Lecture 8: Paging

Operating Systems

Content taken from: https://www.cse.iitb.ac.in/~mythili/os/

https://pages.cs.wisc.edu/~remzi/OSTEP/

Last Two Classes

- Address Translation from virtual address to physical address
- Dynamic Relocation with Base and Bounds registers
- Segmentation

Segment	Base	Size (max 4K)	Grows Positive?	Protection
Code_{00}	32K	2K	1	Read-Execute
Heap_{01}	34K	3K	1	Read-Write
Stack_{11}	28K	2K	0	Read-Write

Figure 16.5: **Segment Register Values (with Protection)**

Issue with Segmentation: Fragmentation

- Different-sized segments are allocated memory.
- Free memory (hole) is too small and scattered
  - Segment to be allocated may be larger than the free memory hole but smaller than the total amount of free memory available

Paging

- Divide the memory into fixed-sized units instead of variable-sized logical segments.
- Each fixed-sized unit is called a page.
- Physical memory can be seen as an array of fixed sized slots called page frames.
- Each page frame can contain a single virtual-memory page.
- **Comparing it to segmentation, each logical segment can be divided to one or more virtual pages.**

Example

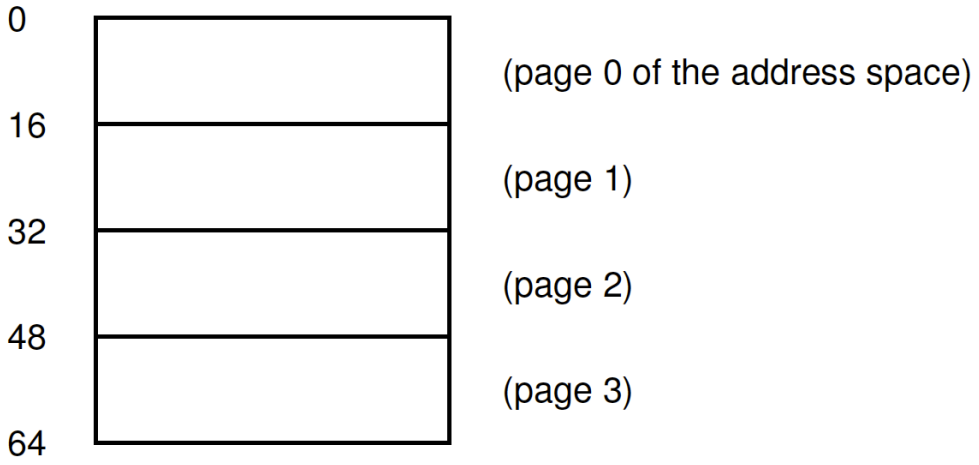


Figure 18.1: **A Simple 64-byte Address Space**

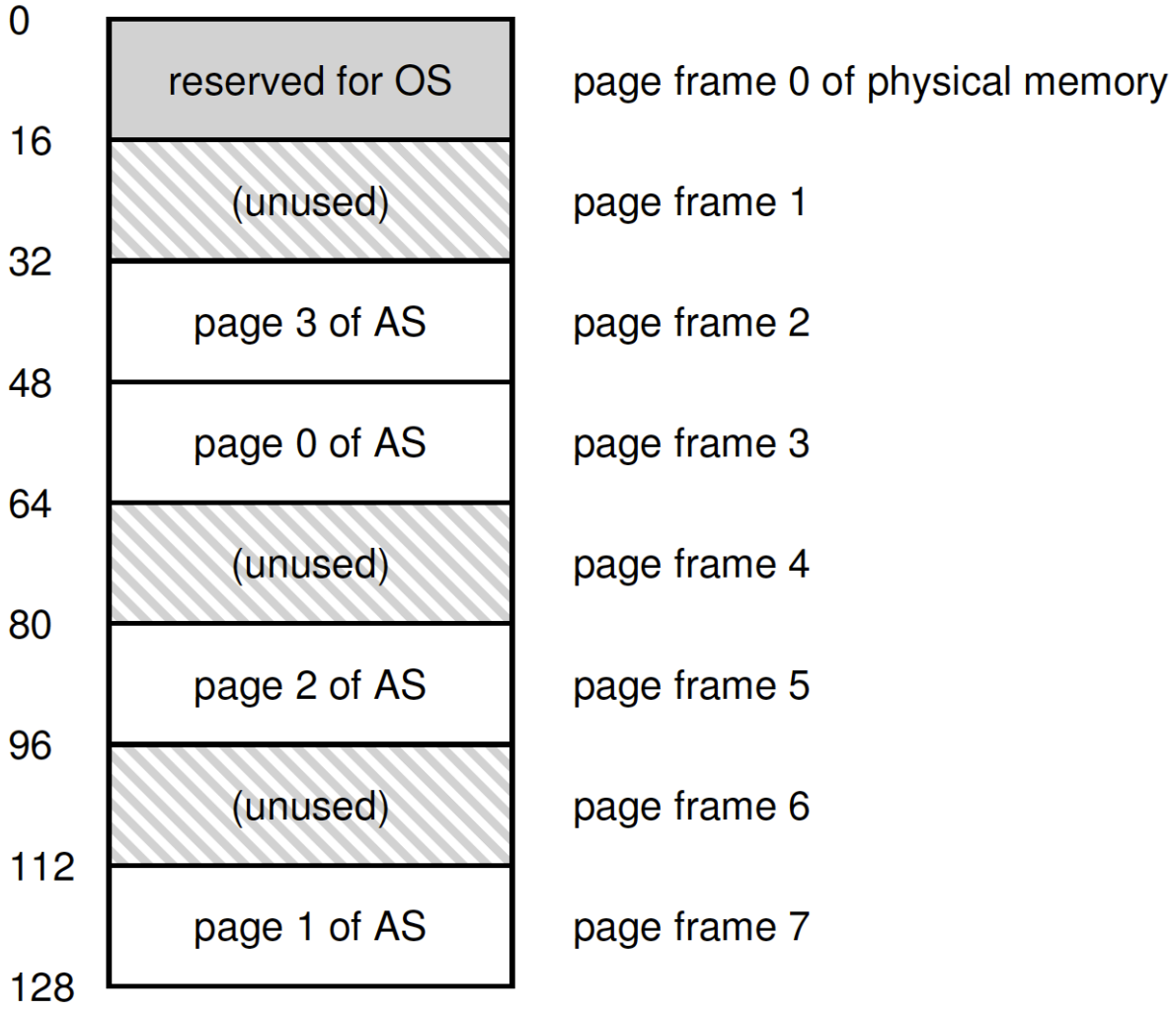
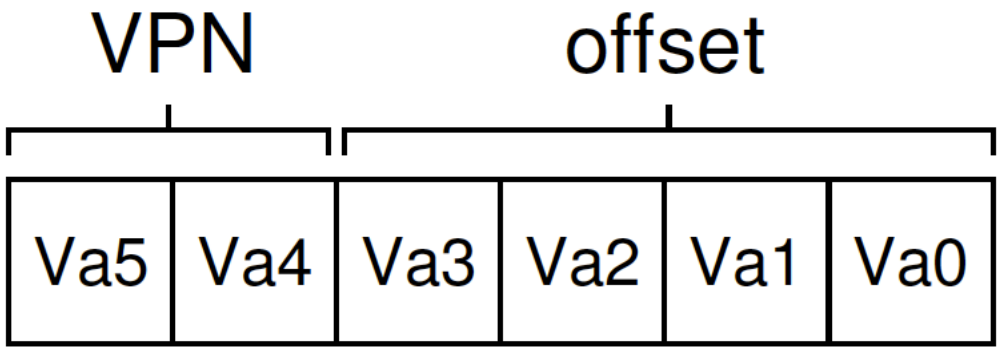


Figure 18.2: **A 64-Byte Address Space In A 128-Byte Physical Memory**

Mapping from Virtual Pages to Physical Page Frames

- OS keeps a per-process data structure known as a **page table**.
- What would be the contents of the page table in the previous example?
    - (Virtual Page 0 → Physical Frame 3), (VP 1 → PF 7), (VP 2 → PF 5), and (VP 3 → PF 2)

Address Translation



- 64 byte address space needs 6 bits
- 4 pages in 64 byte address space. So the size of each page is 16 bytes (4 bits)

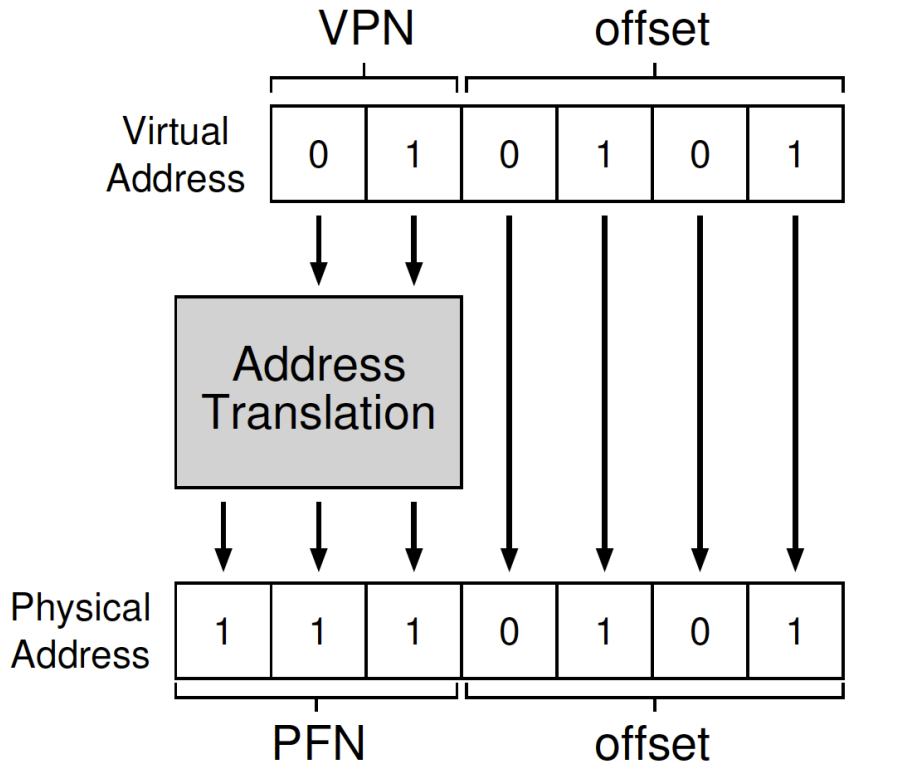


Figure 18.3: **The Address Translation Process**

Where is the Page table stored?

- Page table live in the memory managed by the OS.
- **Page-Table Base Register (PTBR)** contains the physical address of the starting location of the Page Table.

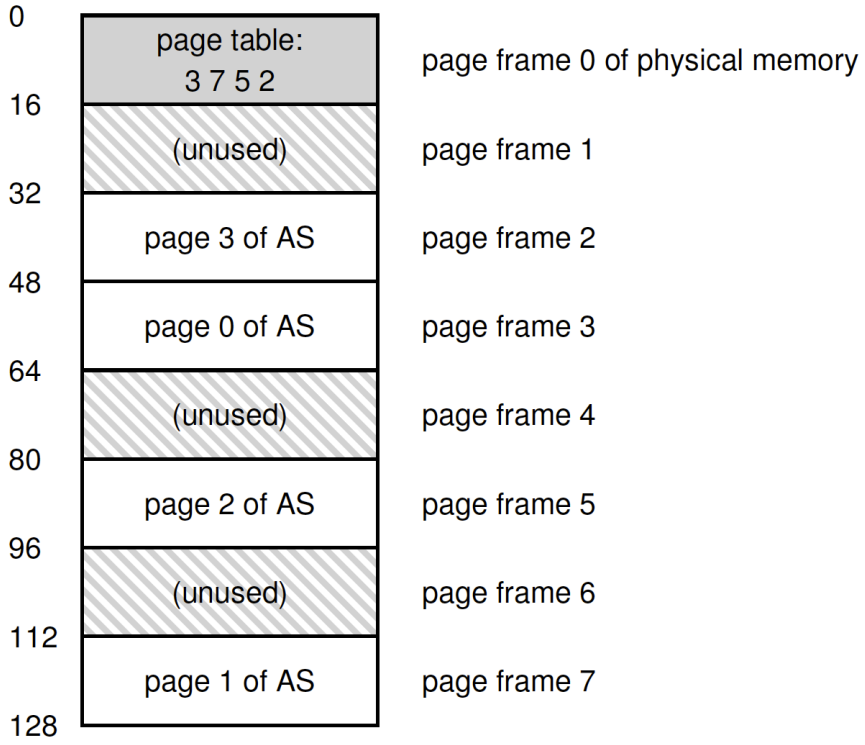


Figure 18.4: **Example: Page Table in Kernel Physical Memory**

What is actually in the Page table?

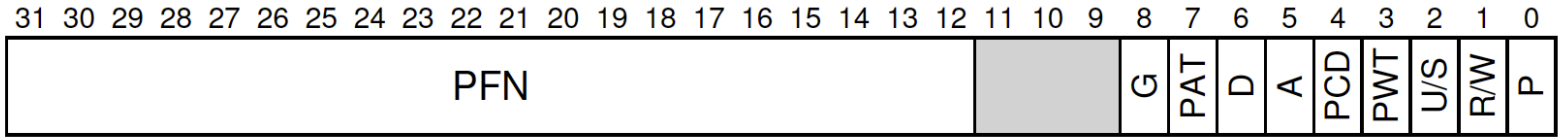


Figure 18.5: **An x86 Page Table Entry (PTE)**

- The present bit (P) indicates whether the page is in physical memory or on disk
- The read/write bit (R/W) indicates whether or not writes are allowed on this page.
- The user/supervisor bit (U/S) determines if the user-mode processes can access this page
- PWT/PCD/PAT and G indicate how hardware caching works for these pages
- The accessed bit (A) indicates whether or not a page has been accessed.
- The dirty bit (D) indicates whether the page has been modified after it was brought into memory

Paging can be too slow

- Let us take an example
- Hardware must first fetch the PTE from memory
- Hardware can then fetch the desired data from memory
- How many memory references did we do?

```
movl 21, %eax
```

```
VPN     = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))
```

```
offset   = VirtualAddress & OFFSET_MASK
PhysAddr = (PFN << SHIFT) | offset
```

- VPN_MASK is 110000
- SHIFT is 4
- OFFSET_MASK is 001111

```
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)
```

Figure 18.6: **Accessing Memory With Paging**

Faster Translation Using TLBs

- Add some hardware support to MMU
  - Translation-lookaside buffer (TLB)
- TLB is simply a hardware cache of popular virtual-to-physical address translations
- Upon each virtual memory reference, the hardware first checks the TLB
- If the desired translation is held in TLB, the translation is performed (quickly) without having to consult the page table (which has all translations)

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)   // TLB Hit
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset   = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register = AccessMemory(PhysAddr)
8      else
9          RaiseException(PROTECTION_FAULT)
10 else                    // TLB Miss
11     PTEAddr = PTBR + (VPN * sizeof(PTE))
12     PTE = AccessMemory(PTEAddr)
13     if (PTE.Valid == False)
14         RaiseException(SEGMENTATION_FAULT)
15     else if (CanAccess(PTE.ProtectBits) == False)
16         RaiseException(PROTECTION_FAULT)
17     else
18         TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19         RetryInstruction()
```

Figure 19.1: **TLB Control Flow Algorithm**