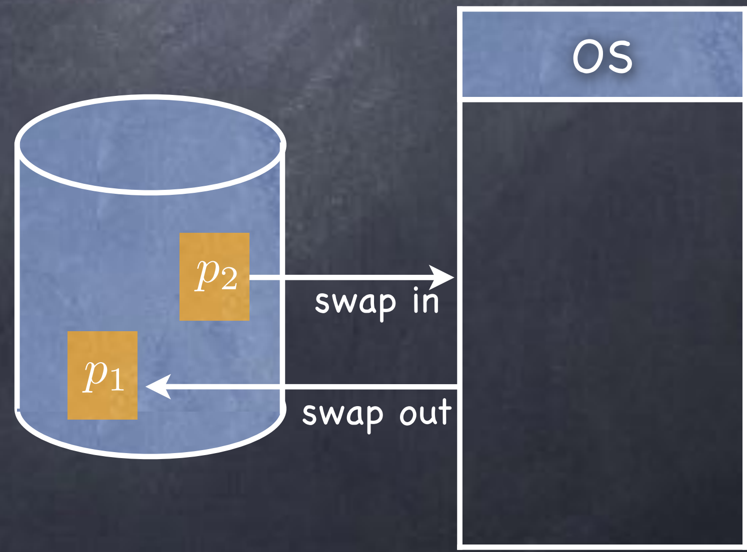
- Over time, memory can become full of small holes
  - Hard to fit more segments
  - Hard to expand existing ones
- Compaction
  - Relocate segments to coalesce holes
    - Copying eats up a lot of CPU time!
      - if 4 bytes in 10ns, 8 GB in 20s!
- But what if a segment wants to grow?

Eliminating External Fragmentation: Swapping Processes

- Preempt processes and reclaim their memory



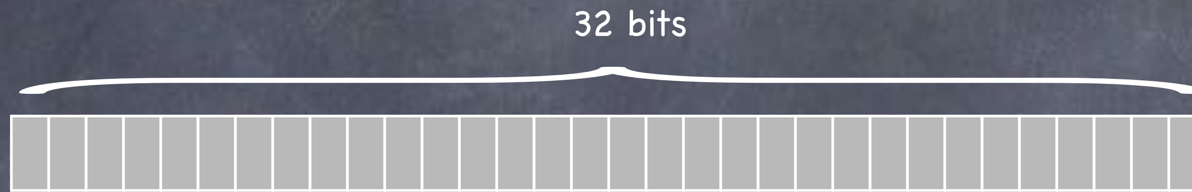
- Move images of suspended processes to backing store



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
    - 0011111001111011110000 one bit/frame
  - no more external fragmentation!
  - but now internal fragmentation (you just can't win...)
    - when memory needs are not a multiple of a page
    - typical size of page/frame: 4KB to 16KB

Virtual address



- Interpret VA as comprised of two components
  - page: which page?
  - offset: which byte within that page?

Virtual address

- Interpret VA as comprised of two components
  - page: which page?
    - no. of bits specifies no. of pages are in the VA space
  - offset: which byte within that page?

Virtual address

- Interpret VA as comprised of two components
  - page: which page?
    - no. of bits specifies no. of pages are in the VA space
  - offset: which byte within that page?
    - no. of bits specifies size of page/frame

Virtual address

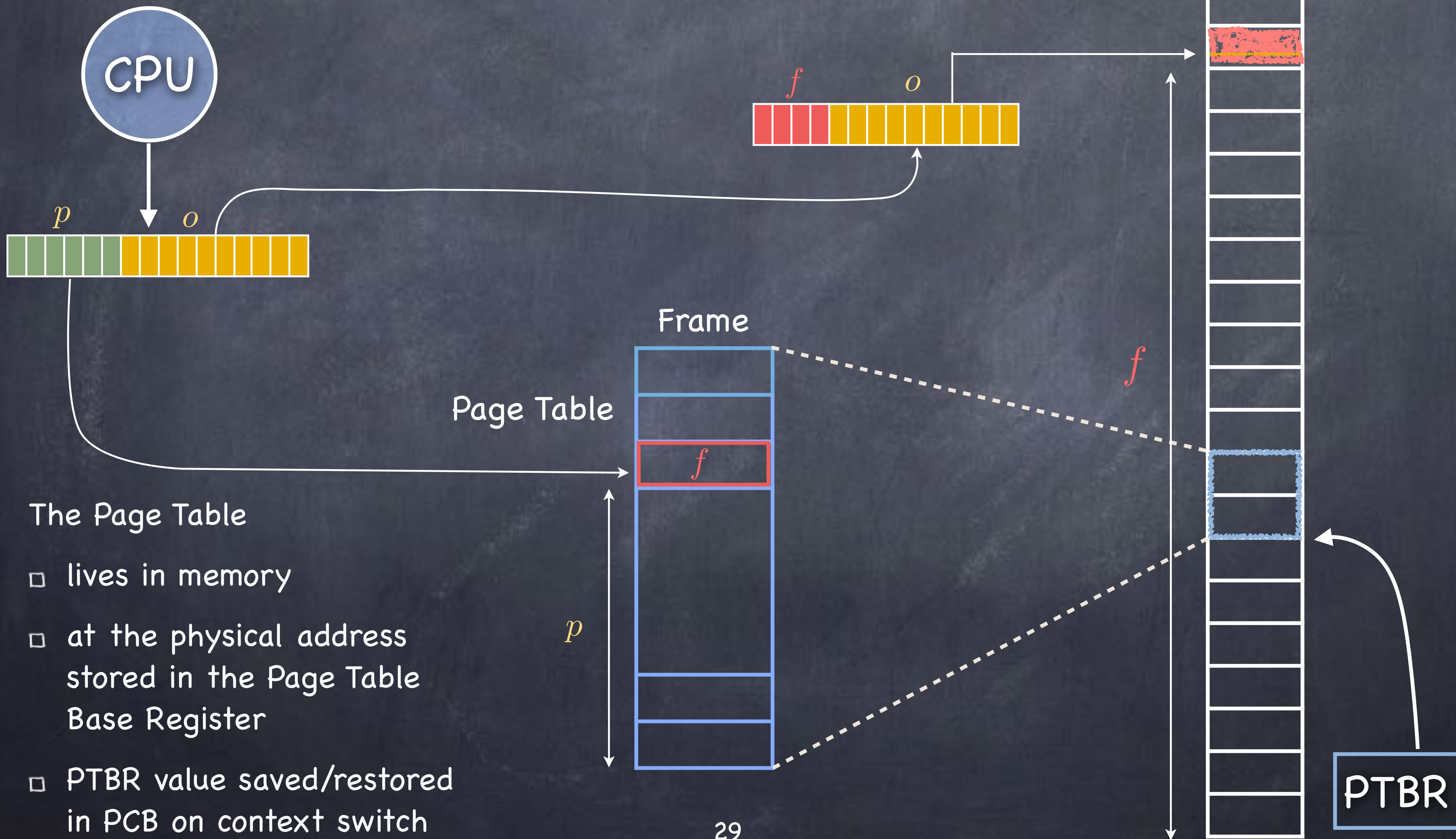
- To access a byte
  - extract page number
  - map that page number into a frame number using a page table
    - Note: not all pages may be mapped to frames
  - extract offset
  - access byte at offset in frame

Page Table

Index	Frame
$2^{20}-1$	8
.	
.	
.	
.	
.	
.	
.	
.	
.	
.	
.	
4	4
3	0
2	6
1	1
0	2

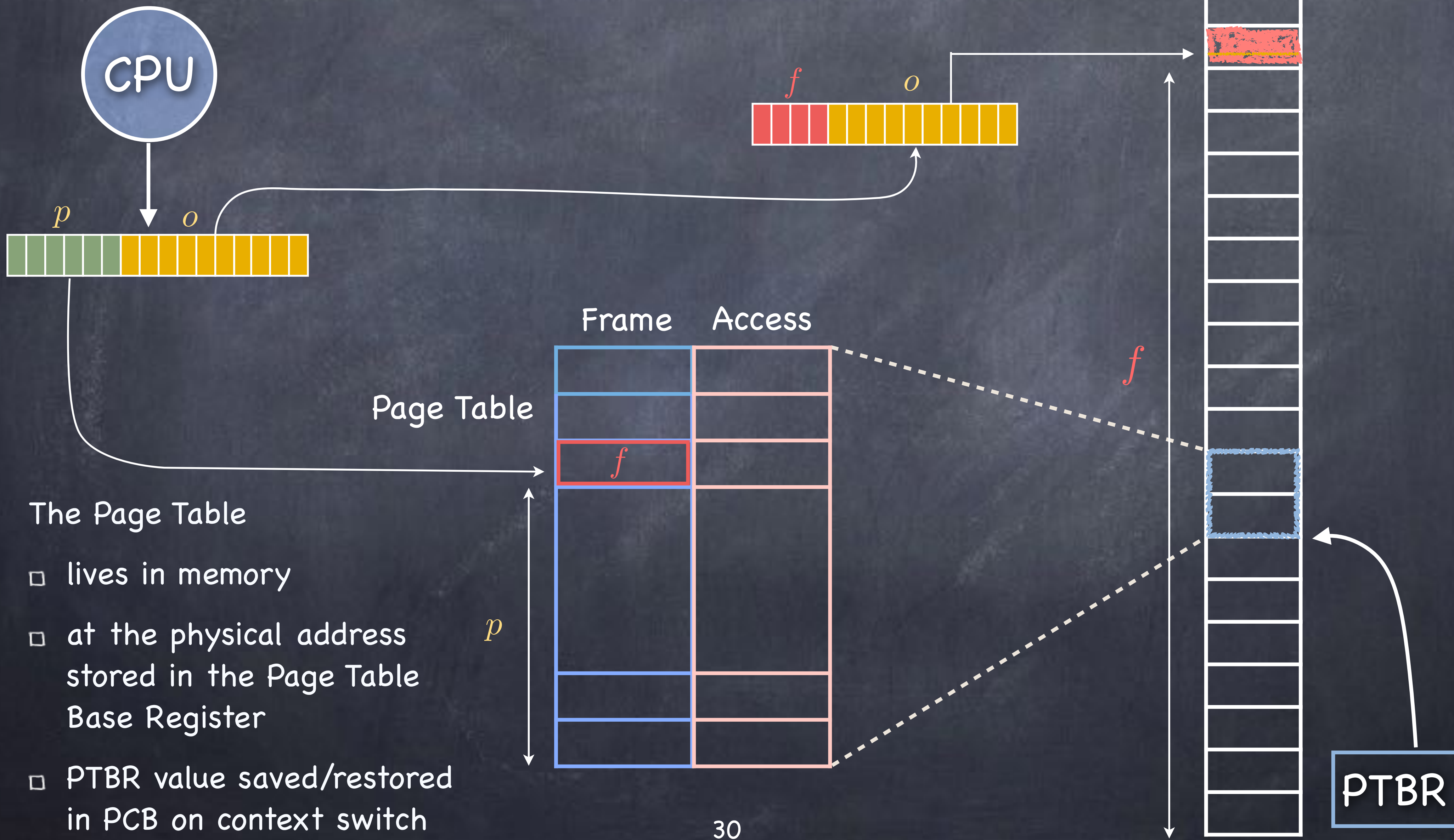
Physical Memory

Basic Paging



Basic Paging

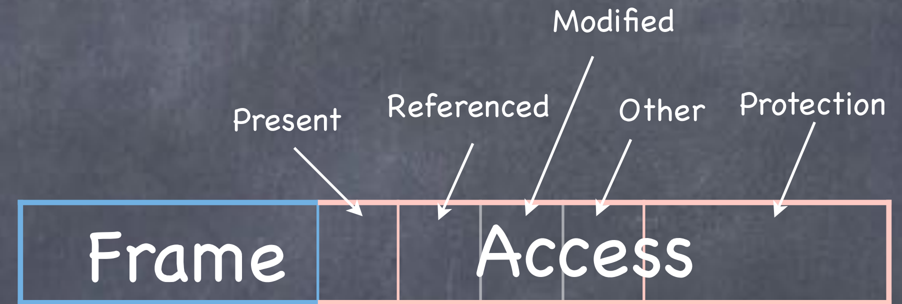
Physical Memory



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

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)

Page table

Protection bits (R/W/X)

Page	Frame	Valid/invalid
15	4	i
14	7	i
13	2	i
12	0	i
11	7	v
10	6	i
9	5	v
8	4	i
7	2	i
6	0	i
5	3	v
4	4	v
3	0	v
2	6	v
1	1	v
0	2	v

Referenced

Modified

Other

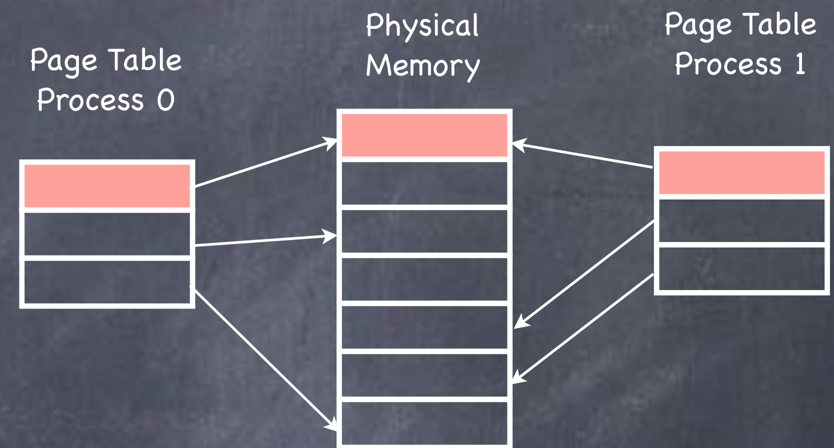
Valid/ invalid

Physical memory

Frame	Contents
7	11
6	2
5	9
4	4
3	5
2	0
1	1
0	3

Sharing

- Processes share a page by each mapping a page of their own virtual address space to the same frame
    - Fine tuning using protection bits (RWX)



Example

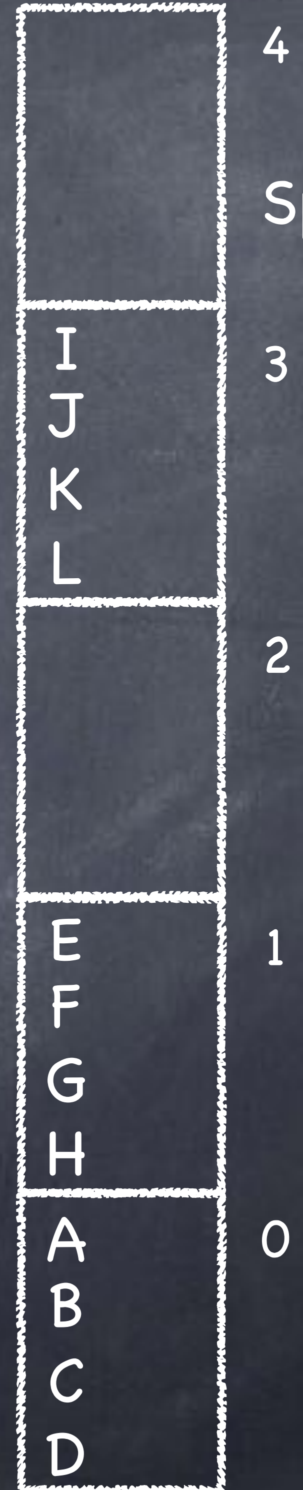
PA Space

VA Space

Page size: 4 bytes

Page Table

0	3
1	1
2	0



Space overhead

- Two sources, in tension:
  - data structure overhead (the Page Table itself)
  - fragmentation
    - How large should a page be?

Overhead for paging:

(#entries x sizeofEntry) + (#"segments" x pageSize/2) =

= ((VA_Size/pagesize) x sizeofEntry) + (#"segments" x pageSize/2)

- Size of entry
  - enough bits to identify physical page ($\log_2$ (PA_Size / page size))
  - should include control bits (present, dirty, referenced, etc)
  - usually word or byte aligned

Computing paging overhead

- 1 MB maximum VA, 1 KB page, 3 segments (program, stack, heap)
  - $((2^{20} / 2^{10}) \times sizeofEntry) + (3 \times 2^{9})$
  - If I know PA is 64 KB then
    - sizeofEntry = sizeofFrameNo + #ofControlBits
    - sizeofEntry =6 (2^{6} frames) + #ofControlBits
    - if 7 control bits, byte aligned size of entry: 16 bits

What's not to love?

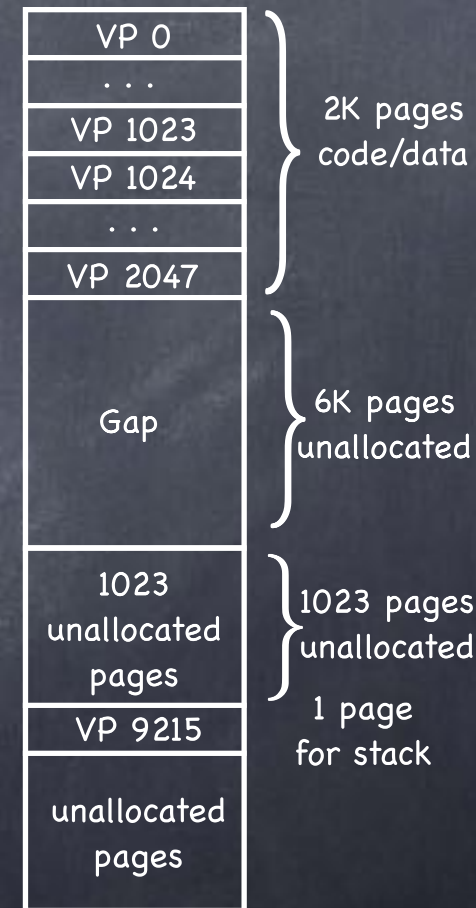
- Space overhead
  - With a 64-bit address space, size of page table can be huge
- Time overhead
  - Accessing data now requires two memory accesses
    - must also access page table, to find mapped frame

Reducing the Storage Overhead of Page Tables

- Size of the page table for a machine with 64-bit addresses and a page size of 4KB?
  - an array of 2^{52} entries!
- Good news
  - most space is unused
- Use a better data structure to express the Page Table
  - a tree!

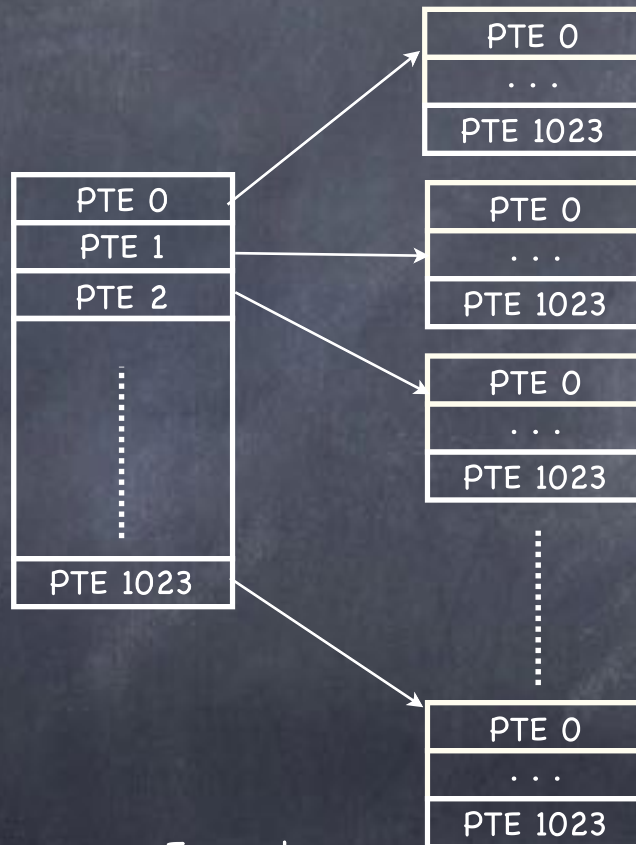
Example

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



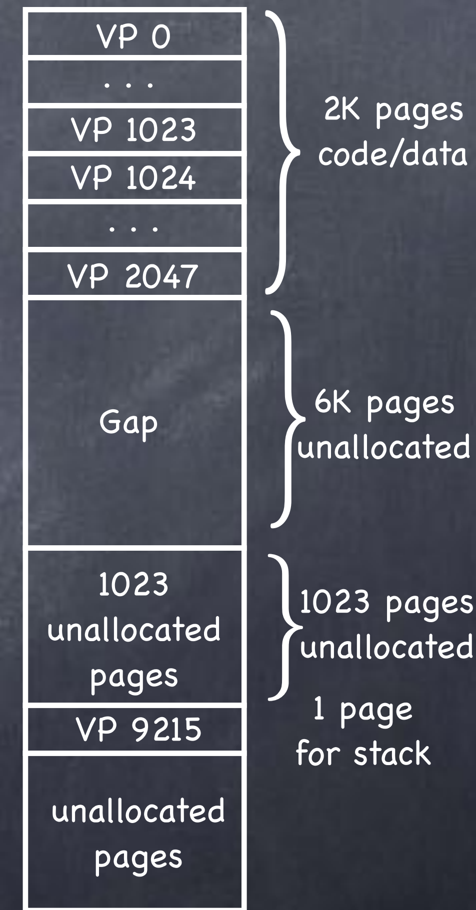
Reducing the Storage Overhead of Page Tables

- Size of the page table for a machine with 64-bit addresses and a page size of 4KB?
  - an array of 2^{52} entries!
- Good news
  - most space is unused
- Use a better data structure to express the Page Table
  - a tree!



Example

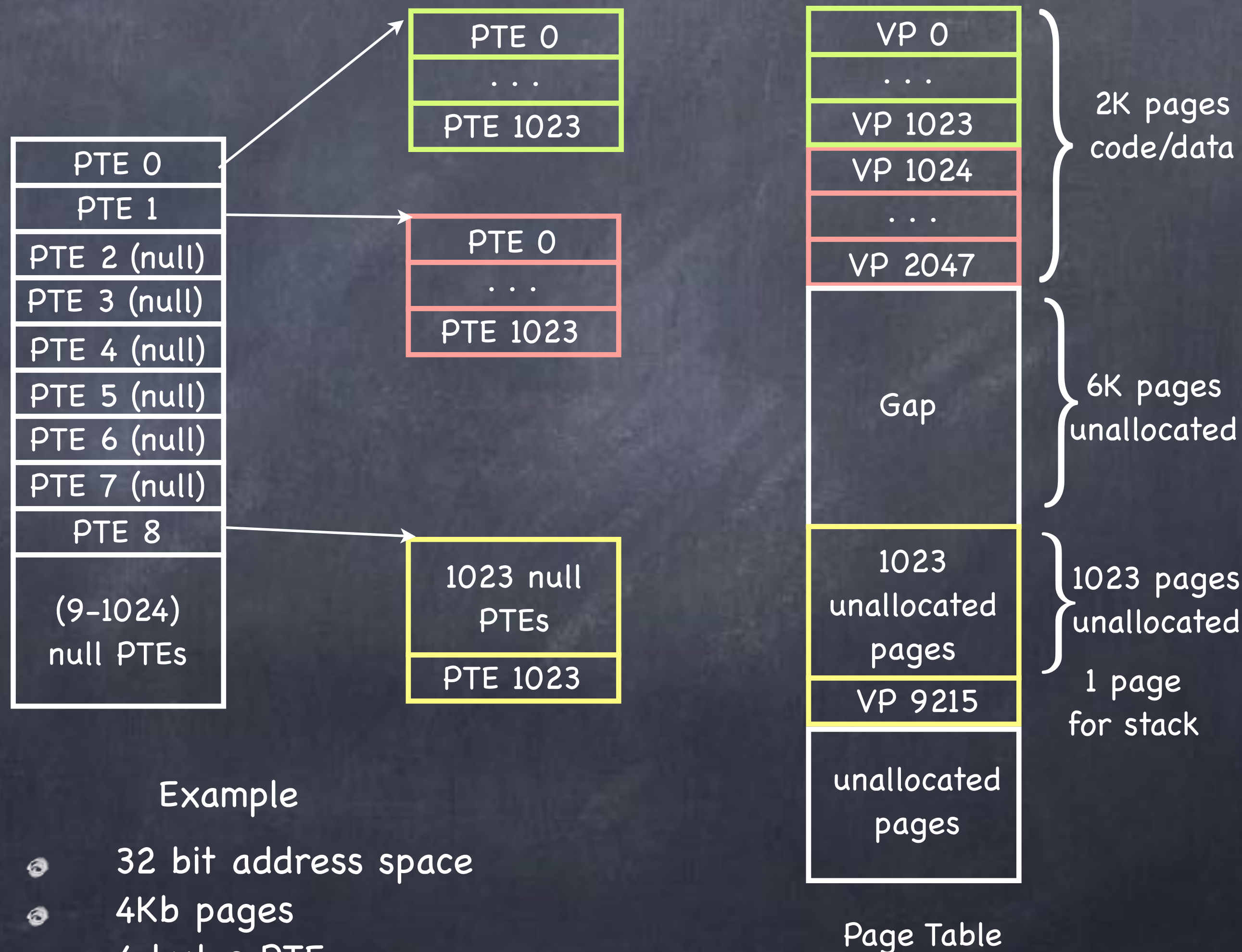
- 32 bit address space
- 4Kb pages
- 4 bytes PTE



Page Table

Reducing the Storage Overhead of Page Tables

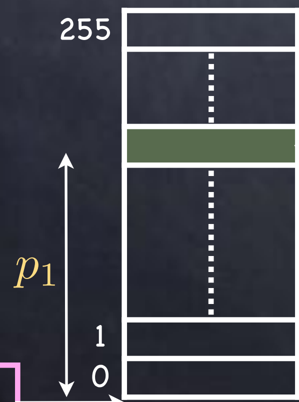
- Size of the page table for a machine with 64-bit addresses and a page size of 4KB?
  - an array of 2^{52} entries!
- Good news
  - most space is unused
- Use a better data structure to express the Page Table
  - a tree!

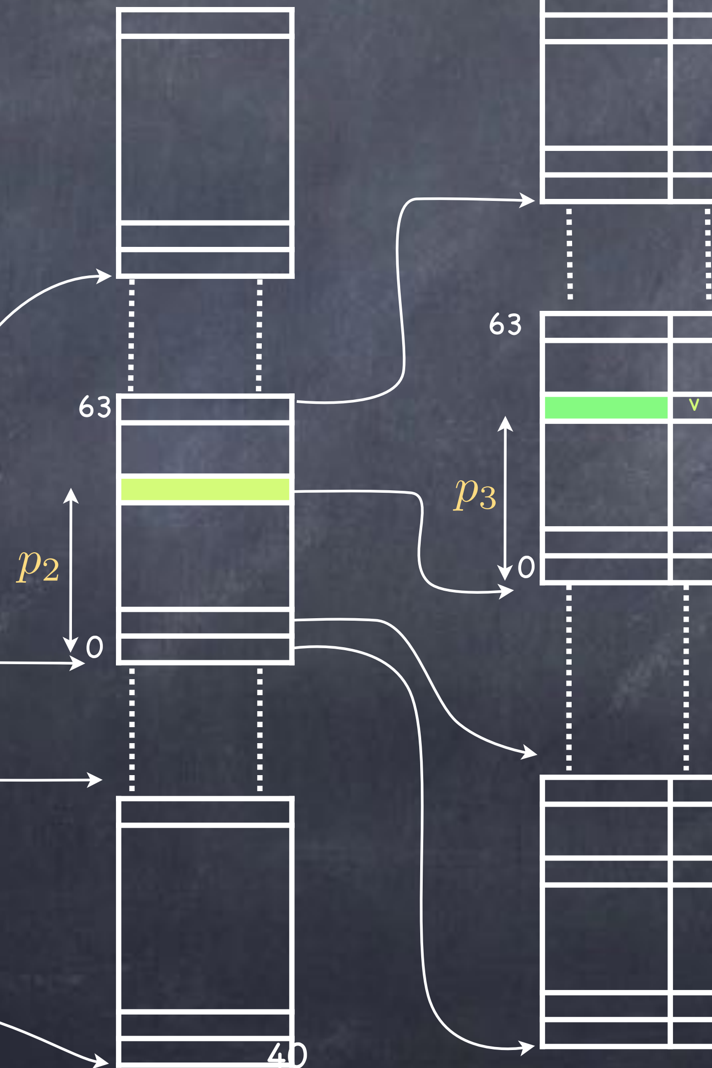


Multi-level Paging

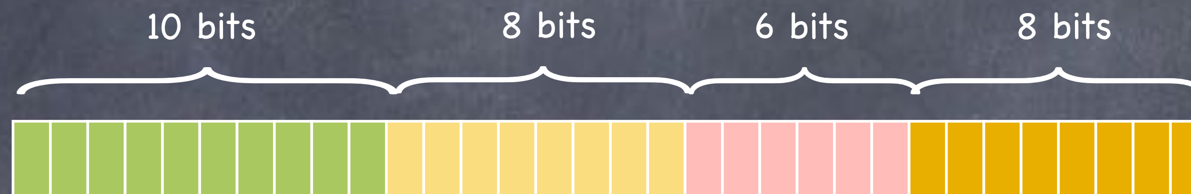
Structure virtual address space as a tree

Virtual address of a SPARC



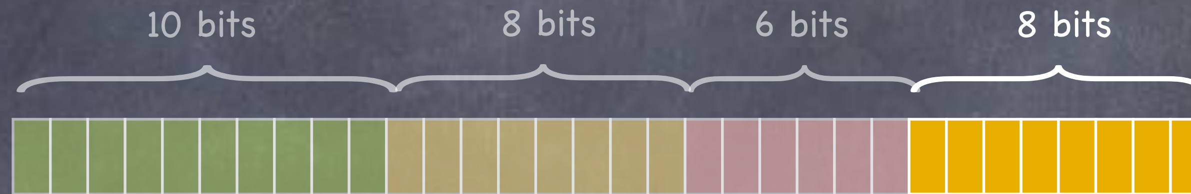


Example



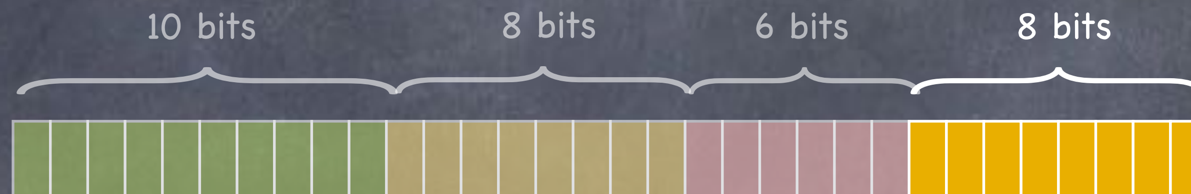
- What is the page size?

Example



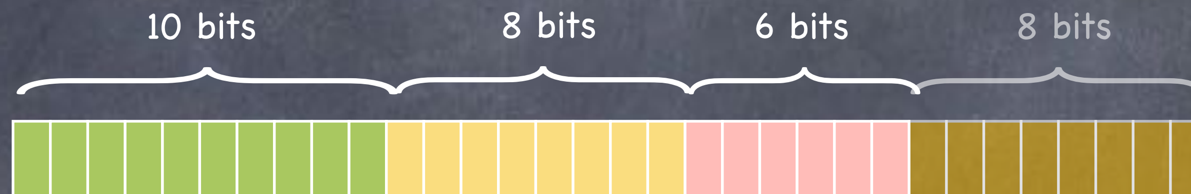
- What is the page size? Page size is 256 bytes (2^8)
- What is the Page Table size for a process that uses 256 contiguous KB of its VAS starting at address 0? [Assume each PTE is 2 bytes]
  - if we used a linear representation of the page table:

Example



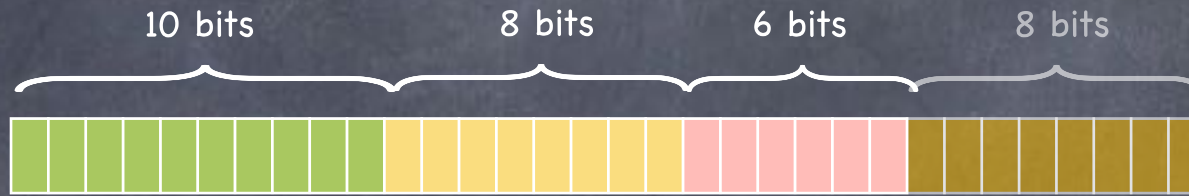
- What is the page size? Page size is 256 bytes (2^8)
- What is the Page Table size for a process that uses 256 contiguous KB of its VAS starting at address 0? [Assume each PTE is 2 bytes]
  - if we used a linear representation of the page table:
    - Page Table has 2^{24} entries

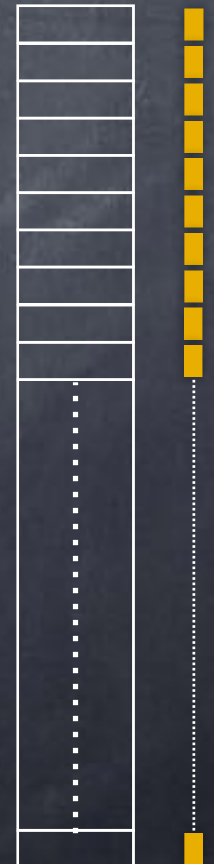
Example



- What is the page size? Page size is 256 bytes (2^8)
- What is the Page Table size for a process that uses 256 contiguous KB of its VAS starting at address 0? [Assume each PTE is 2 bytes]
  - if we used a linear representation of the page table:
    - Page Table has 2^{24} entries
    - PT Size: 2^{24} x 2 bytes = 2^{25} bytes = 32MB

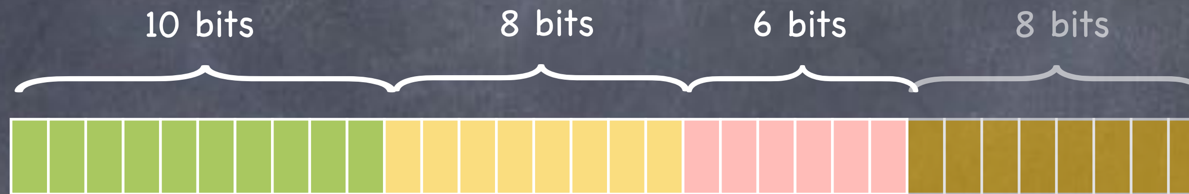
Example



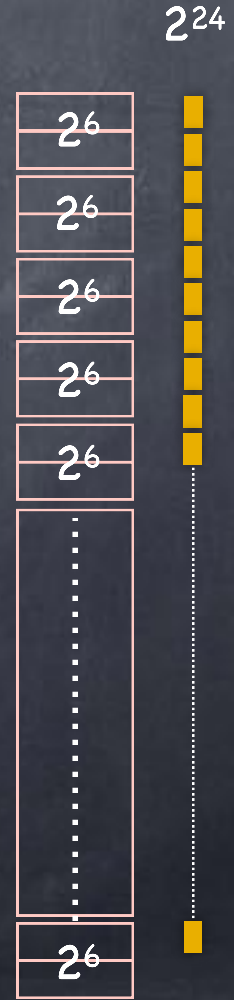


- What is we use a tree?
  - We still need to account for 2^{24} pages...
  - ...but we are going to partition the PT in a sequence of chunks, each with 2^{6} entries

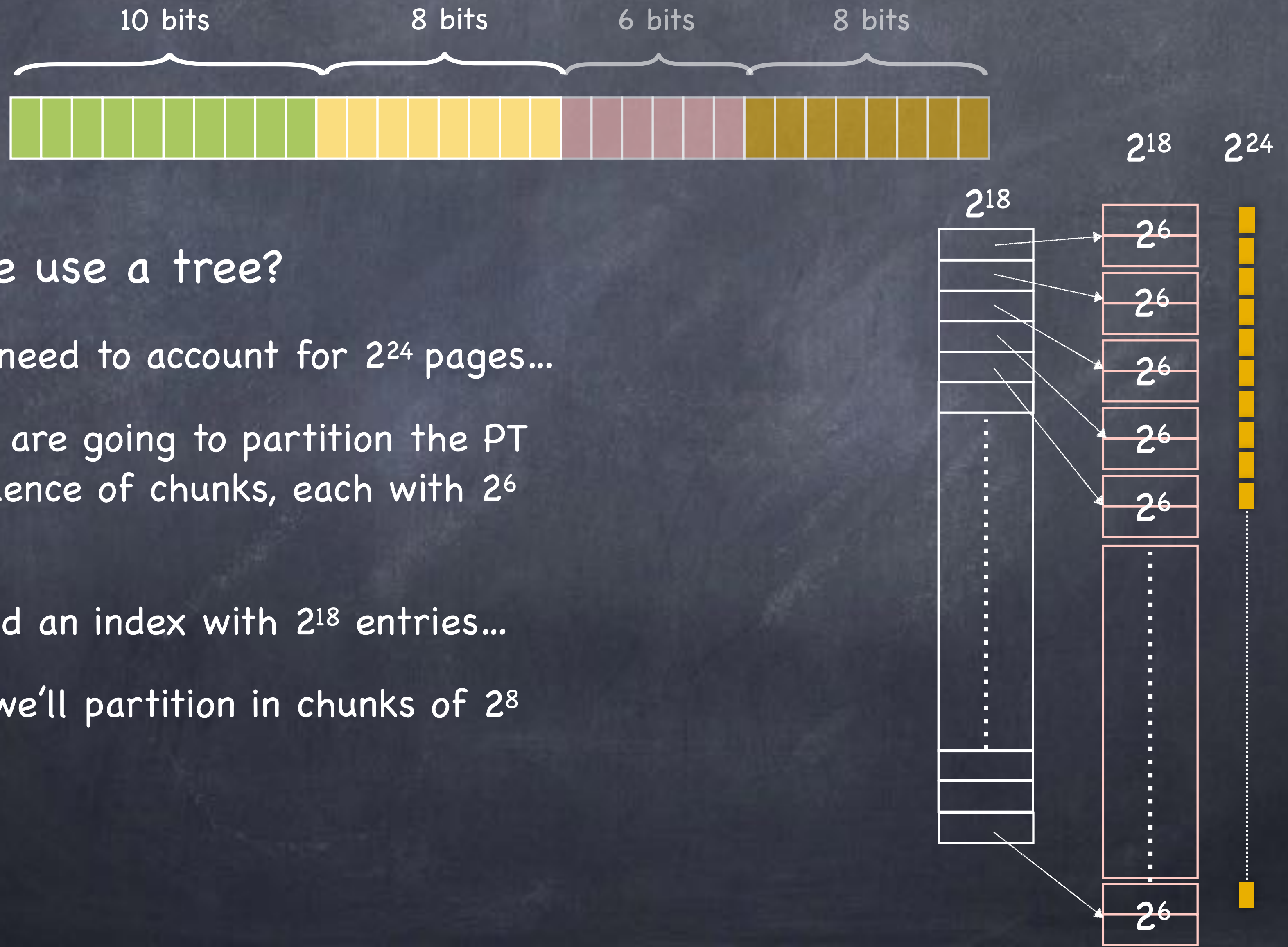
Example



- What is we use a tree?
  - We still need to account for 2^{24} pages…
  - …but we are going to partition the PT in a sequence of chunks, each with 2^6 entries

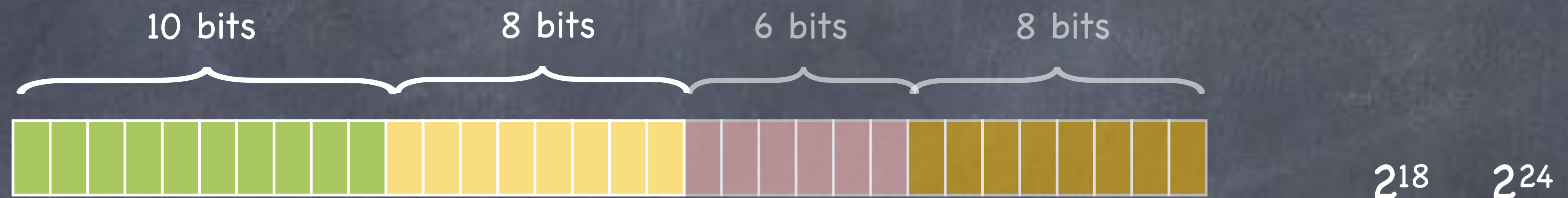


Example

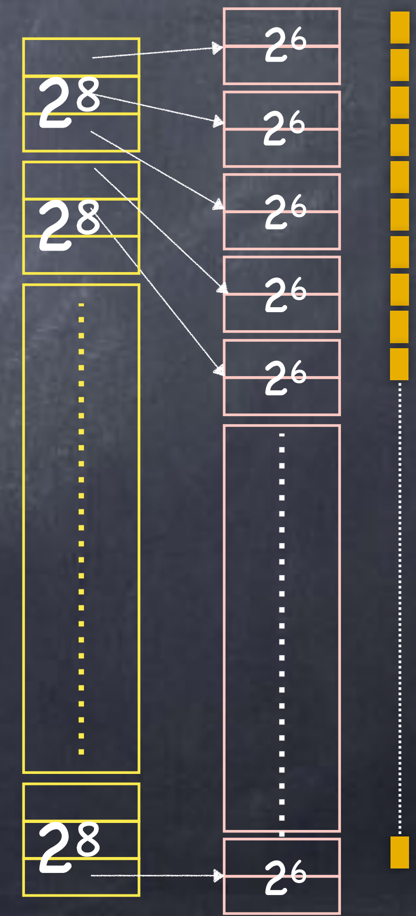


- What is we use a tree?
  - We still need to account for 2^{24} pages…
  - …but we are going to partition the PT in a sequence of chunks, each with 2^6 entries
  - we'll need an index with 2^{18} entries…
  - …which we'll partition in chunks of 2^8 entries

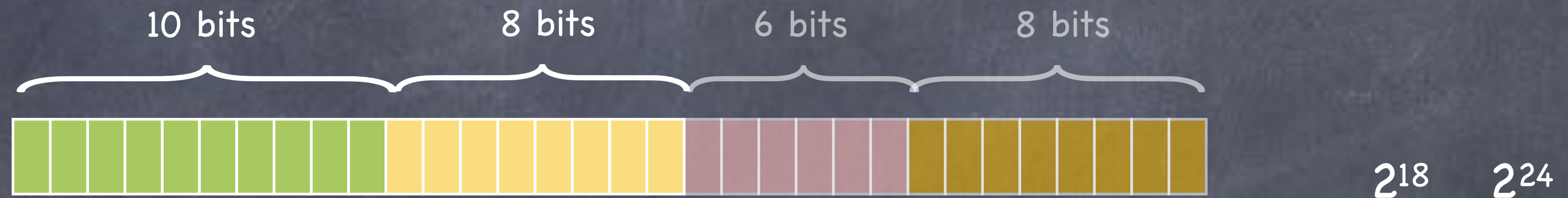
Example



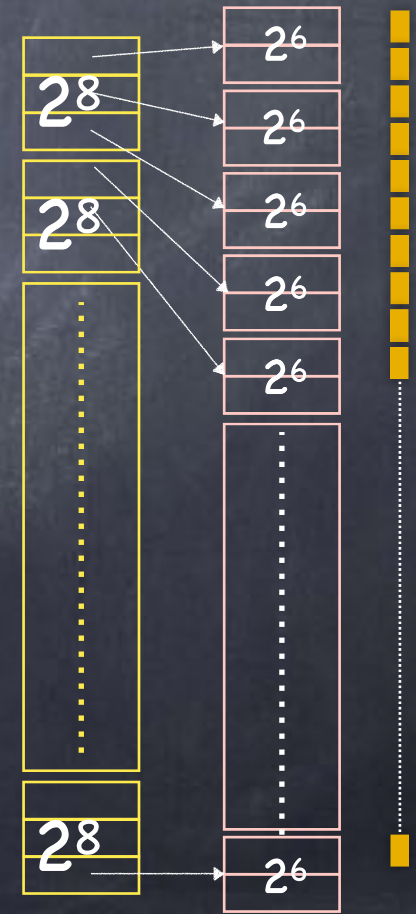
- What is we use a tree?
  - We still need to account for 2^{24} pages…
  - …but we are going to partition the PT in a sequence of chunks, each with 2^{6} entries
  - we'll need an index with 2^{18} entries…
  - …which we'll partition in chunks of 2^{8} entries



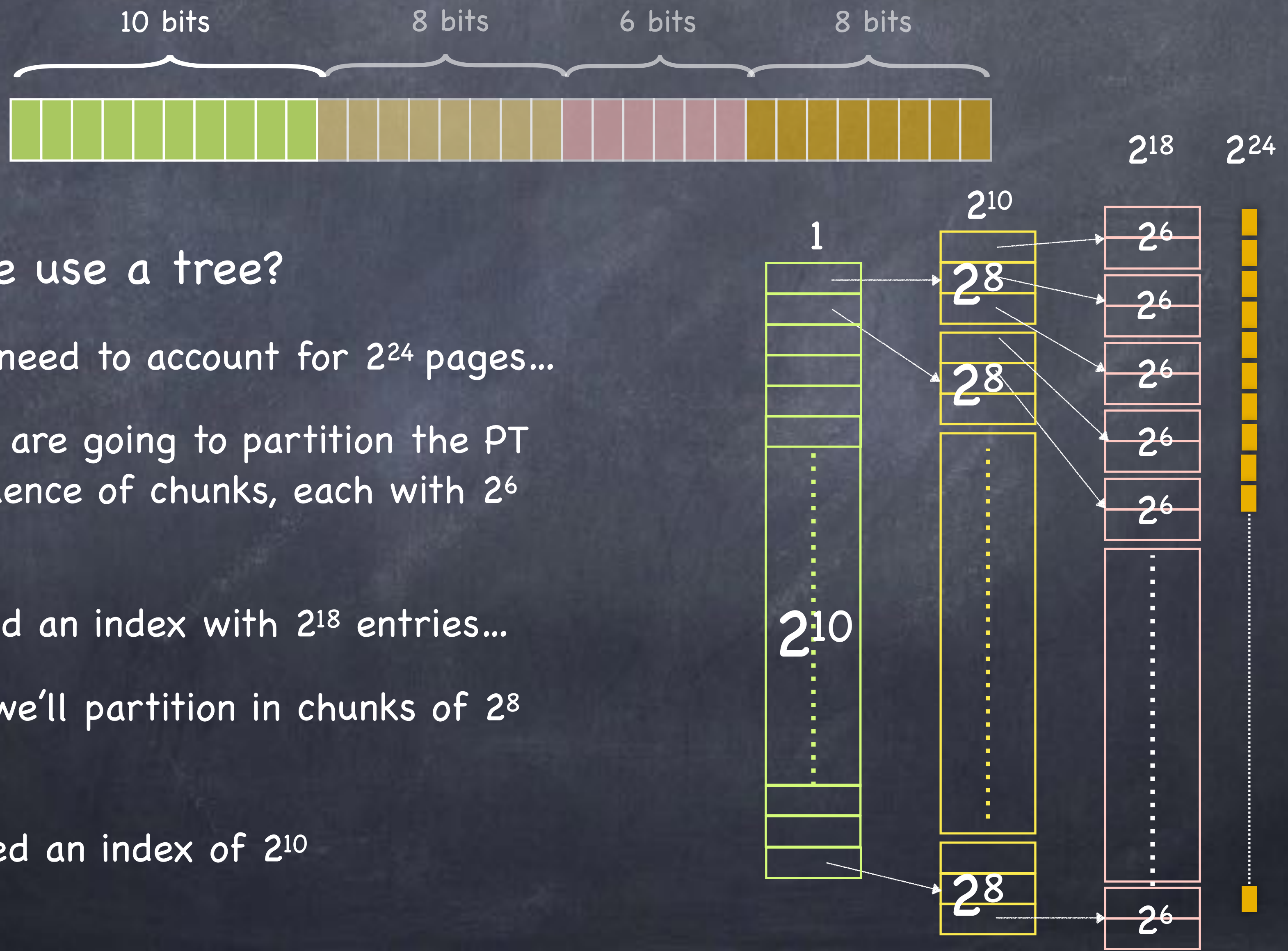
Example



- What is we use a tree?
  - We still need to account for 2^{24} 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^{18} entries…
  - …which we'll partition in chunks of 2^8 entries
  - We'll need an index of 2^{10}

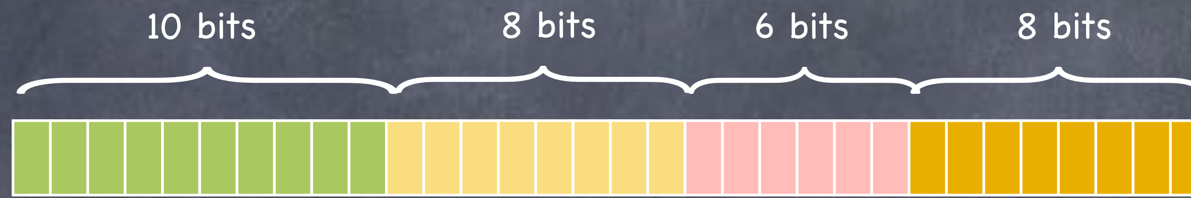


Example

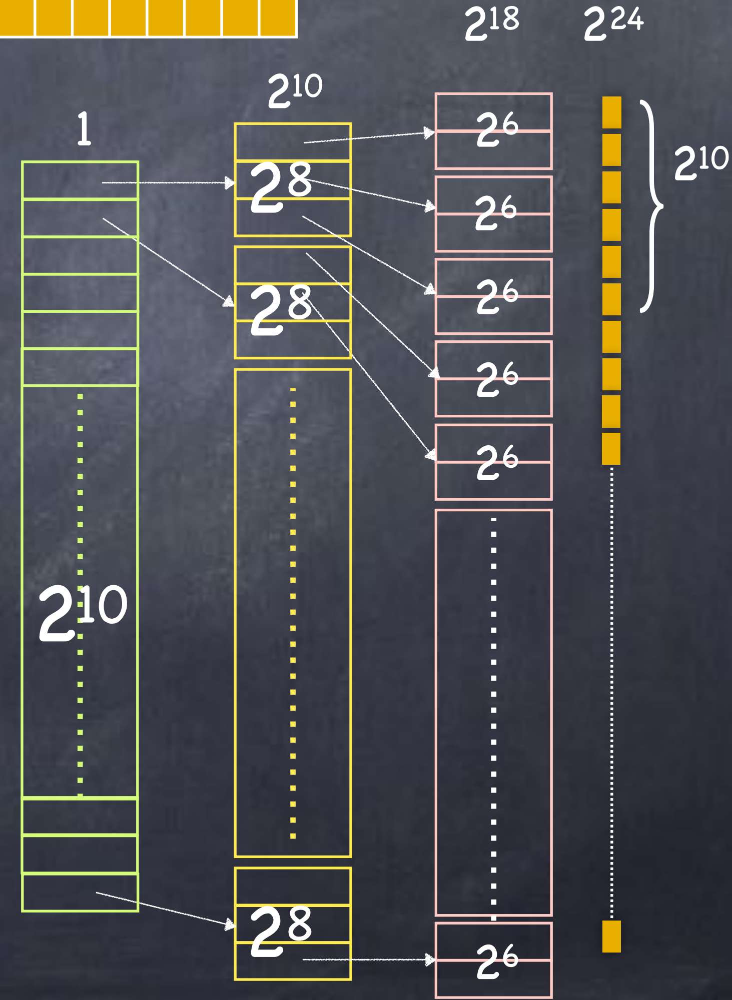


- What is we use a tree?
  - We still need to account for 2^{24} pages...
  - ...but we are going to partition the PT in a sequence of chunks, each with 2^{6} entries
  - we'll need an index with 2^{18} entries...
  - ...which we'll partition in chunks of 2^{8} entries
  - We'll need an index of 2^{10}

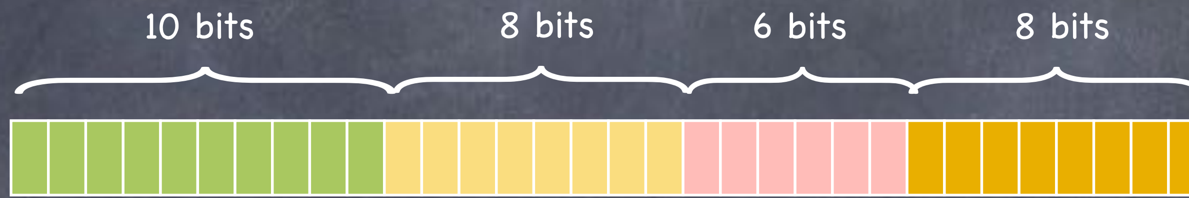
Example



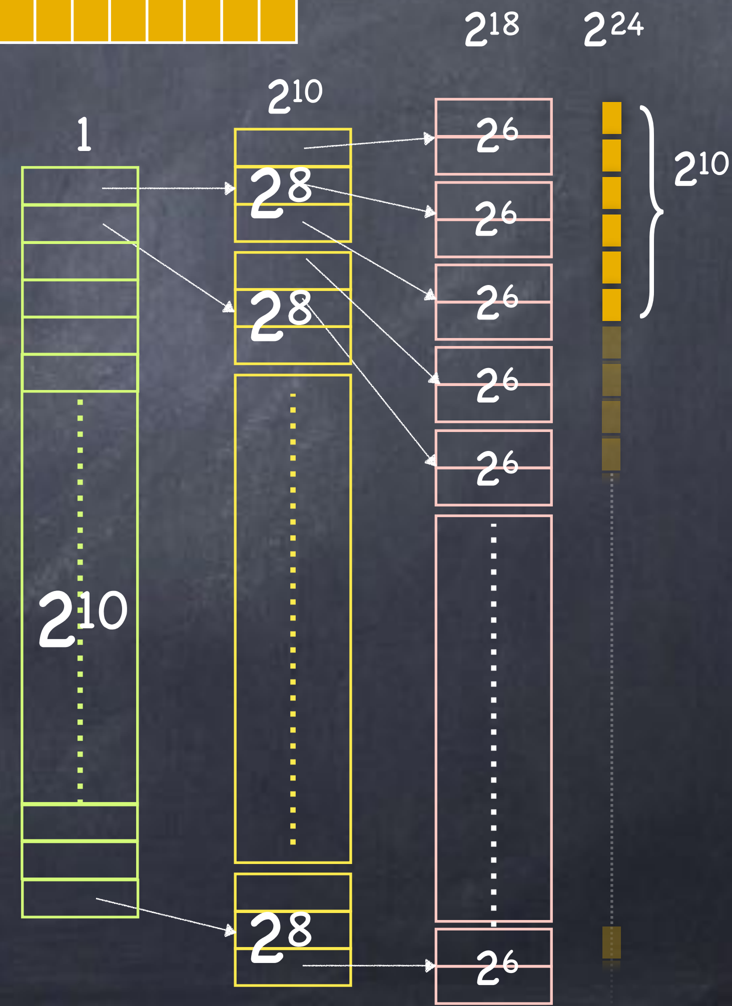
- Are we better off?
  - The number of PT entries now is $(2^6 \times 2^{18}) + (2^{10} \times 2^8) + 2^{10} > 2^{24}$!!
  - But we only need the portion of the tree needed to map the first 1K (2^{10}) pages!



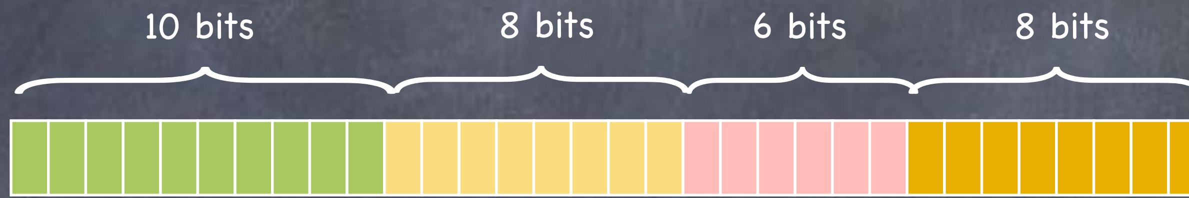
Example



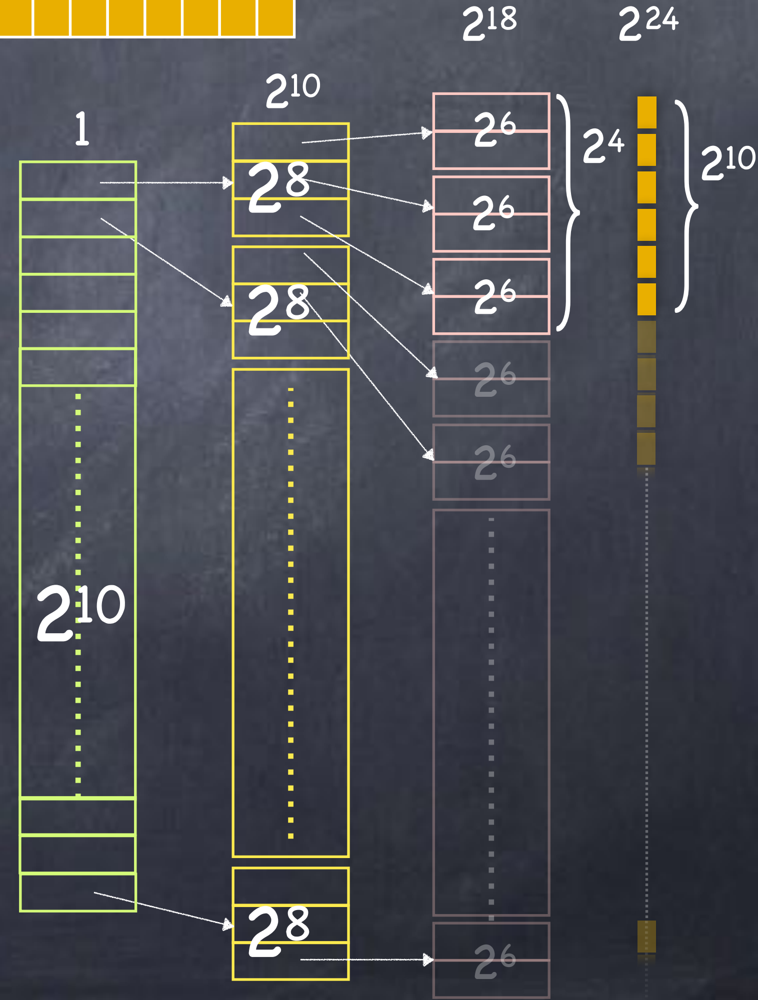
- How many chunks of size 2^6 are needed to hold 2^{10} PTEs of consecutive pages starting at 0?
  - $2^{10}/2^6 = 2^4 = 16$



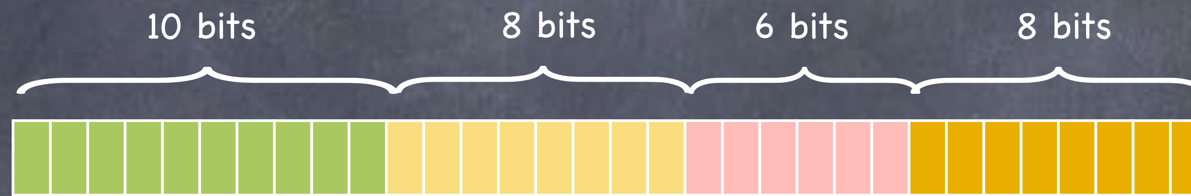
Example



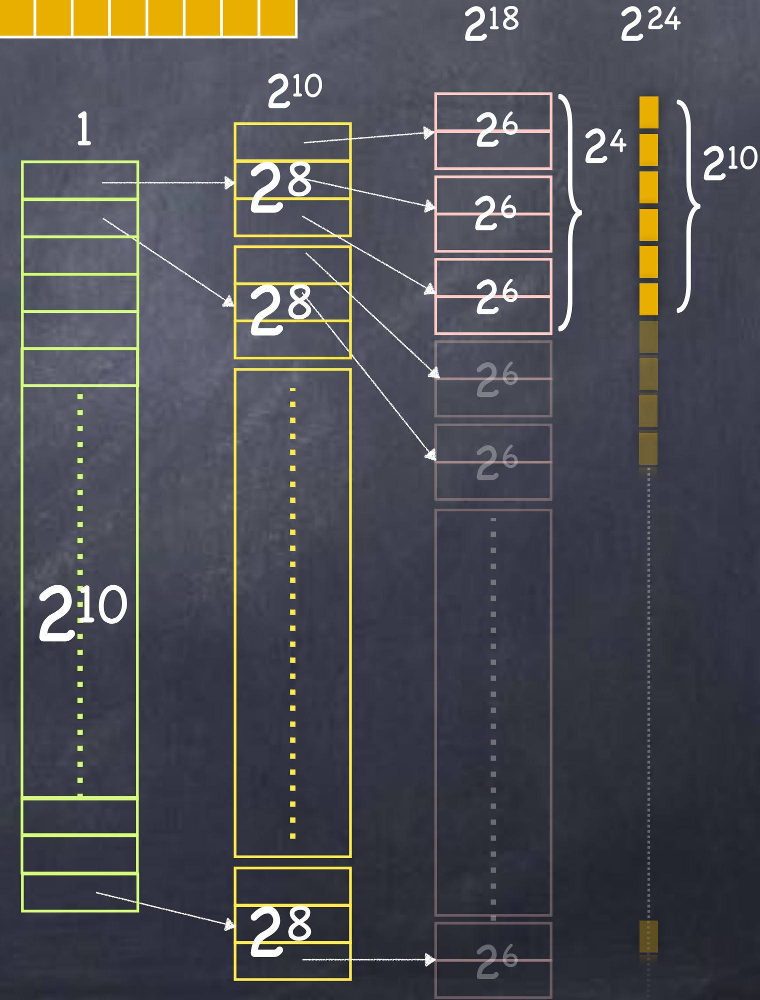
- How many chunks of size 2^6 are needed to hold 2^{10} PTEs of consecutive pages starting at 0?
  - $2^{10}/2^6 = 2^4 = 16$



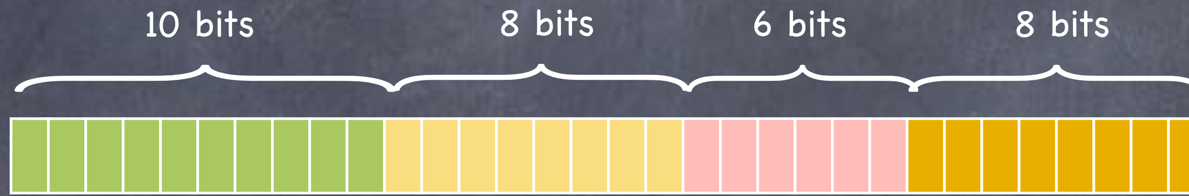
Example



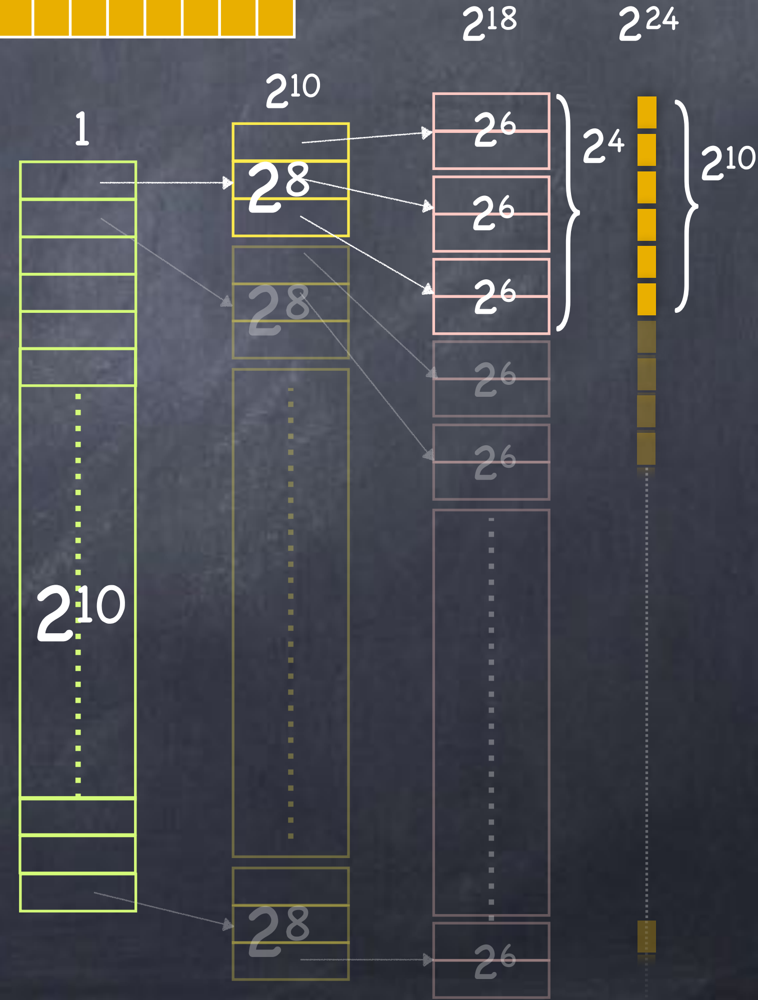
- How many chunks of size 2^6 are needed to hold 2^{10} PTEs of consecutive pages starting at 0?
  - $2^{10}/2^6 = 2^4 = 16$
- How many chunks of size 2^8 are needed to hold pointers to 16 pink chunks?
  - 1



Example



- How many chunks of size 2^6 are needed to hold 2^{10} PTEs of consecutive pages starting at 0?
  - $2^{10}/2^6 = 2^4 = 16$
- How many chunks of size 2^8 are needed to hold pointers to 16 pink chunks?
  - 1
- So, if each PTE is 2 bytes, the PT takes
  - 2 x (1 x 1024 + 1 x 256 + 16 x 64) = 4608 bytes



Getting slooower

- Every new level of paging
  - reduces the memory overhead for computing the mapping function...
  - ... but increases the time necessary to perform the mapping function