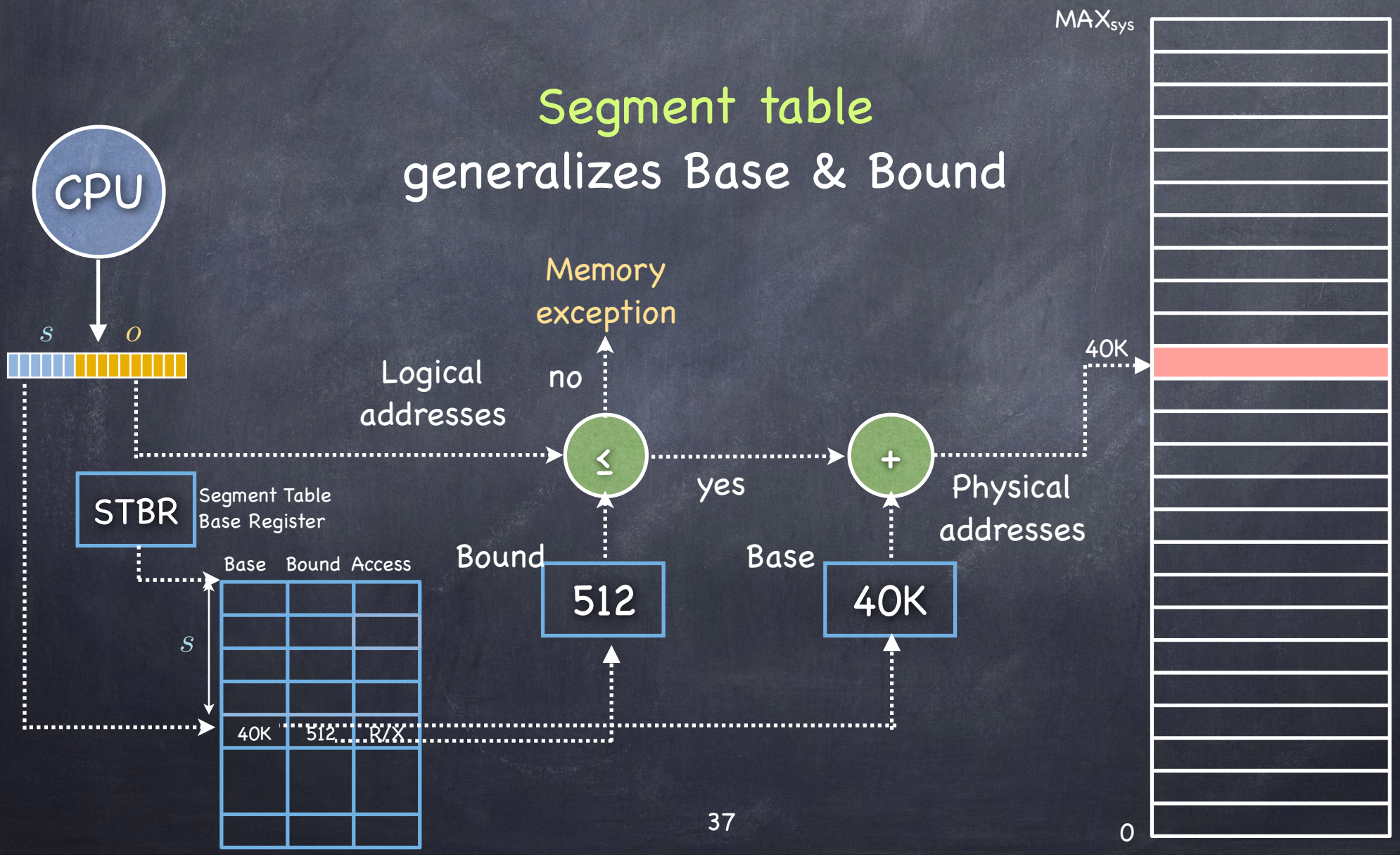
Segment Table

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

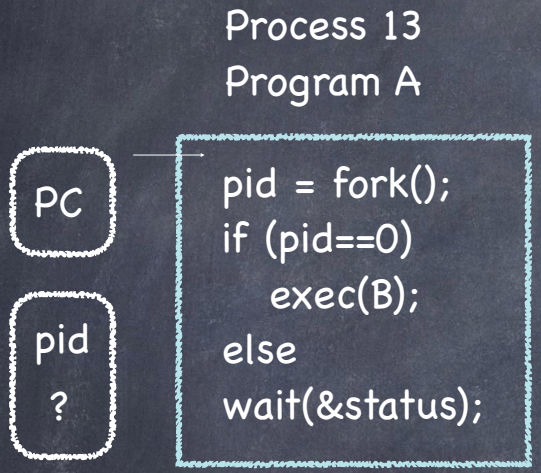
	Base	Bound (Max 4k)	Access
$Code_{00}$	32K	2K	Read/Execute
$Heap_{01}$	34K	3K	Read/Write
$Stack_{10}$	28K	3K	Read/Write

- 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



Revisiting fork()



Revisiting fork()

Process 13
Program A

PC

pid
?

```
pid = fork();
if (pid==0)
    exec(B);
else
wait(&status);
```

Process 13
Program A

PC

pid
14

```
pid = fork();
if (pid==0)
    exec(B);
else
wait(&status);
```

Process 14
Program B

PC

pid
0

```
main() {
pid = fork();
if (pid==0)
    ...
    exec(B);
    exit(3);
else
}
wait(&status);
```

Revisiting fork()

Process 13
Program A

PC

pid
?

```
pid = fork();
if (pid==0)
    exec(B);
else
wait(&status);
```

Process 13
Program A

PC

pid
14

```
pid = fork();
if (pid==0)
    exec(B);
else
wait(&status);
```

Status
3

Process 14
Program B

PC

```
main() {
    ...
  exit(3);
}
```

Revisiting fork()

- Copying an entire address space can be costly…
  - especially if you proceed to obliterate it right away with exec()!

Revisiting fork(): Segments to the Rescue

- Instead of copying entire address space, copy just segment table (the VA->PA mapping)

	Base	Bound	Access
Code	32K	2K	RX
Heap	34K	3K	RW
Stack	28K	3K	RW

Parent

	Base	Bound	Access
Code	32K	2K	RX
Heap	34K	3K	RW
Stack	28K	3K	RW

Child

- but change all writeable segments to Read only

Revisiting fork(): Segments to the Rescue

- Instead of copying entire address space, copy just segment table (the VA->PA mapping)

	Base	Bound	Access
Code	32K	2K	RX
Heap	34K	3K	R
Stack	28K	3K	R

Parent

	Base	Bound	Access
Code	32K	2K	RX
Heap	34K	3K	R
Stack	28K	3K	R

Child

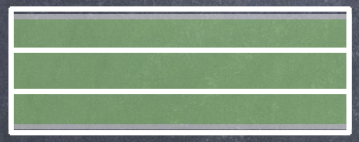
- but change all writeable segments to Read only
- Segments in VA spaces of parent and child point to same locations in physical memory

Copy on Write (COW)

- When trying to modify an address in a COW segment:
  - exception!
    - exception handler copies just the affected segment, and changes both the old and new segment back to writeable
- If exec() is immediately called, only stack segment is copied!
  - it stores the return value of the fork() call, which is different for parent and child

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 (think "assigning classrooms to courses")
  - 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

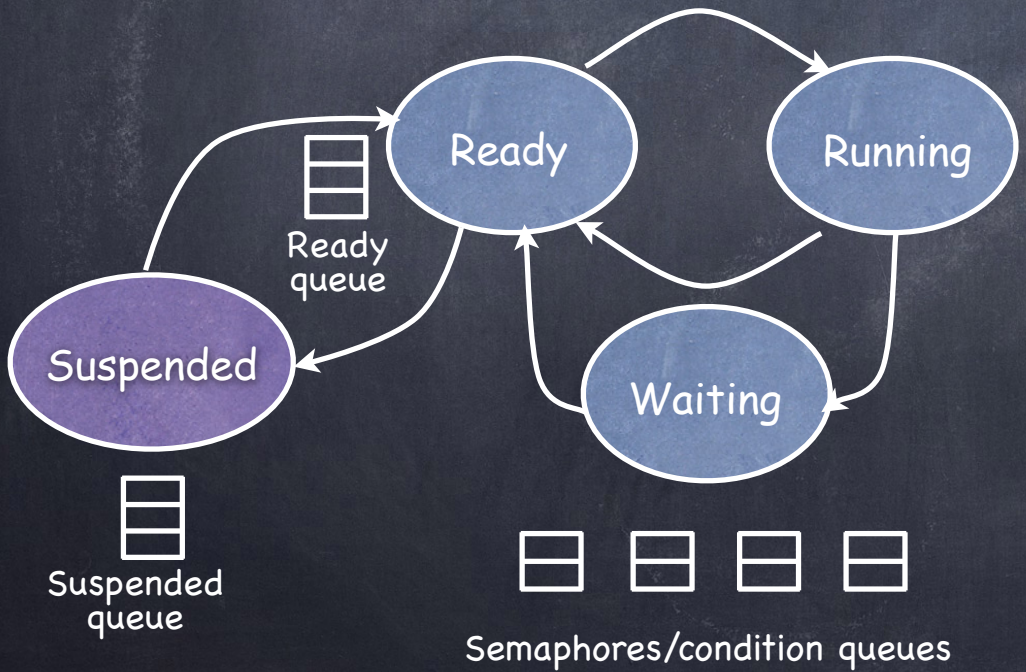
- Over time, memory can become full of small holes
  - Hard to fit more segments
  - Hard to expand existing ones
- Compaction
  - Relocate segments to coalesce holes

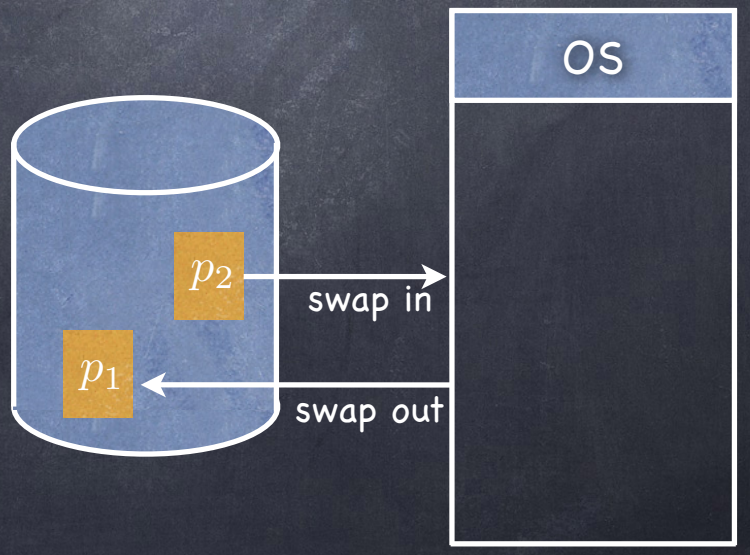
External Fragmentation

- 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

- Preempt processes and reclaim their memory
- Move images of suspended processes to backing store

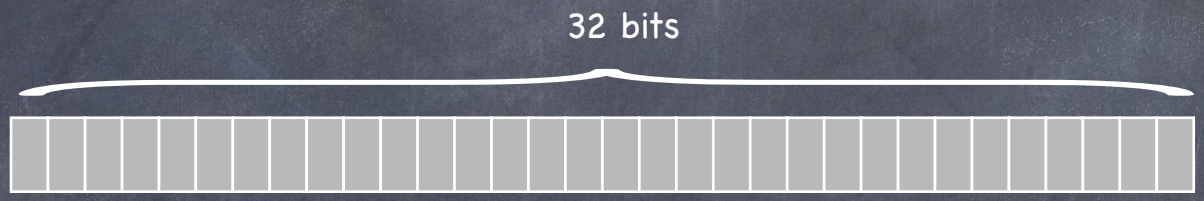




Paging

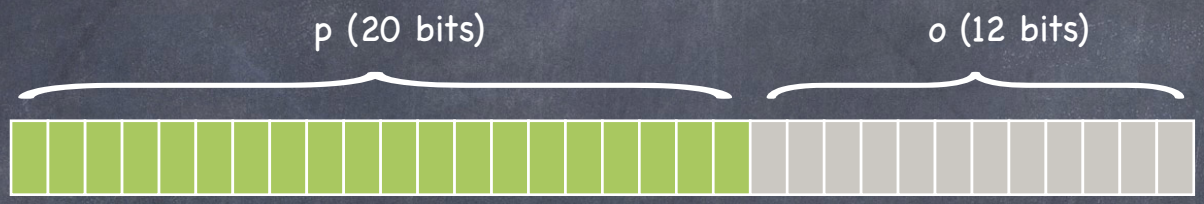
- Allocate VA & PA memory in chunks of the same, fixed size (pages and frames, respectively)
- Adjacent pages in VA (say, within the stack) need not map to contiguous frames in PA!
  - free frames can be tracked using a simple bitmap
    - 00111110011110111110000 one bit/frame
  - no more external fragmentation!
  - 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



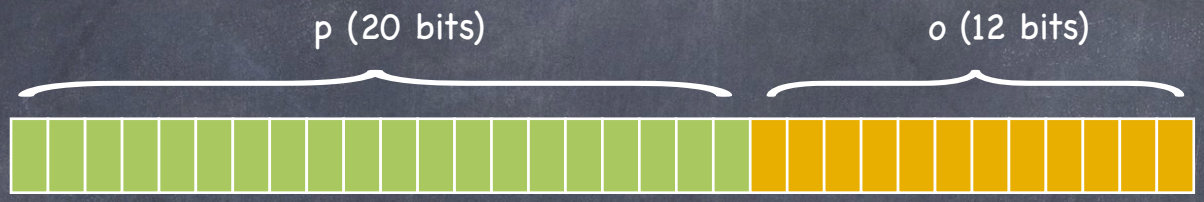
- Interpret VA as comprised of two components
  - page: which page?
  - offset: which byte within that page?

Virtual address



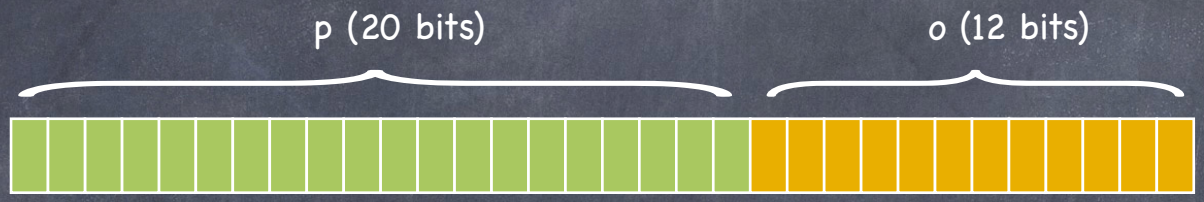
- Interpret VA as comprised of two components
  - page: which page?
    - no. of bits specifies no. of pages are in the VA space
  - offset: which byte within that page?

Virtual address

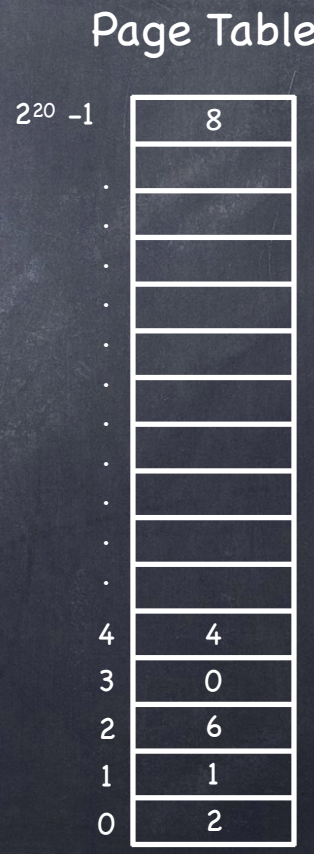


- Interpret VA as comprised of two components
  - page: which page?
    - no. of bits specifies no. of pages are in the VA space
  - offset: which byte within that page?
    - no. of bits specifies size of page/frame

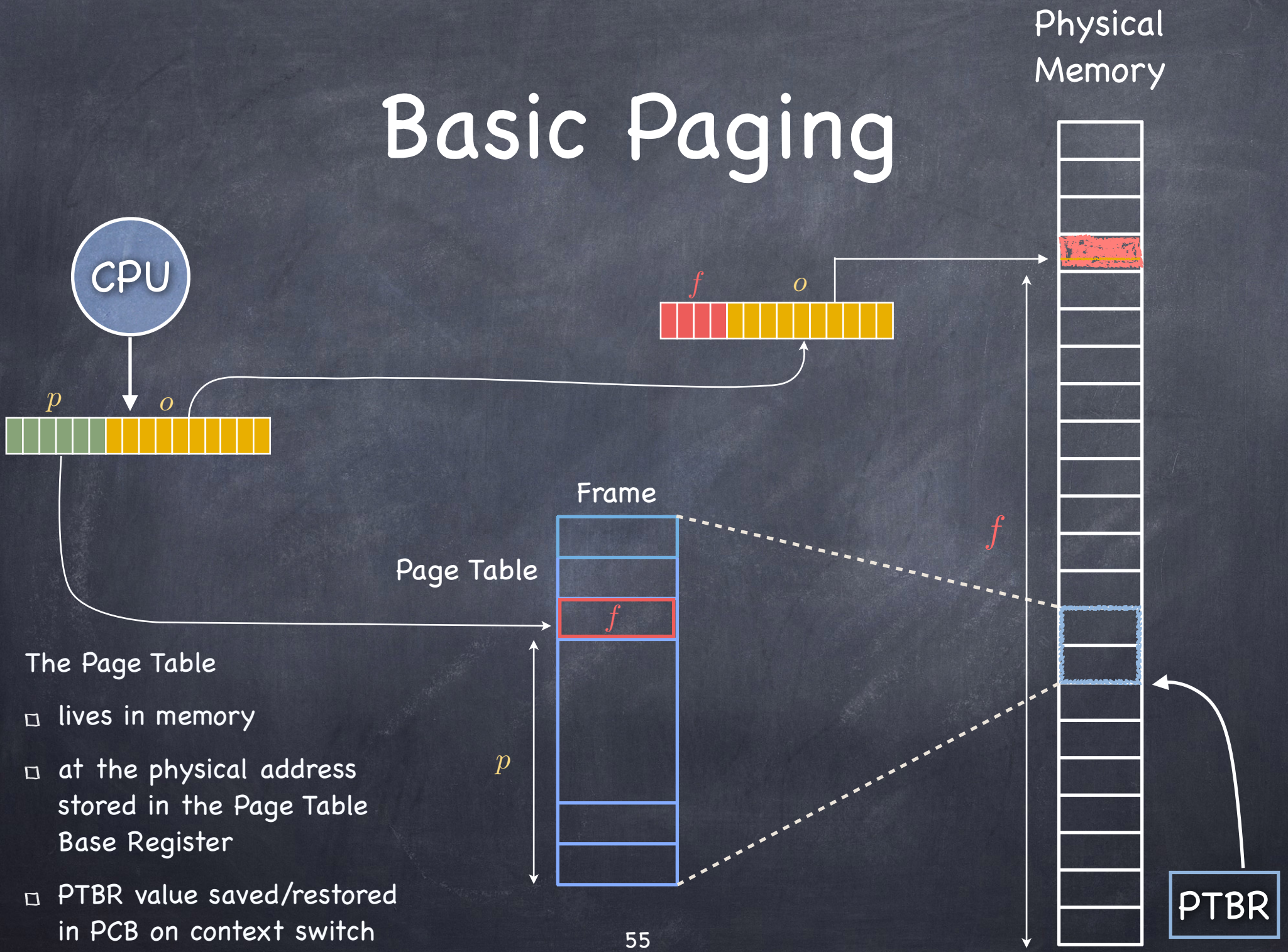
Virtual address



- 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



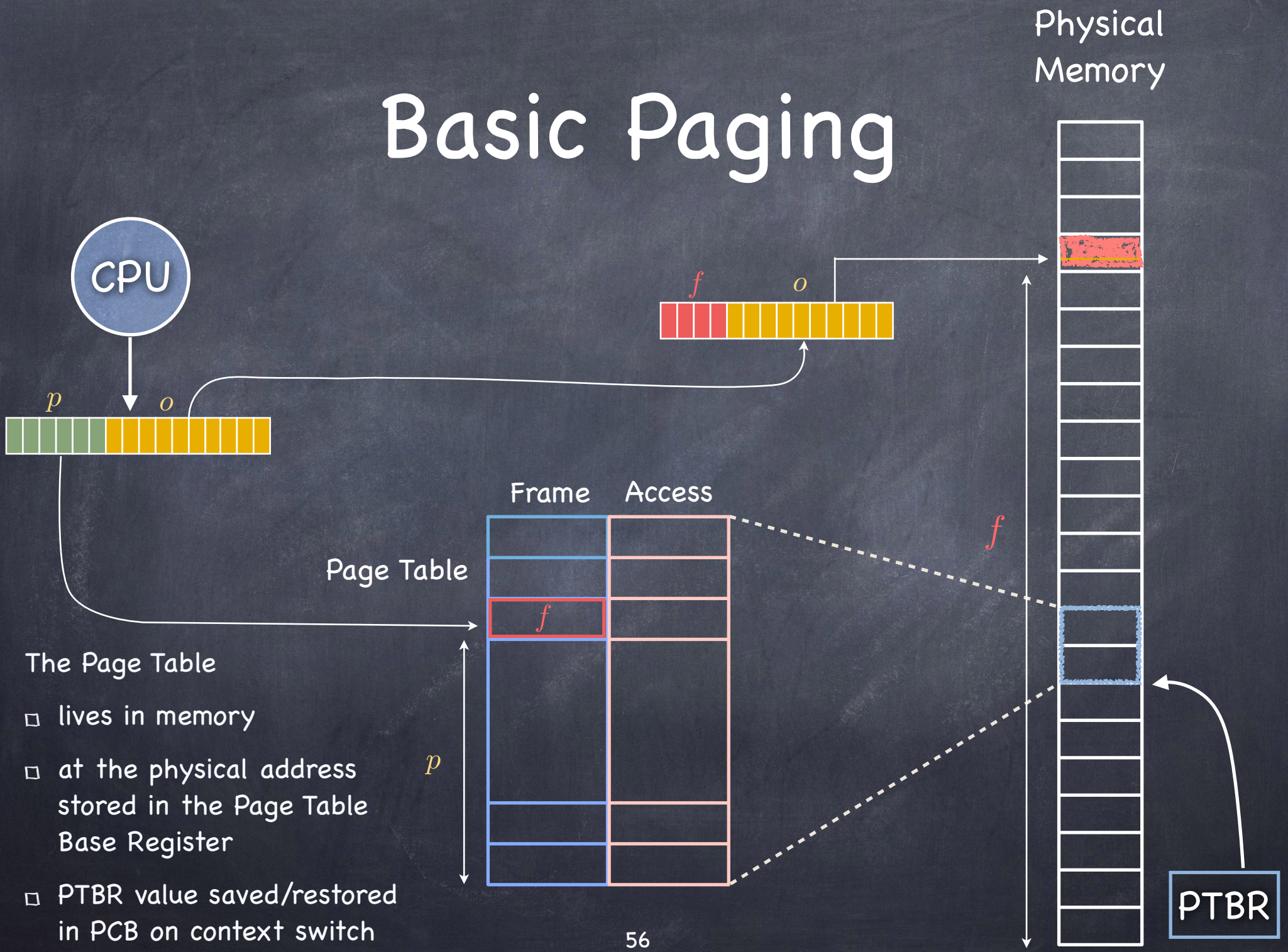
Basic Paging



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

Basic Paging

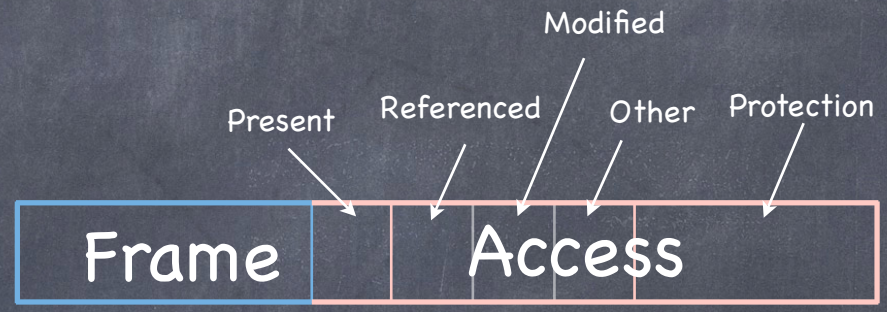


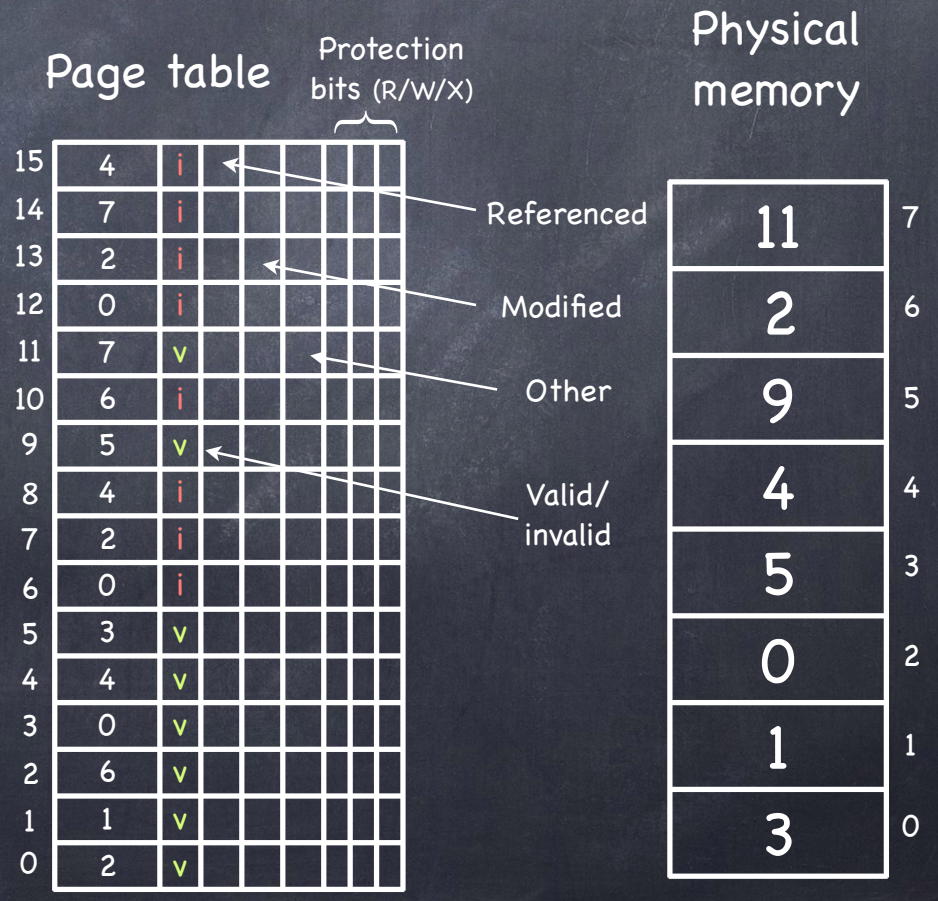
The Page Table

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

Page Table Entries

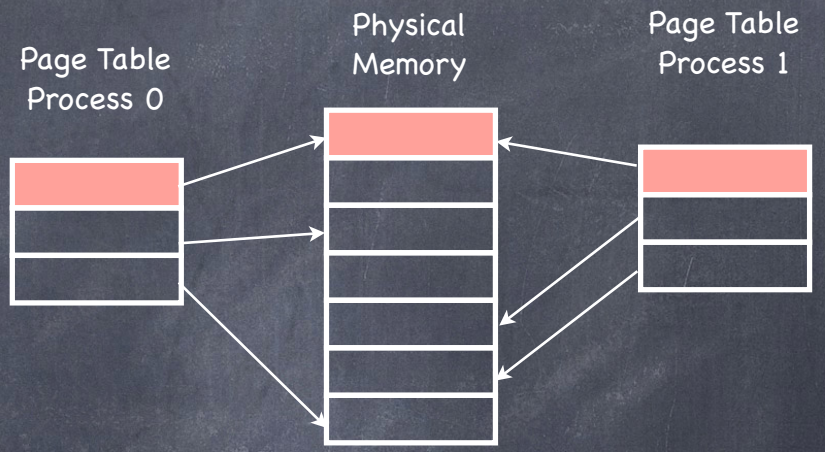
- Frame number
- Valid/Invalid (Present) bit
  - Set if entry stores a valid mapping. If not, and accessed, page fault
- Referenced bit
  - Set if page has been referenced
- Modified bit
  - Set if page has been modified
- Protection bits (R/W/X)





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)
- We can refine COW to operate at the granularity of pages
  - on fork, mark all pages in page table Read only

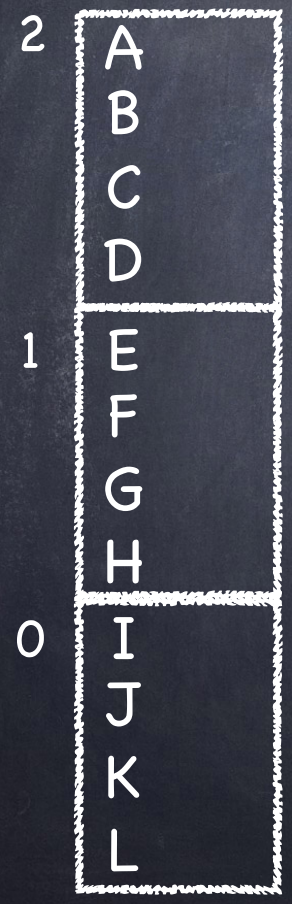


  - on write:
    - page fault
    - allocate new frame
    - copy page
    - mark both pages R/W

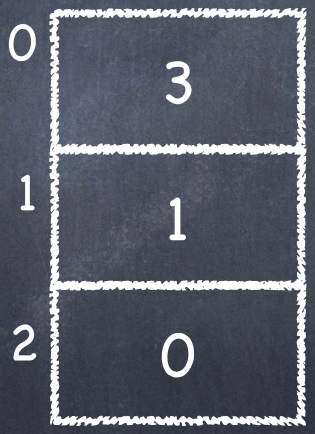
Example

Page size: 4 bytes

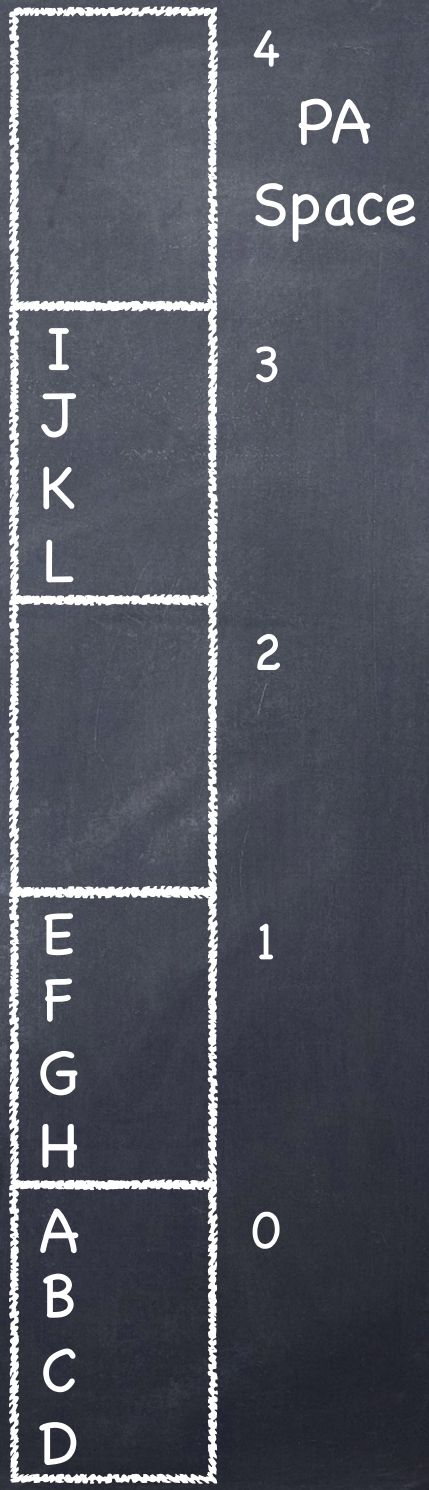
VA Space



Page Table



PA Space



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)
  - $((2^{20} / 2^{10}) \times \text{sizeofEntry}) + (3 \times 2^9)$
  - If I know PA is 64 KB then sizeofEntry = sizeofFrameNo + #ofAccessBits= 6 (2^6 frames) + #ofAccessBits
    - if 7 access bits, byte aligned size of entry: 16 bits

What's not to love?

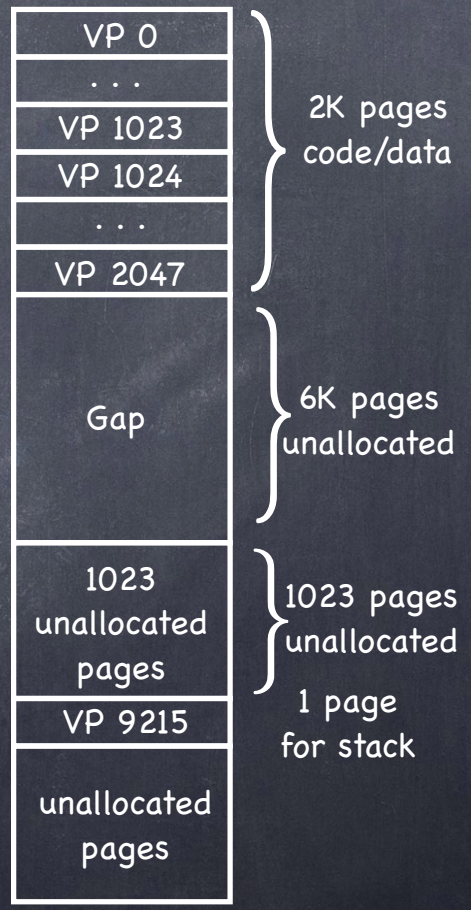
- Space overhead
  - With a 64-bit address space, size of page table can be huge
- Time overhead
  - Accessing data now requires two memory accesses
    - must also access page table, to find mapped frame

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

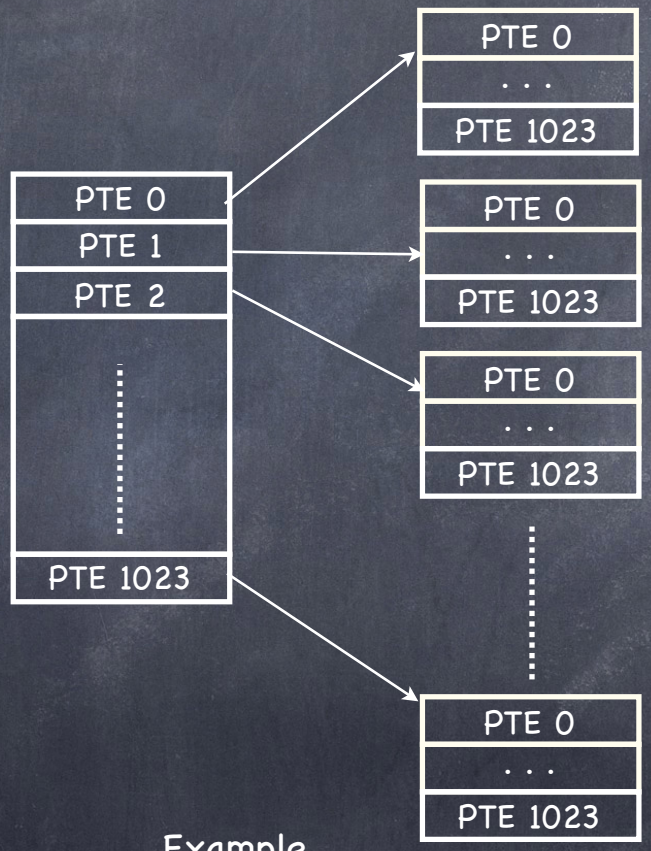
- 32 bit address space
- 4Kb pages
- 4 bytes PTE



Page Table

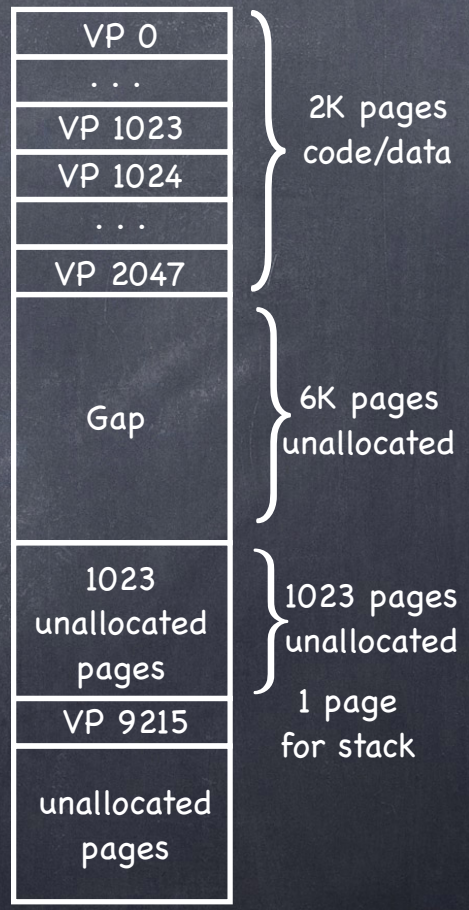
Reducing the Storage Overhead of Page Tables

- 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!



Example

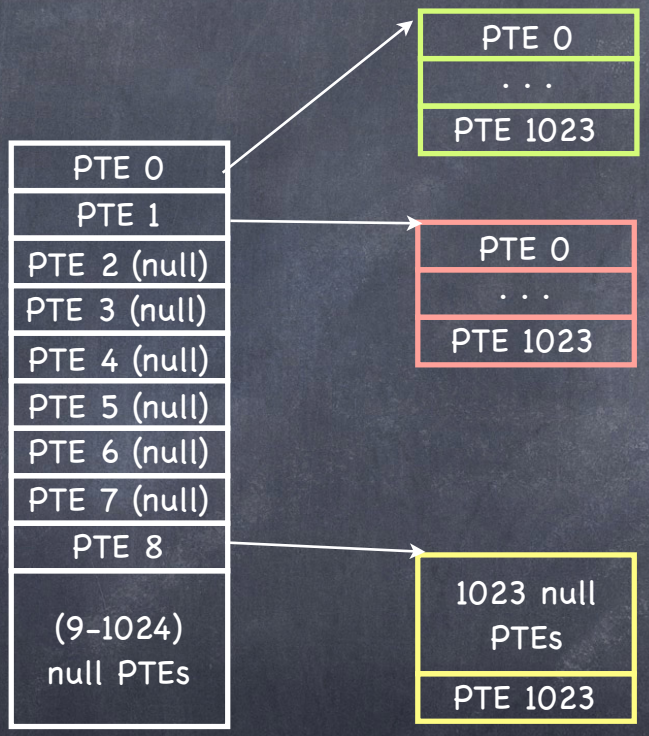
- 32 bit address space
- 4Kb pages
- 4 bytes PTE



Page Table

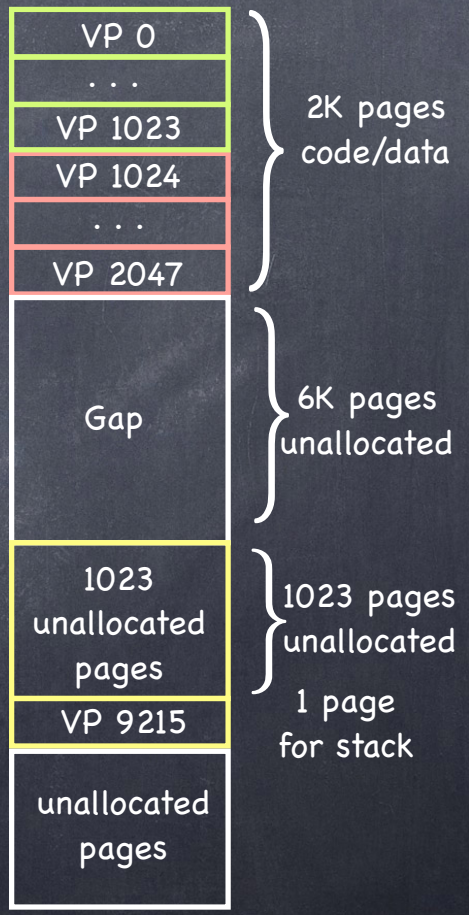
Reducing the Storage Overhead of Page Tables

- 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!



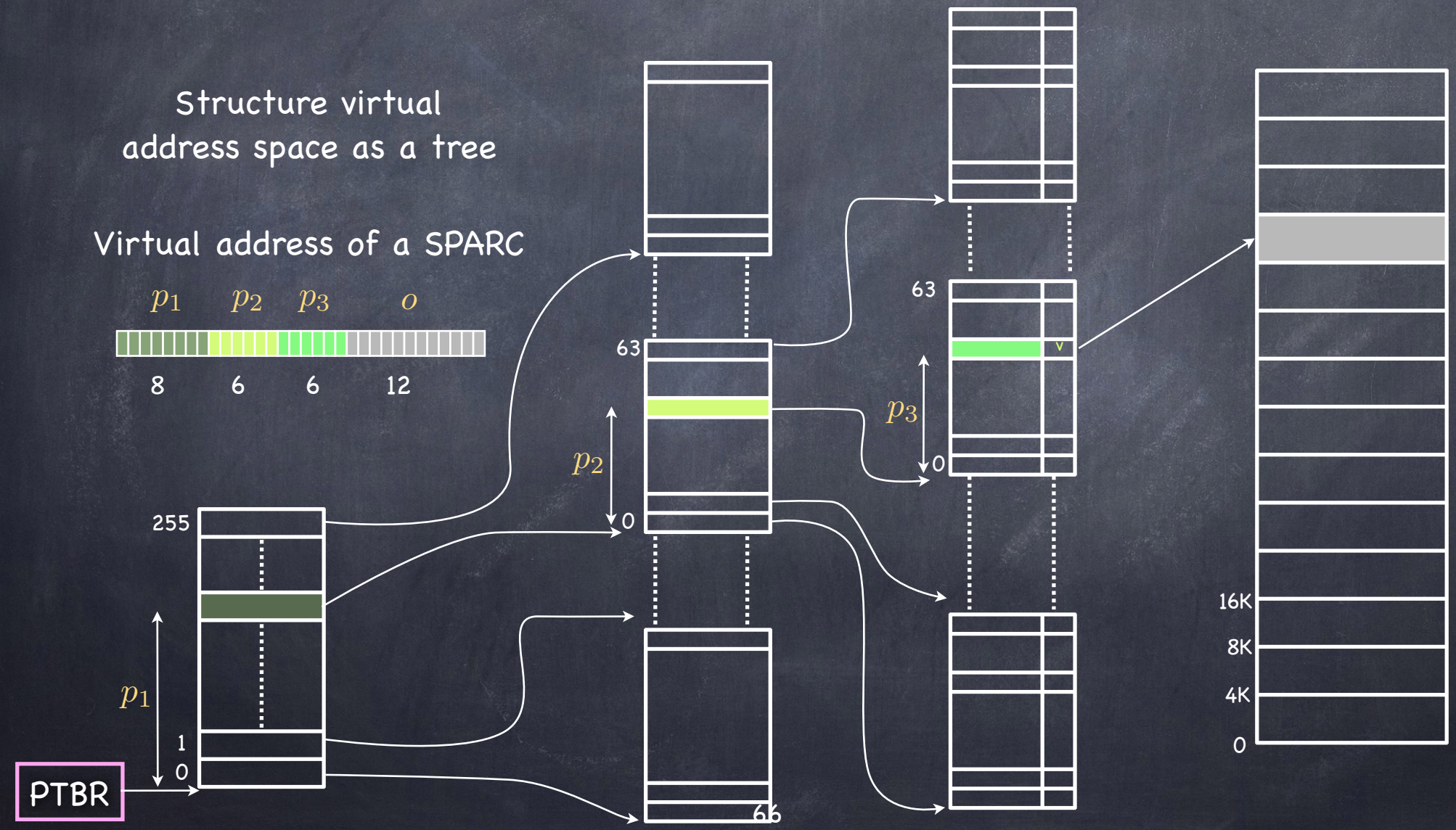
Example

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

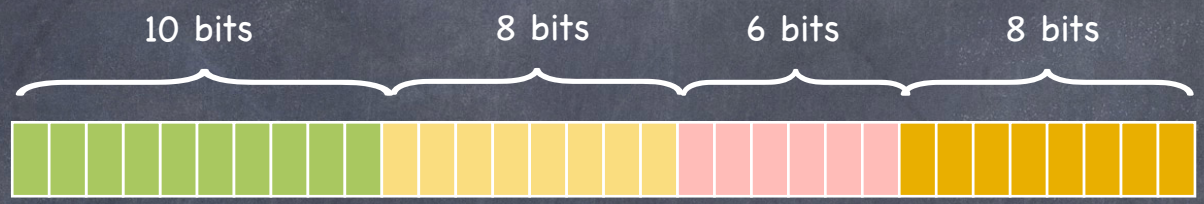


Page Table

Multi-level Paging

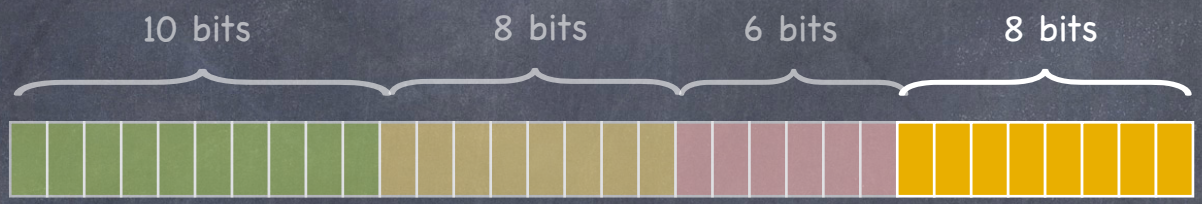


Example



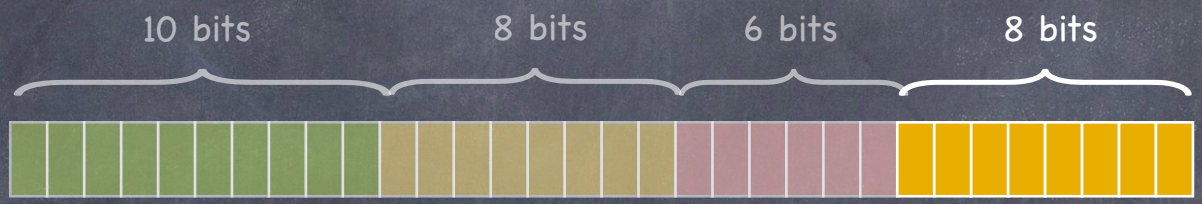
- What is the page size?

Example



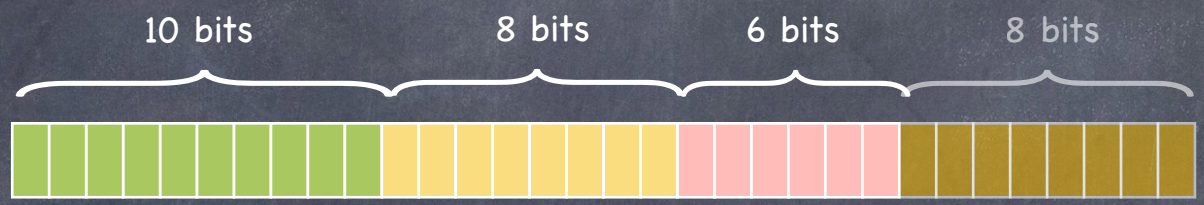
- What is the page size? Page size is 256 bytes (2^8)
- What is the Page Table size for a process that uses 256 contiguous KB of its VAS starting at address 0? [Assume each PTE is 2 bytes]
  - if we used a linear representation of the page table:

Example



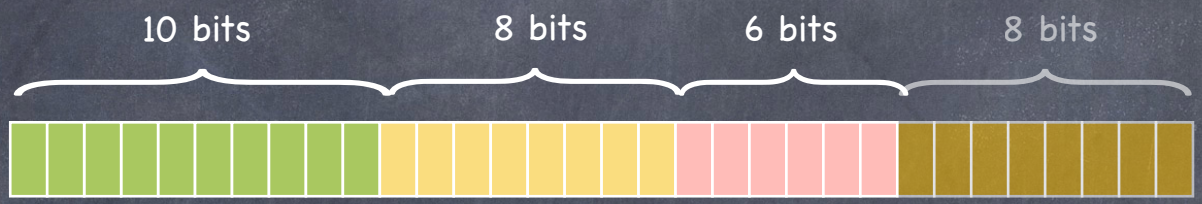
- What is the page size? Page size is 256 bytes (2^8)
- What is the Page Table size for a process that uses 256 contiguous KB of its VAS starting at address 0? [Assume each PTE is 2 bytes]
  - if we used a linear representation of the page table:
    - Page Table has 2^{24} entries

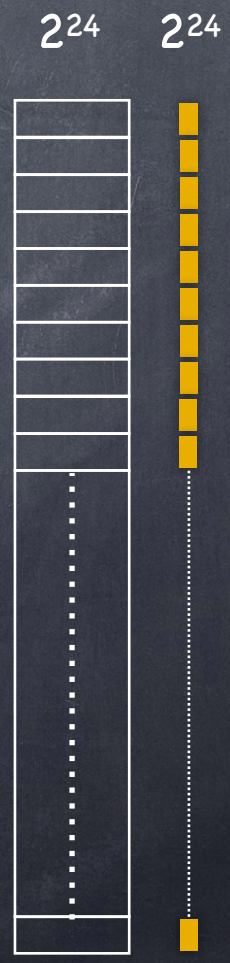
Example



- 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

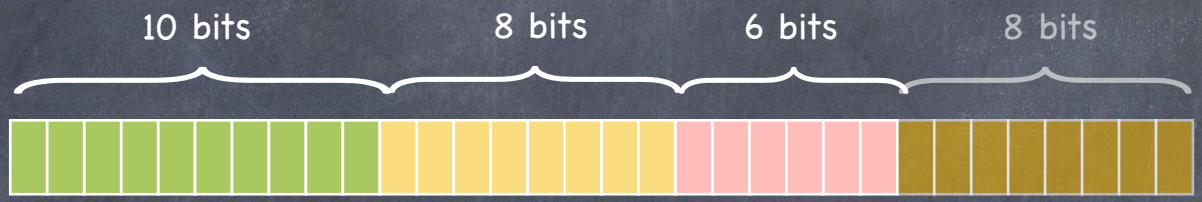
Example



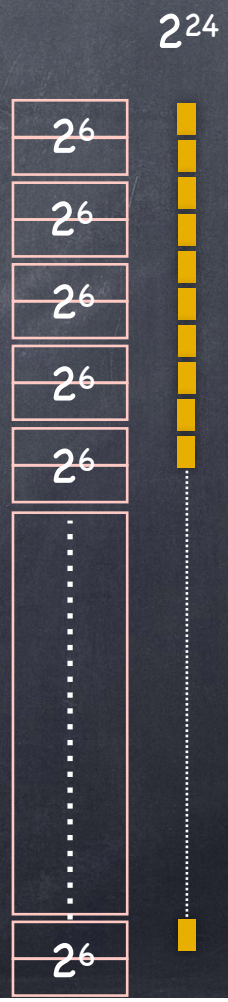


- What is we use a tree?
  - We still need to account for 2^{24} pages...
  - ...but we are going to partition the PT in a sequence of chunks, each with 2^{6} entries

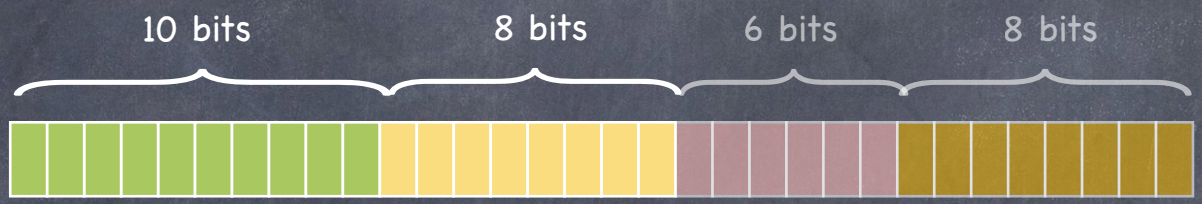
Example



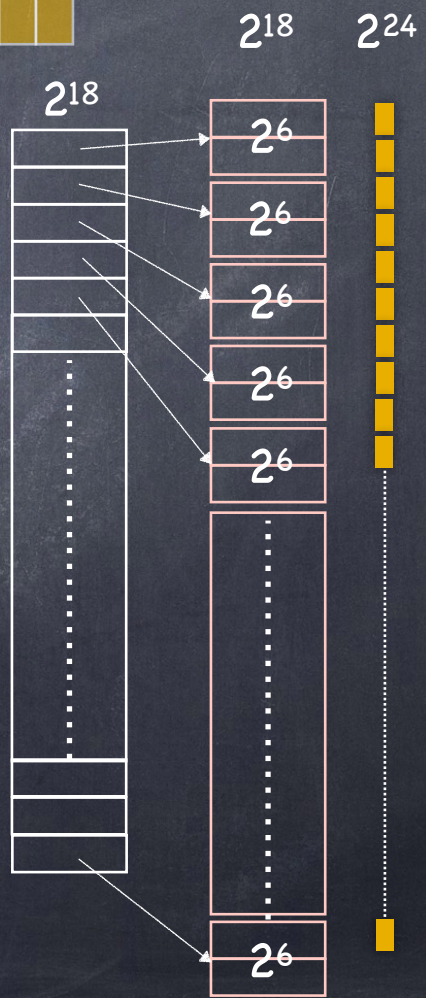
- What is we use a tree?
  - We still need to account for 2^{24} pages...
  - ...but we are going to partition the PT in a sequence of chunks, each with 2^6 entries



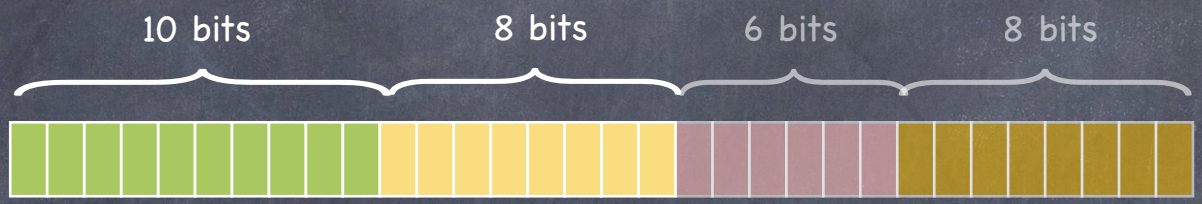
Example



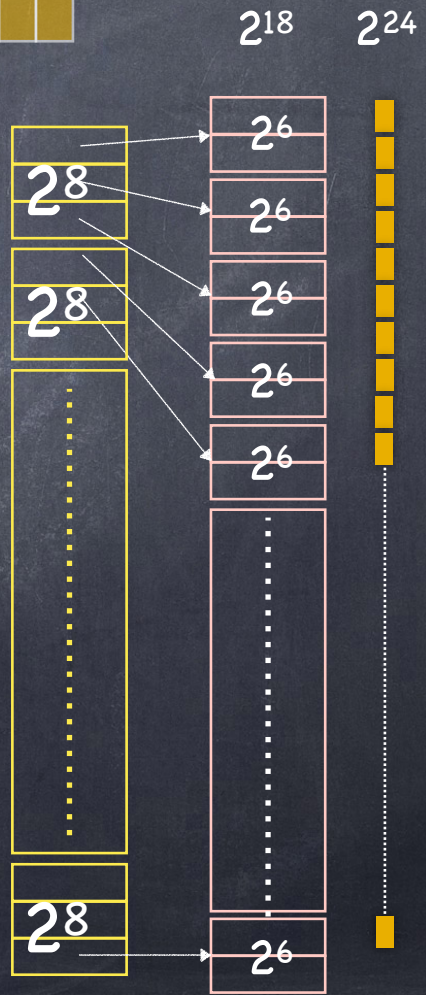
- What is we use a tree?
  - We still need to account for 2^{24} pages...
  - ...but we are going to partition the PT in a sequence of chunks, each with 2^6 entries
  - we'll need an index with 2^{18} entries...
  - ...which we'll partition in chunks of 2^8 entries



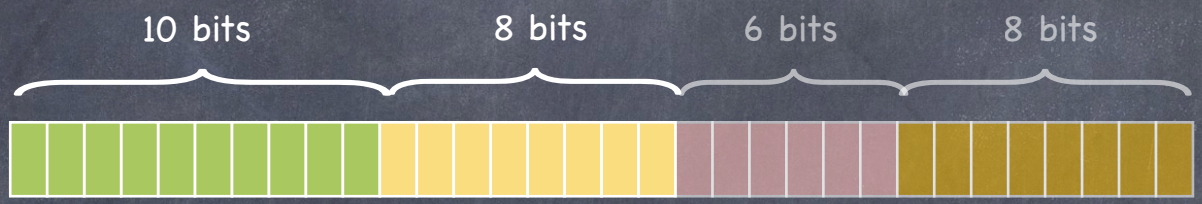
Example



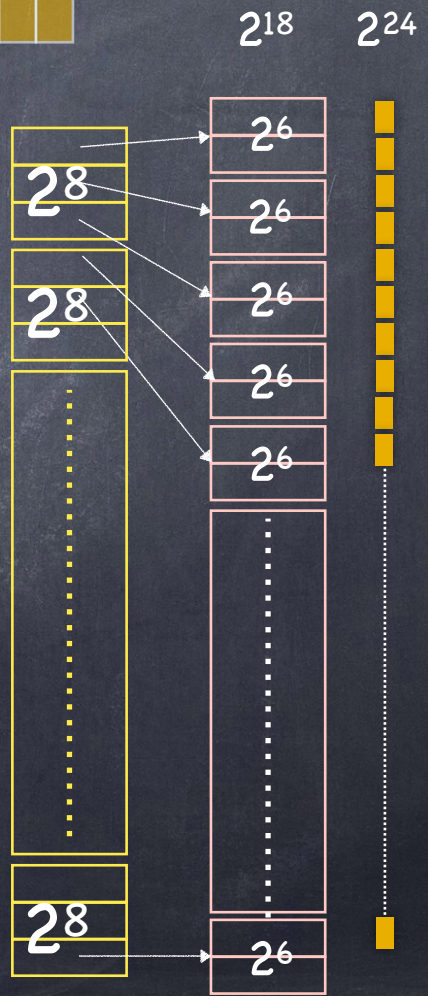
- What is we use a tree?
  - We still need to account for 2^{24} pages...
  - ...but we are going to partition the PT in a sequence of chunks, each with 2^6 entries
  - we'll need an index with 2^{18} entries...
  - ...which we'll partition in chunks of 2^8 entries



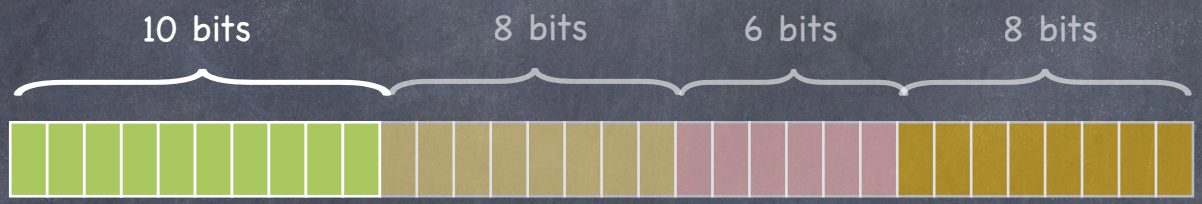
Example



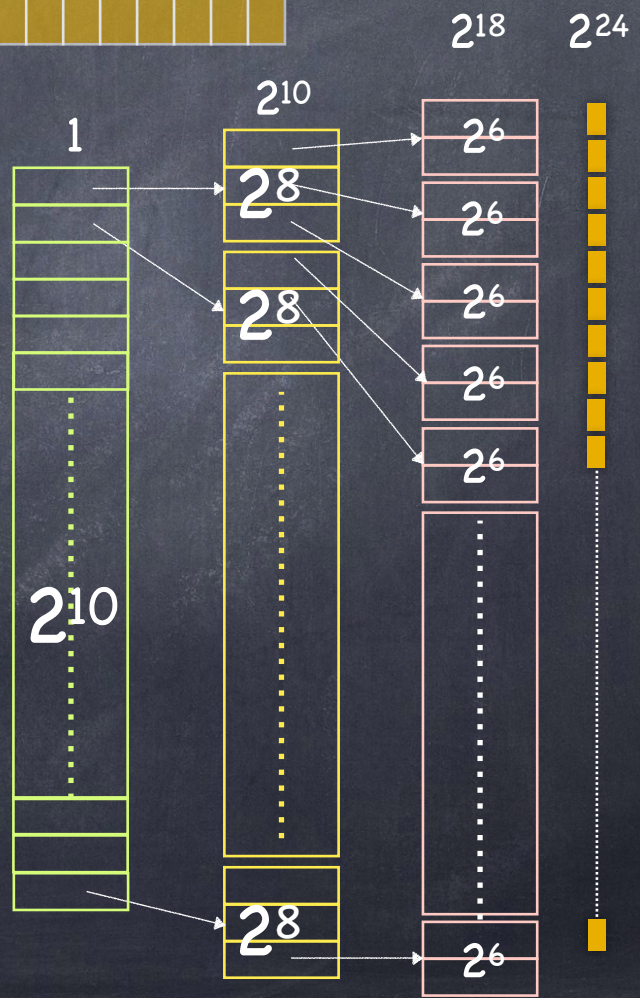
- What is we use a tree?
  - We still need to account for 2^{24} pages...
  - ...but we are going to partition the PT in a sequence of chunks, each with 2^6 entries
  - we'll need an index with 2^{18} entries...
  - ...which we'll partition in chunks of 2^8 entries
  - We'll need an index of 2^{10}



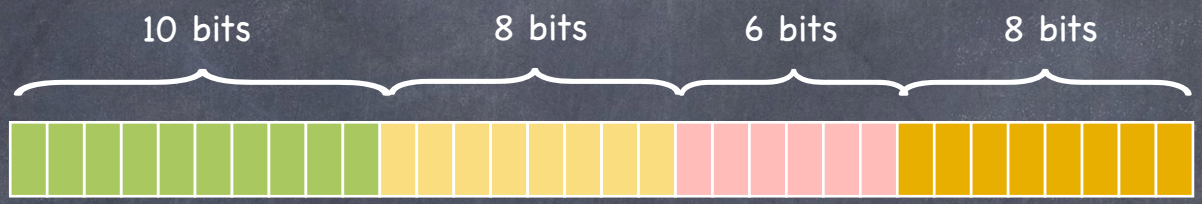
Example



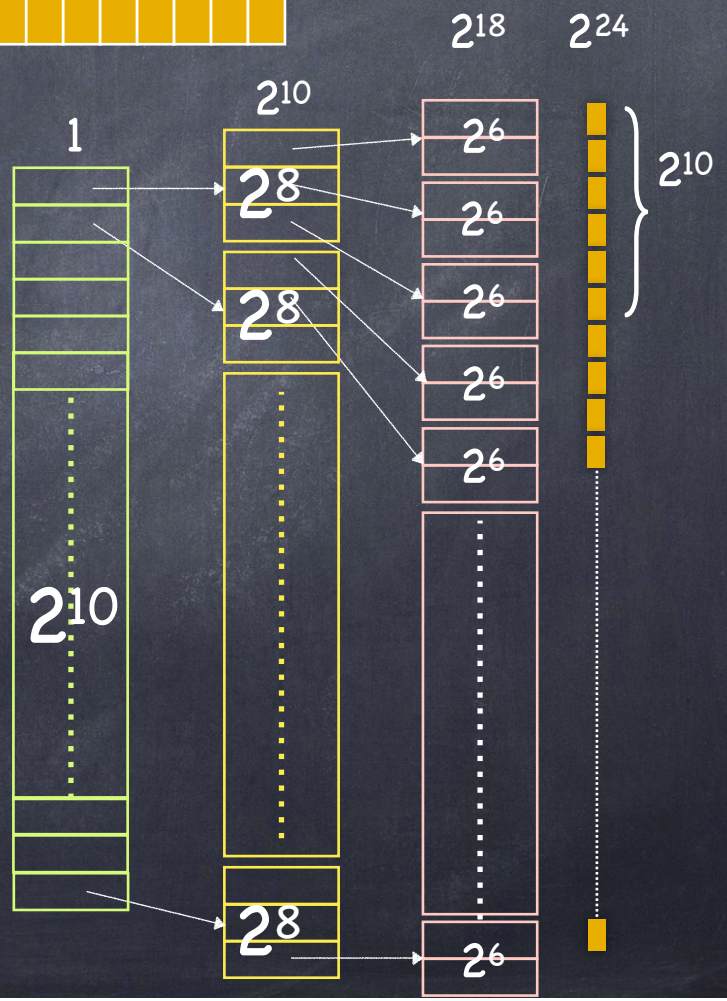
- What is we use a tree?
  - We still need to account for 2^{24} pages...
  - ...but we are going to partition the PT in a sequence of chunks, each with 2^6 entries
  - we'll need an index with 2^{18} entries...
  - ...which we'll partition in chunks of 2^8 entries
  - We'll need an index of 2^{10}



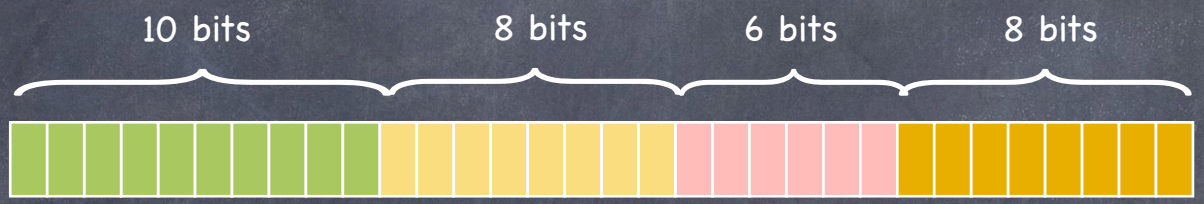
Example



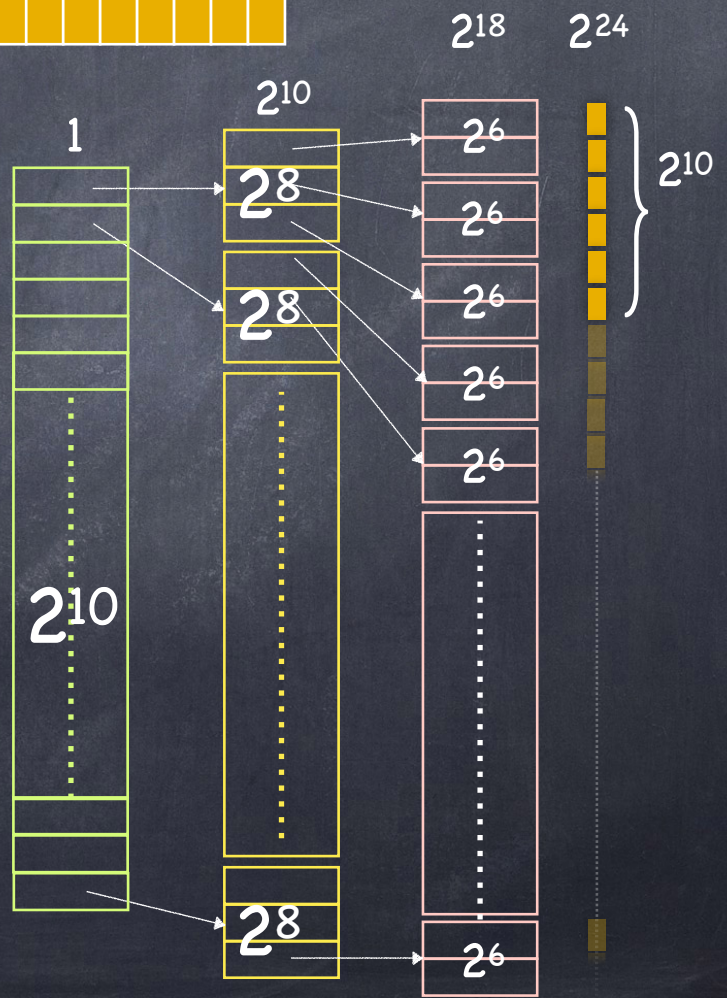
- Are we better off?
  - The number of PT entries now is $(2^6 \times 2^{18}) + (2^{10} \times 2^8) + 2^{10} > 2^{24}$!!
  - But we only need the portion of the tree needed to map the first 1K (2^{10}) pages!



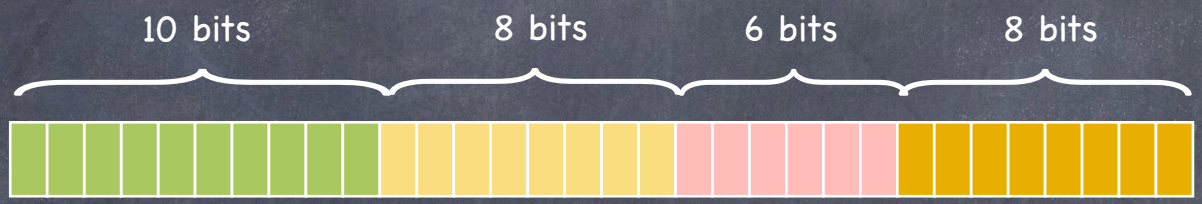
Example



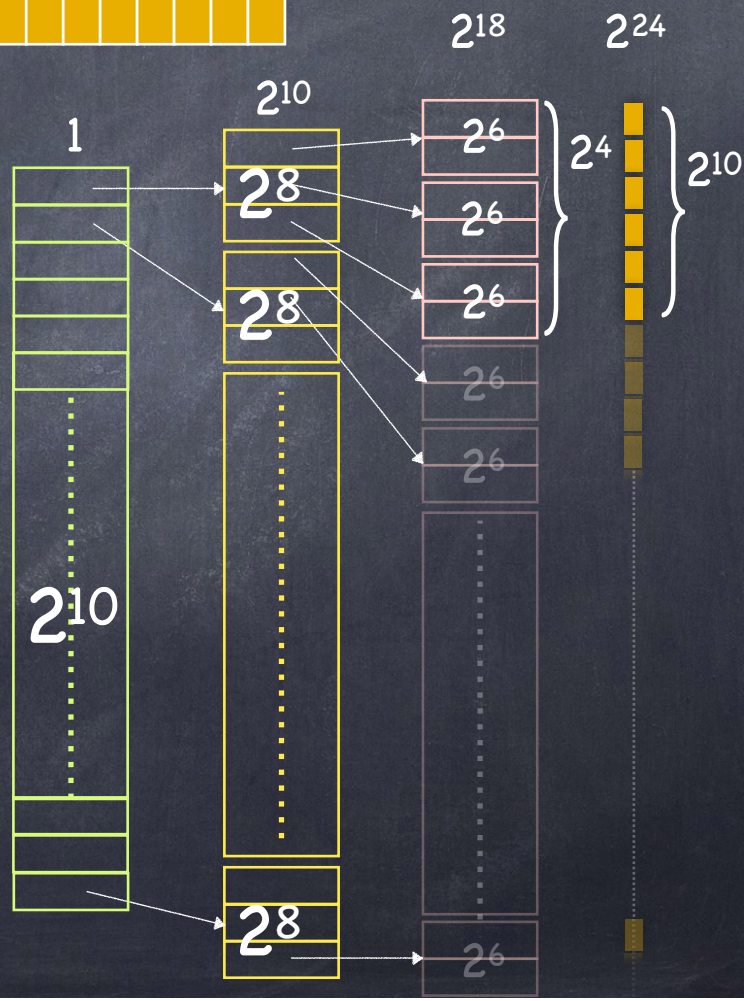
- How many chunks of size 2^6 are needed to hold 2^{10} PTEs of consecutive pages starting at 0?
  - $2^{10}/2^6 = 2^4 = 16$



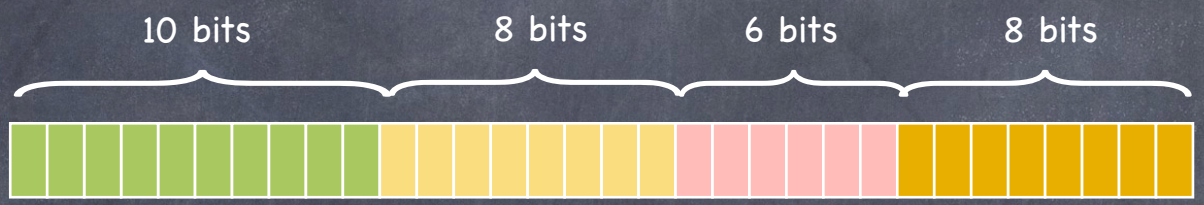
Example



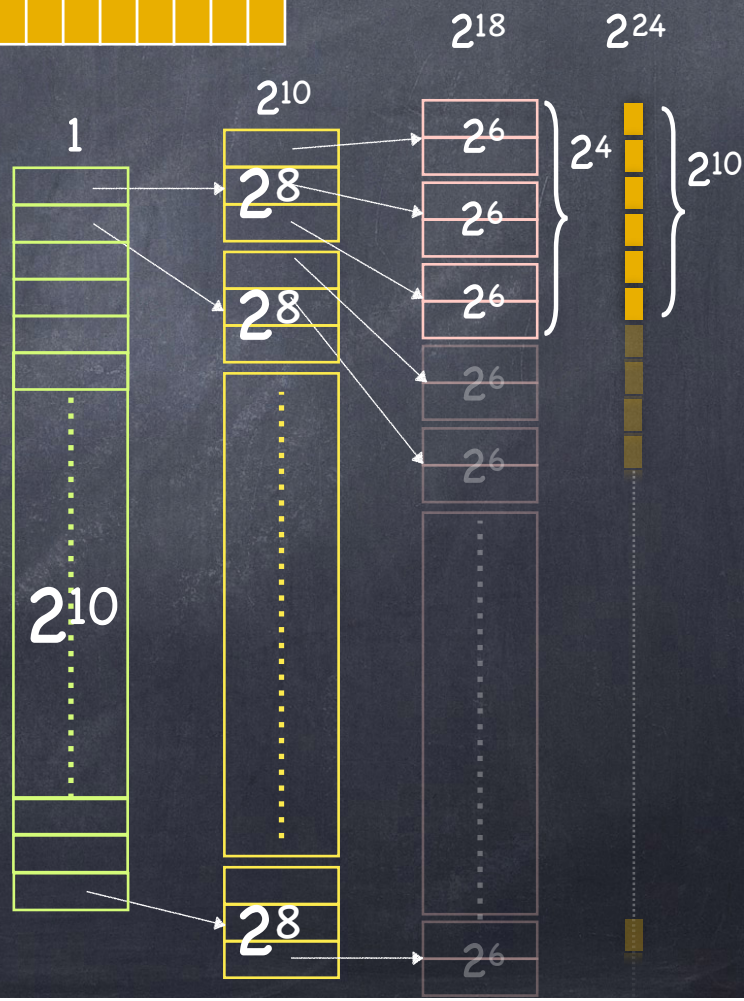
- How many chunks of size 2^6 are needed to hold 2^{10} PTEs of consecutive pages starting at 0?
  - $2^{10}/2^6 = 2^4 = 16$



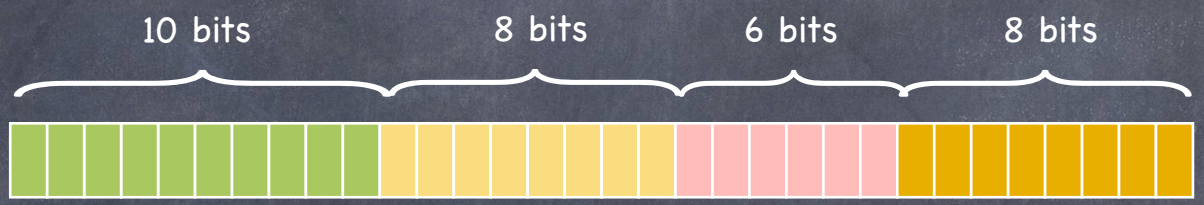
Example



- 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