

Introduction to Paging

Cons of segmentation

- * Complex memory management → need to find chunk of a particular size
→ may need to rearrange memory from time-to-time to make room for new segment or growing segment.
- * External fragmentation → wasted space b/w chunks.

In contrast to the variable sized pieces, paging splits up address space into fixed-size units, each of which is called a page.

- with paging, the physical memory → an array of fixed-sized slots, called page frames (of the same size as the virtual pages)

How to map a virtual page to a physical frame?

↳ a page table per process is used for this translation.

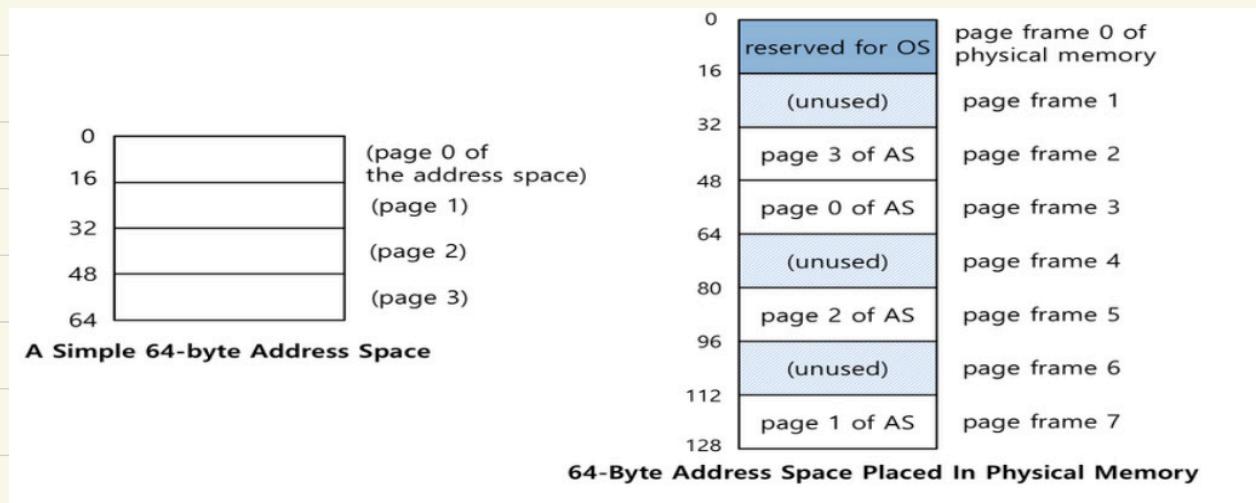
Is used to find, in which page frame does a virtual page resides in the physical memory.

Advantages of Paging

- * flexibility: supporting the abstraction of the address space effectively.
 - don't need to bother how differently heap and stack grow!
- * Simplicity: ease of free-space management
 - the page in the address space and the page frame, both are of the same size.
 - easy to allocate and keep a free list.

Example: A simple paging

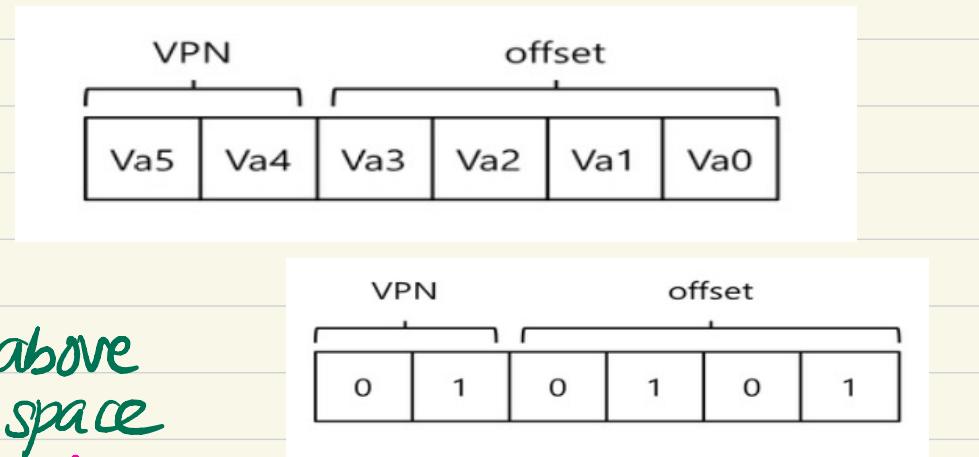
- * 128-byte physical memory with 16 byte page frames
- * 64-byte address space with 16 byte pages



Address translation

* Two components in the virtual address.

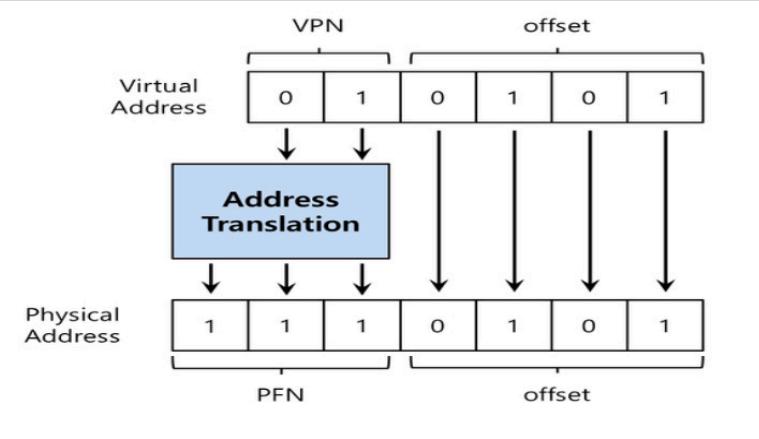
- VPN : virtual page number
- offset : offset within the page



e.g.: Consider the virtual address 21 in the above 64-byte address space

In the above figure it is in the 5th byte (010) of virtual page 1 (01) offset

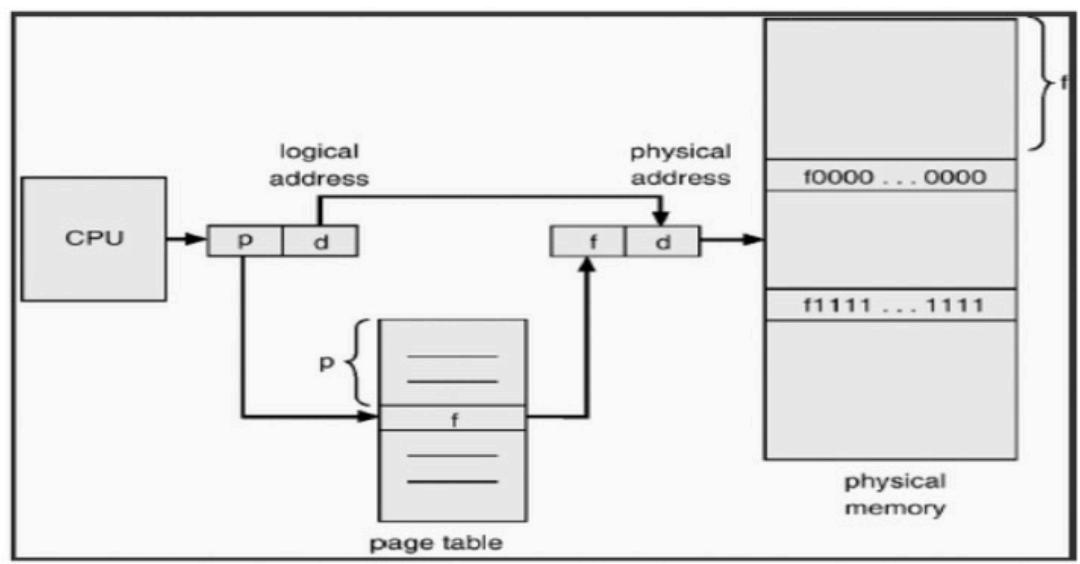
Page table \Rightarrow VPN 1 is mapped to physical frame number (PFN) 7 (111)



Note that the offset remains the same (as the virtual pages and physical pages have the same size)

\Rightarrow final physical address = 1110101

=====



let 2^m = size of virtual
(logical) address space

2^n = size of a page

virtual pages = 2^{m-n}

$\Rightarrow p = m-n$ higher order bits of VA.

$d = n$ lower order bits of VA.

where are page tables stored?

Page tables can be awfully large!

- eg:- consider a 32-bit address space with 4KB pages;
20 bits for VPN

\Rightarrow Entries in the page table = 2^{20} .

\Rightarrow page table size is (roughly) = $2^{20} \times 4$ bytes per entry

$$= 4\text{MB!}$$

Since we need a page table for each process,

100 processes running \Rightarrow 400 MB of memory for translations!

The page tables are stored in "memory" somewhere.

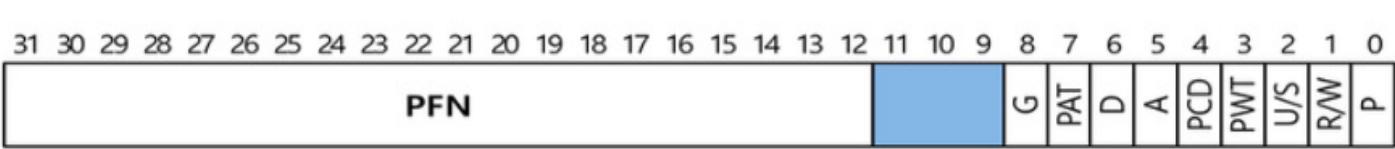
will see the details later!

what's actually in the page table?

- * The page table is just a data structure used to map the virtual address to physical address.
 - ↳ simplest form: a linear page table, an array.
- * The OS indexes the array by VPN, and looks up the page table entry.

Common flags of page table Entry (PTE)

- * Valid bit: indicating whether the translation is valid.
(for eg:- the unused pages in the add-space are marked invalid)
- * Protection bit: indicating whether the page could be read from, written to, or executed etc..
- * Present bit: indicating whether the page is in the physical memory or on the disk (swapped out)
will see later!
- * Dirty bit: indicating whether the page has been modified since it was brought into memory.
- * Reference bit (accessed bit): indicating that a page has been accessed.



an x86 page table entry

- * P: present
- * R/W: read/write
- * U/S: supervisor
- * A: accessed
- * PFN: the page frame number

Overheads in Paging

* To fetch the data from the physical address → first system need to fetch the page table entry.

How does the hardware knows where exactly is the page table of the running process?

→ using a single page table base register that contains the physical address of the starting location of the page table.

location of the desired PTE: Extra memory reference!

$$\text{VPN} = (\text{Virtual Address} \& \text{VPN-MASK}) \gg \text{SHIFT}$$

$$\text{PTE Addr} = \text{Page Table Base Register} + (\text{VPN} * \text{size of (PTE)})$$

in the previous example of 64-byte address space, $\text{VPN_MASK} = 110000$,
 $\text{SHIFT} = 4$

After fetching PTE from PTEAddr,

$$\text{Offset} = \text{VirtualAddress} \& \text{OFFSET_MASK}$$

$$\text{PhysAddr} = (\text{PTE} \cdot \text{PFN} \ll \text{SHIFT}) \mid \text{Offset}$$

Summary of accessing memory with paging

```
1 // Extract the VPN from the virtual address
2 VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4 // Form the address of the page-table entry (PTE)
5 PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7 // Fetch the PTE
8 PTE = AccessMemory(PTEAddr)
9
10 // Check if process can access the page
11 if (PTE.Valid == False)
12     RaiseException(SEGMENTATION_FAULT)
13 else if (CanAccess(PTE.ProtectBits) == False)
14     RaiseException(PROTECTION_FAULT)
15 else
16     // Access is OK: form physical address and fetch it
17     offset = VirtualAddress & OFFSET_MASK
18     PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19     Register = AccessMemory(PhysAddr)
```

Extra memory references are costly, and will likely slow down the process by a factor of two or more!

An example to demonstrate the overhead by the extra memory reference

Consider the following code snippet.

int array;

....

```
for (i=0 ; i<1000 ; i++)
    array [i] = 0;
```

Disassembling the above, there are 4 instructions.

1. mov \$0x0, (%edi,%eax,4) → stores 0 into the array element
(fetching instruction → requires 2 memory references (PTE + instruction))
writing 0 into the array element → again 2 memory references
(PTE + store)
2. incl %eax → increments loop counter
(2 memory references (PTE + instruction))
3. cmpl \$0x3e8, %eax → compares the counter with 1000
(2 memory references (PTE + instruction))
4. jne 0x1024 → if more elements remain, jump back to start of the loop.
(2 memory references (PTE + instruction))

In total, a single loop iteration in the above simple code
causes 10 memory references!

⇒ without careful design of both hardware and software

Problem 1: page tables will slow down the system!

Question: How to speed up address translation and generally avoid the extra memory reference?

↳ using hardware support

"Translation look aside Buffer (TLB)"

(will see in the next chapter!)

Problem 2: Page tables can be huge in size, as a result memory might ends up filled with page table instead of useful application data.

An easy solution → use bigger pages

(⇒ # pages is less ⇒ # PTE are less ⇒ page table is small)

Major problem with this approach

→ Internal fragmentation

(allocating pages that has been used only very less inside

⇒ memory gets filled up with overly large pages)

∴ To solve the above problem, we need advanced page tables.

(will see in the next to next chapter!)