14. Introduction to Paging

OS takes one of 2 approaches when solving most any space-management problem

1. chop things up into variable-sized pieces (ex. segmentation in VM)
   - fragmentation problem
2. chop up space into fixed-sized pieces (ex. paging in VM)

Paging

- page : splitting up a process's address into fixed-sized units, each of which we call a page.
- page frames : physical memory as an array of fixed-sized slots
- each frames can contain a single virtual-memory page

How to virtualize memory with PAGES? to avoid the problems of segmenation?

a 64-bit address space is amazingly large.
if you think of a 32-bit address space as the size of a tennis court, a 64-bit address space is about the size of Europe

Physical memory

consists of a number of fixed-sized slot

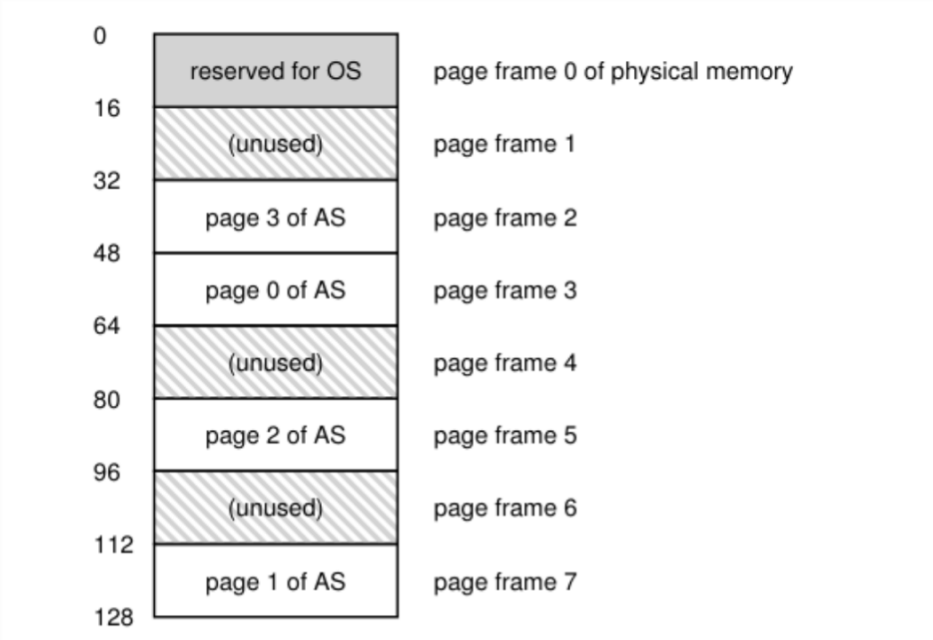

Figure 18.2: **A 64-Byte Address Space In A 128-Byte Physical Memory**

the pages of the virtual address space have been placed at a different locations throughout physical memory.
the diagram also shows the OS using some of physical memory for itself

Advantages of Paging

flexibility :
the system will be able to support the abstraction of an address space effectively, regardless of how a process uses the address space

- ex. don't need to make assumptions about the direction the heap and stack grow and how they are used - it just maps pages on demand (segment can collasp if there are not enough contiguous free space)

simplicity : just grab first free pages since all page sizes are same

Page table

- a per-process data structure
- why : to record where each virtual page of the address space is placed in physical memory
- major role : store address translations for each of the virtual pages of the address space -> letting us know where in physical memory each page resides

If another process were to run, the OS would have to manage a different page table for it.
(its virtual pages obviously map to different physical pages)

Virtual address = virtual page number (VPN) + offset

physical frame number (PFN) == physical page number (PPN)

we can translate virtual address by replacing the VPN with PFN
and then issue the load to physical memory

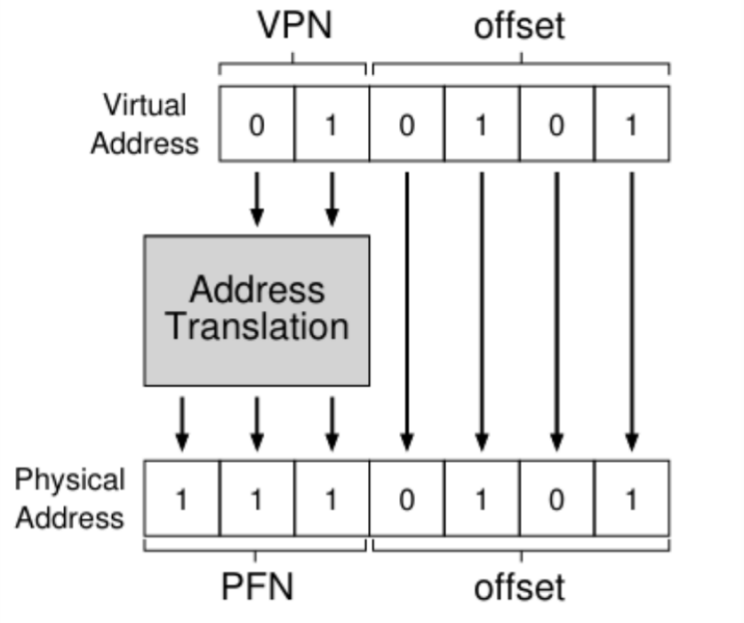

Figure 18.3: **The Address Translation Process**

page tables can get terribly large

much bigger than the small segment table or base/bounds pair

Ex. a typical 32-bit address space, with 4KB pages.

- virtual address : 20 bits for VPN / 2^10 *4 => 12 bits for offset
- 20 bits for VPN => possible translations for each process : 2^20
- if we assume we need 4 bytes per page table engry (PTE) to hold the physical translation + any other useful stuff => we get an immense 4 * 2^20 = 4MB memory needed for each page table
- imagine there are 100 processes running => the OS needs 100 * 4MB = 400MB of memory just for all those address translations!

Think about how big such a page table would be for a 64-bit address space!

Because page tables are so big, we don't keep any special on-chip hardware in the MMU to store the page table of the currently-running process.
=> we store the page table for each process in memory somewhere

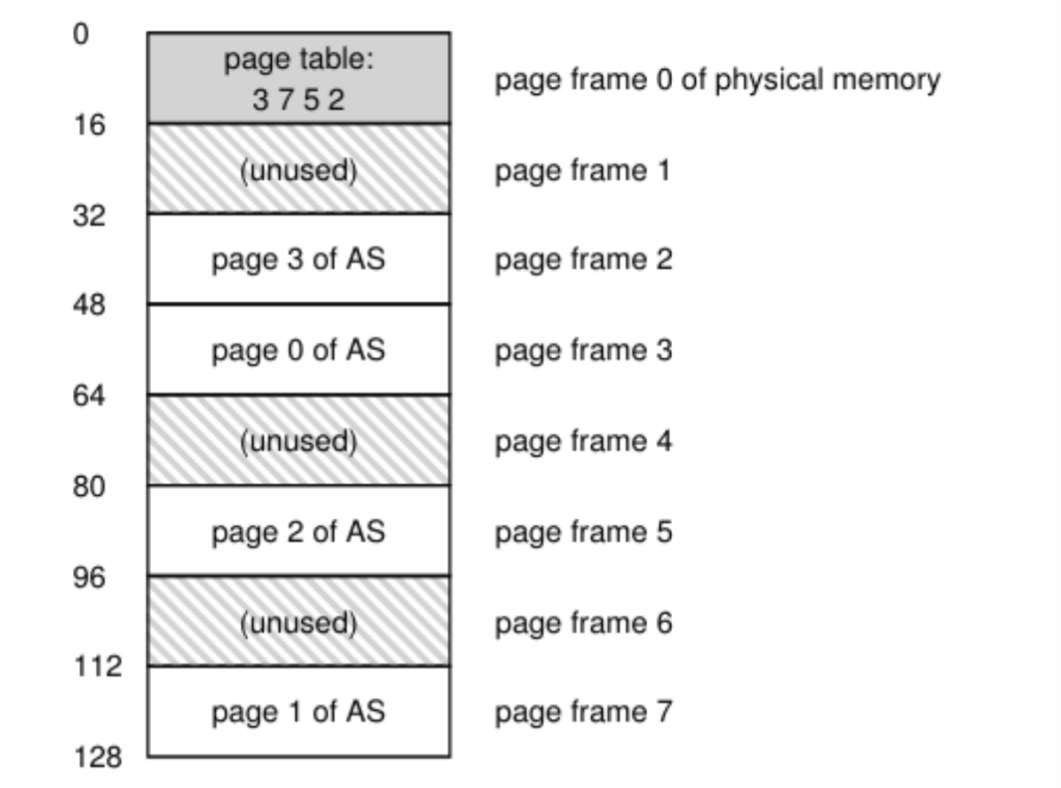

Figure 18.4: **Example: Page Table in Kernel Physical Memory**

memory filled with page tables instead of useful application data

Data structure for Page Table

page table is just a data structure that is used to map virtual address (or really virtual page numbers) to physical addresses (physical frame numbers).
=> any data structure could work, the simplest : a linear page table = array

the OS indexes the array by VPN , and looks up PTE at that index in order to find the desired PFN

contents of each PTE

valid bit

: common to indicate whether the particular translation is valid
ex) unused space in-between stack and heap

if the process tries to access such memory, it will generate a trap to the OS which will likely terminate the process.
=> valid bit is crucial for supporting a sparse address space.

by simply marking all the unused pages in the address space invalid, we remove the need to allocate physical frames for those pages => save a great deal of memory

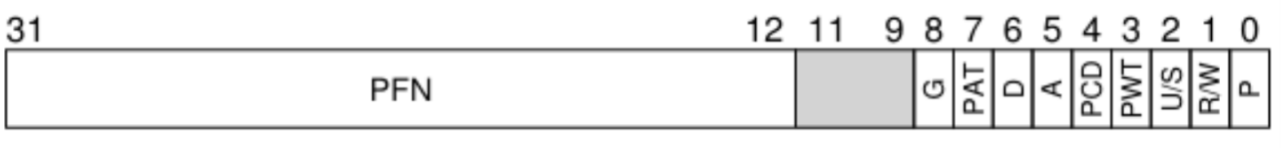

Figure 18.5: **An x86 Page Table Entry (PTE)**

Protection bits

whether the page could be read from, written to, or executed from

present bit

whether this page is in physical memory or on disk (has been swapped out)

dirty bit
indicating whether the page has been modified since it was brought into memory

reference bit = accessed bit
used to track whether a page has been accessed.
useful in determining which pages are popular => should be kept in memory
critical during page replacement!

Paging is too slow

not only it might be too big, but also it can slow!

```
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 OK: form physical address and fetch it
17     offset   = VirtualAddress & OFFSET_MASK
18     PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19     Register = AccessMemory(PhysAddr)
```

Figure 18.6: **Accessing Memory With Paging**

Once physical address is known, the hardware can fetch the PTE from memory, extract the PFN, and concatenate it with the offset from the virtual address to form the desired physical address.

For every memory reference (whether an instruction fetch or an explicit load or store), paging requires us to perform **one extra memory reference in order to first fetch the translation from the page table**.

That is a lot of work!

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

=> without careful design of both hardware and software page tables will cause the system to run too slowly, as well as take up too much memory.

While seemingly a great solution for our memory virtualization needs, these 2 crucial problems (slow, big) must first be overcome.

Memory trace

c:

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

x86 :

```
1024 movl $0x0,(%edi,%eax,4)
1028 incl %eax
1032 cmpl $0x03e8,%eax
1036 jne  1024
```

- $0x0 = 0 / $0x03e8 = 1000 / %edi : base addr / %eax : idx

each instruction fetch basically will generate 2 memory references:

1. to the page table to find the physical frame that the instruction resides within
2. instruction itself to fetch it to the CPU for processing

Especially, mov instruction adds 2 more memory reference

1. page table access : to translate the array virtual address to the correct physical one
2. the array access itself

There are 10 memory accesses per loop

instruction fetch : 4
update of memory : 1
page table accesses : 5

- to translate those 4 instruction fetches
- explicit update : 1