IT308: Operating Systems

Virtual memory: Paging, TLB

How to fix both internal and external fragmentation?

Today: small fixed-size luggage

Paging

- Instead of dealing with variable sized segments, we should use fixed size chunks
- These are called pages

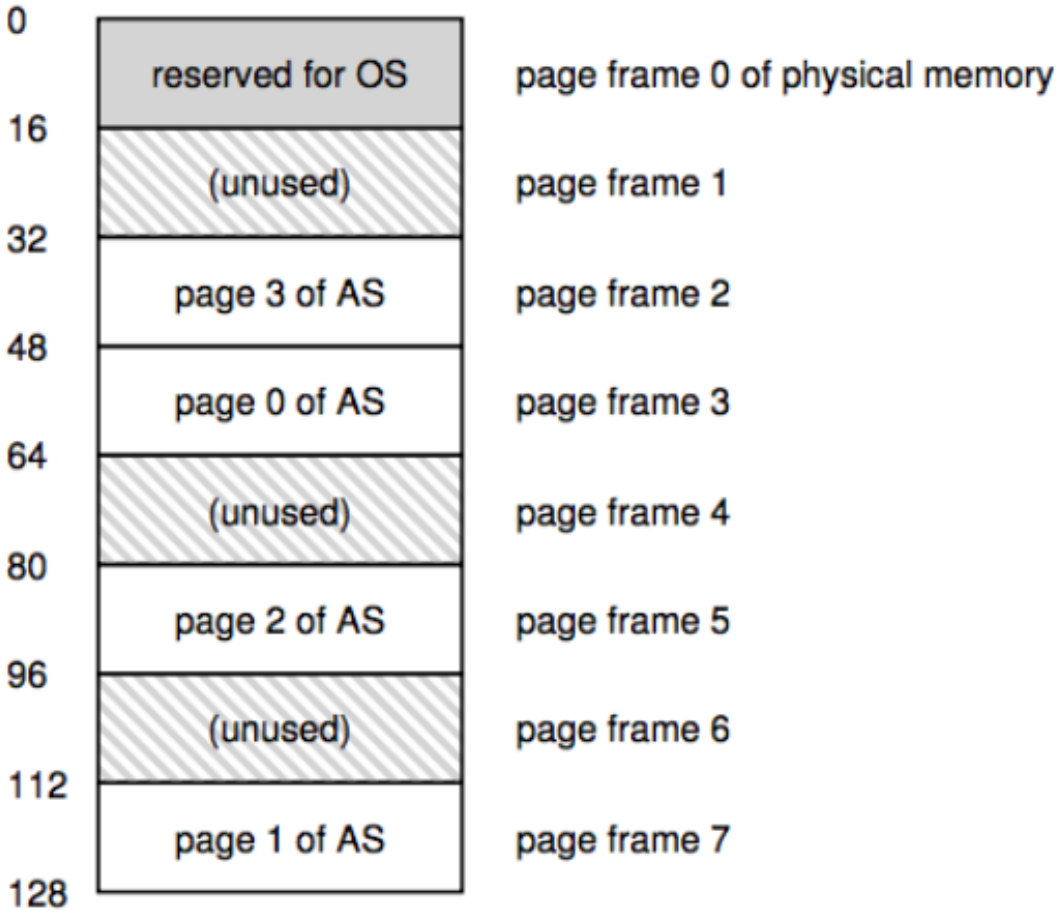
Paging

- Instead of dealing with variable sized segments, we should use fixed size chunks
- These are called pages
- We divide both virtual memory and physical memory into pages of equal size (typically 4kB)

Paging

- Instead of dealing with variable sized segments, we should use fixed size chunks
- These are called pages
- We divide both virtual memory and physical memory into pages of equal size (typically 4kB)
- Physical memory pages are called page frames

Example


0
16
32
48
64
80
96
112
128
reserved for OS
(unused)
page 3 of AS
page 0 of AS
(unused)
page 2 of AS
(unused)
page 1 of AS
page frame 0 of physical memory
page frame 1
page frame 2
page frame 3
page frame 4
page frame 5
page frame 6
page frame 7

Free Space Management

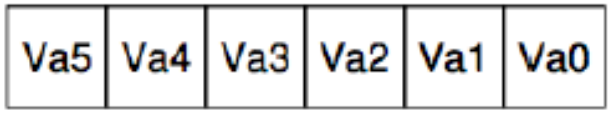
- The OS can keep a list of free pages
- Allocation of a page is $O(1)$

Mapping Virtual to Physical

- We need to keep a map of virtual pages to physical pages
- This is called a page table and there is one per process

Mapping Virtual to Physical

- We need to keep a map of virtual pages to physical pages
- This is called a page table and there is one per process
- Let's start with a simple example with 16 byte pages and 6 bit addresses



Question

If we have 6 bit virtual addresses and 16 byte pages, how many bits of the virtual address tells us which virtual page an address is?

(A) 0

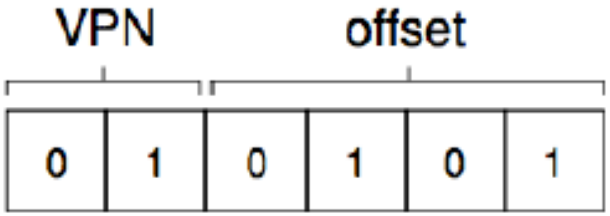
(B) 1

(C) 2

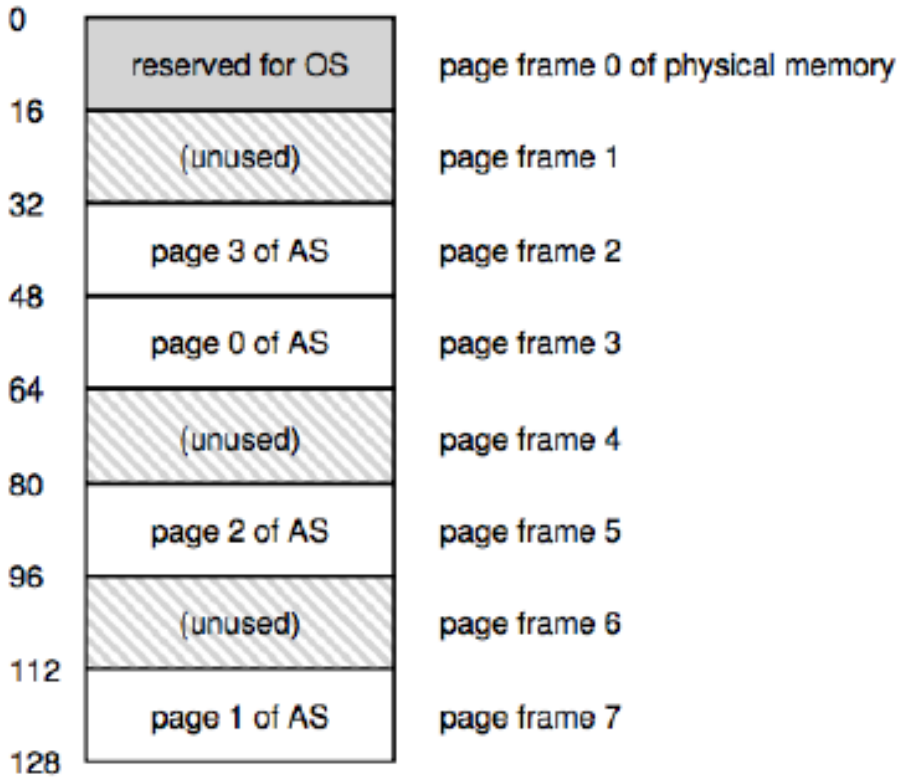
(D) 3

(E) 4

Example

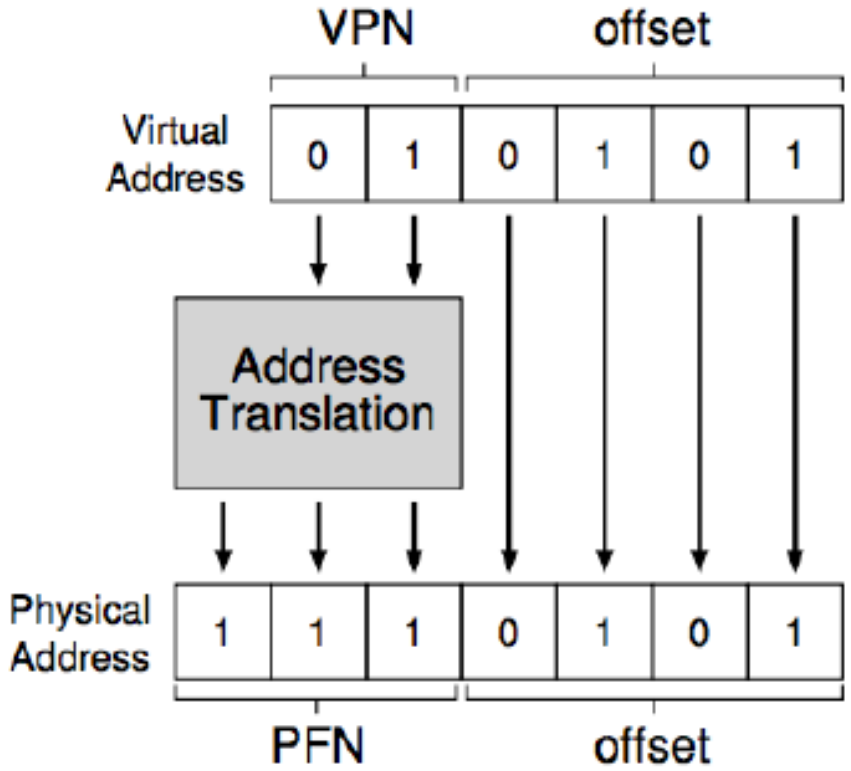


- movl 21, %eax
- 21 = 010101
- So the 5th byte of the virtual page 1
- Which is the 5th byte of physical page 7



Summary of Address Translation

- The physical page is looked up in the page table at index virtual page number
- Then at the offset within that physical page



Question

If we have a 32 bit machine with 4kB pages and each page table entry is 4 bytes, how big is each process page table?

(A) 32 bits

(B) 2^{20} bytes

(C) 2^{22} bytes

(D) 2^{32} bytes

Question

If we have a 32 bit machine with 4kB pages and each page table entry is 4 bytes, how big is each process page table?

(A) 32 bits

(B) 2^{20} bytes

(C) 2^{22} bytes

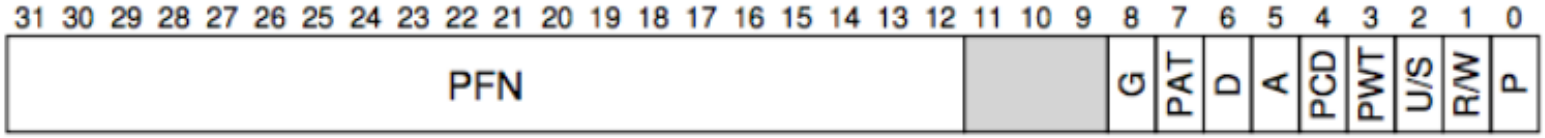
(D) 2^{32} bytes

So 2^{22} bytes and 100 processes is 400 MB

Where do we store page tables?

- In physical memory (as opposed to in a CPU register)
- A simple, linear Page Table is just an array
- Each entry in the array is a Page Table Entry (PTE)

What are we keeping in the PTE?



- Page Frame Number (PFN), the most important part!
- Protection bits (r/w), Present bit (swapped out), Referenced bit (recently accessed)
  - More on these later

How long does it take?

- On each load or store from/to memory, we have to fetch a PTE from memory
- So each memory access is now two memory access?
- So page tables are **big and slow** . . .

Enter: the TLB!

- TLB is a cache of the page table
- Each TLB entry should cache all the information in a PTE
- It also needs to store the VPN as a tag
  - To be used when the hardware searches TLB for a particular VPN
- A typical TLB entry

VPN | PFN | other bits

Reading an Array (no TLB)

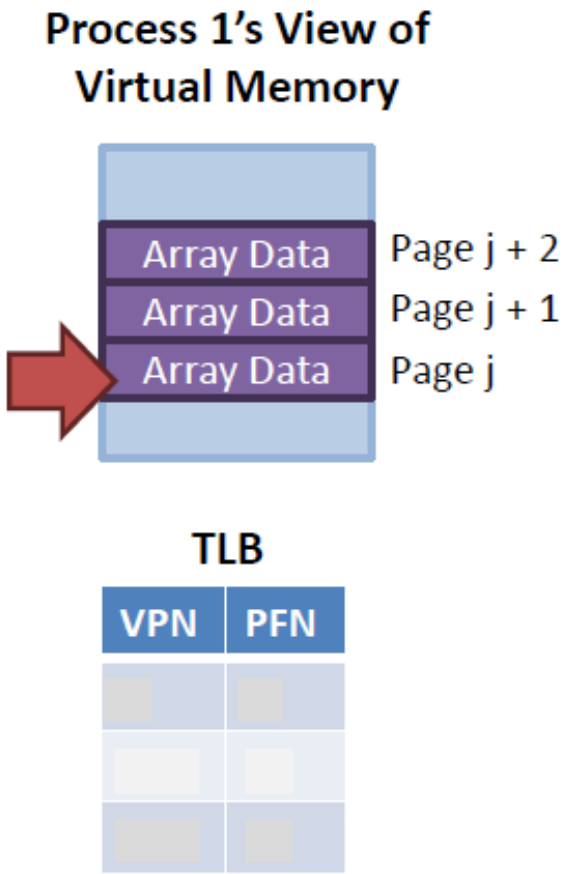
- Suppose we have a 10KB array of integers
  - Assume 4KB pages
- With no TLB, how many memory accesses are required to read the whole array?

Reading an Array (no TLB)

- Suppose we have a 10KB array of integers
  - Assume 4KB pages
- With no TLB, how many memory accesses are required to read the whole array?
  - 10KB / 4 = 2560 integers in the array
  - Each requires one page table lookup, one memory read
  - 5120 reads, plus more for the instructions themselves

Reading an Array (with TLB)

- Same example, now with TLB
  - 10KB integer array
  - 4KB pages
  - Assume the TLB starts off cold (i.e. empty)
- How many memory accesses to read the array?
  - 2560 to read the integers
  - 3 page table lookups
  - 2563 total reads



Locality

- TLB, like any cache, is effective because of locality

Locality

- TLB, like any cache, is effective because of locality
- Spatial locality: if you access memory address x, it is likely you will access $x + 1$ soon
  - Most of the time, x and $x + 1$ are in the same page

Locality

- TLB, like any cache, is effective because of locality
- Spatial locality: if you access memory address x, it is likely you will access $x + 1$ soon
  - Most of the time, x and $x + 1$ are in the same page
- Temporal locality: if you access memory address x, it is likely you will access x again soon
  - The page containing x will still be in the TLB, hopefully