Chapter 18: Paging Introduction

Adam Disney

THE UNIVERSITY OF
TENNESSEE
KNOXVILLE

Crux: How To Virtualize Memory With Pages

THE CRUX:
HOW TO VIRTUALIZE MEMORY WITH PAGES

How can we virtualize memory with pages, so as to avoid the problems of segmentation? What are the basic techniques? How do we make those techniques work well, with minimal space and time overheads?

Overview

- Instead of variable-sized segments, let's have fixed-sized **pages**.
- We divide the virtual address space into fixed-sized **pages.**
- We divide the physical address space into fixed-sized **page frames.**
- This approach is simple for free-space management because of fixed-sized units.
  - No assumptions about how the program will use address space.
  - No external fragmentation

A Simple Example

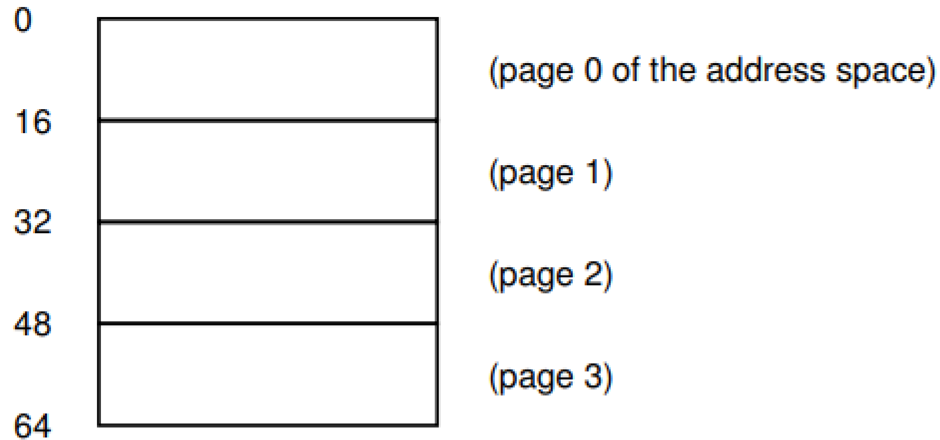


Figure 18.1: **A Simple 64-byte Address Space**

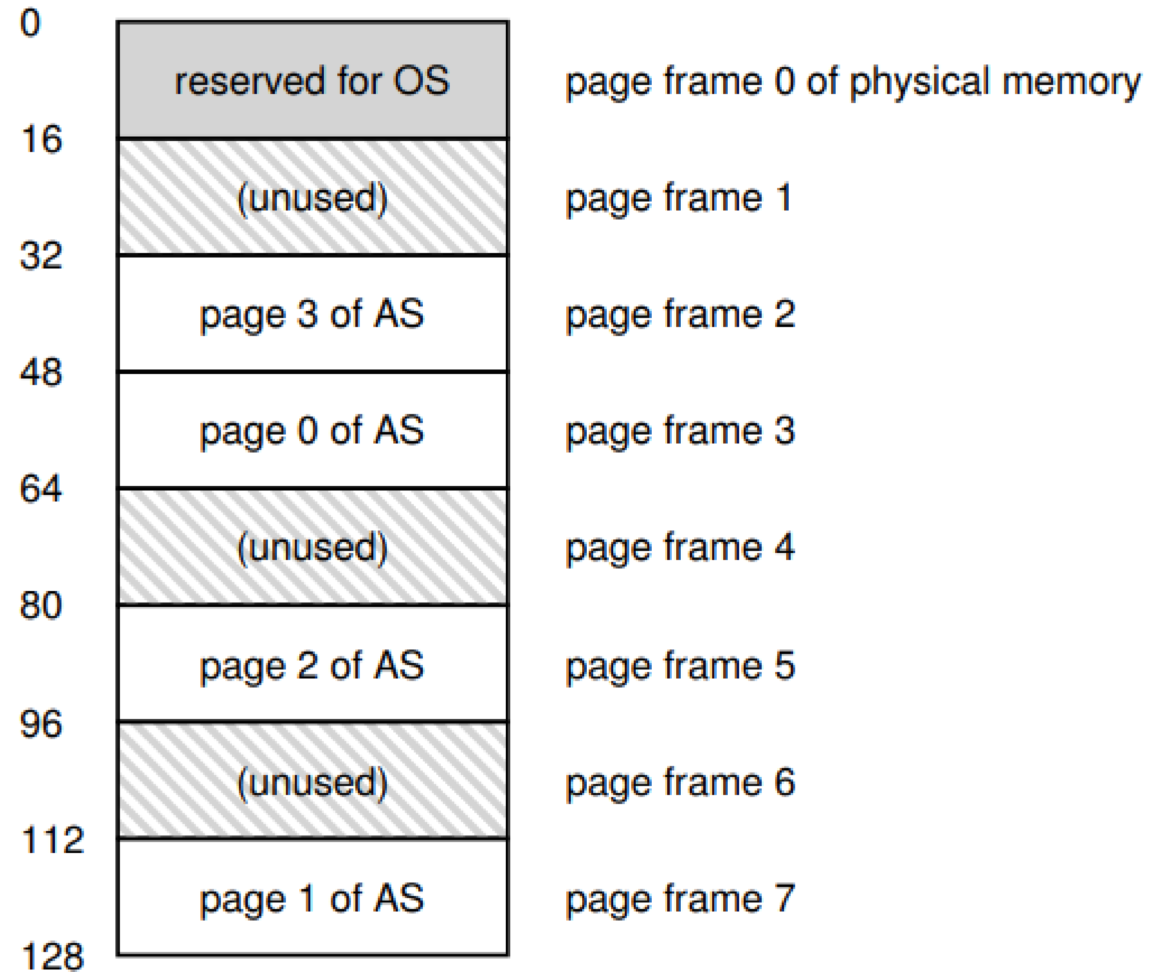
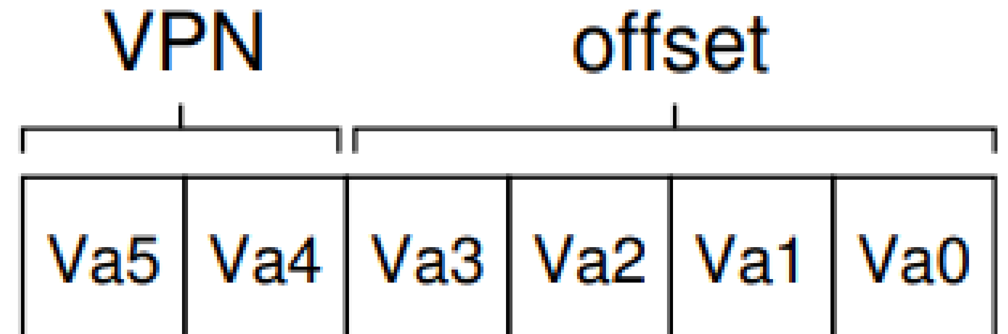


Figure 18.2: **A 64-Byte Address Space In A 128-Byte Physical Memory**

Address Translation

- OS maintains a **page table** per-process.
  - For example, VPage 0 -> PPage 3
- To translate, we split the virtual address into virtual page number (VPN) and offset.
- The VPN translates to a physical frame number (PFN).



Address Translation

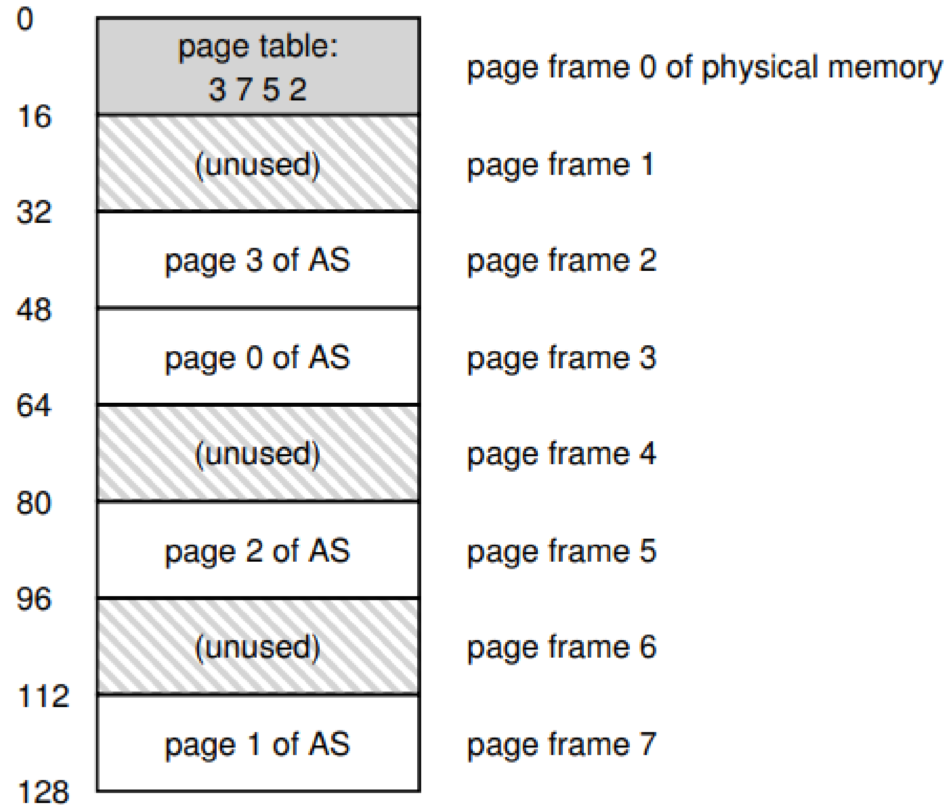


Figure 18.4: **Example: Page Table in Kernel Physical Memory**

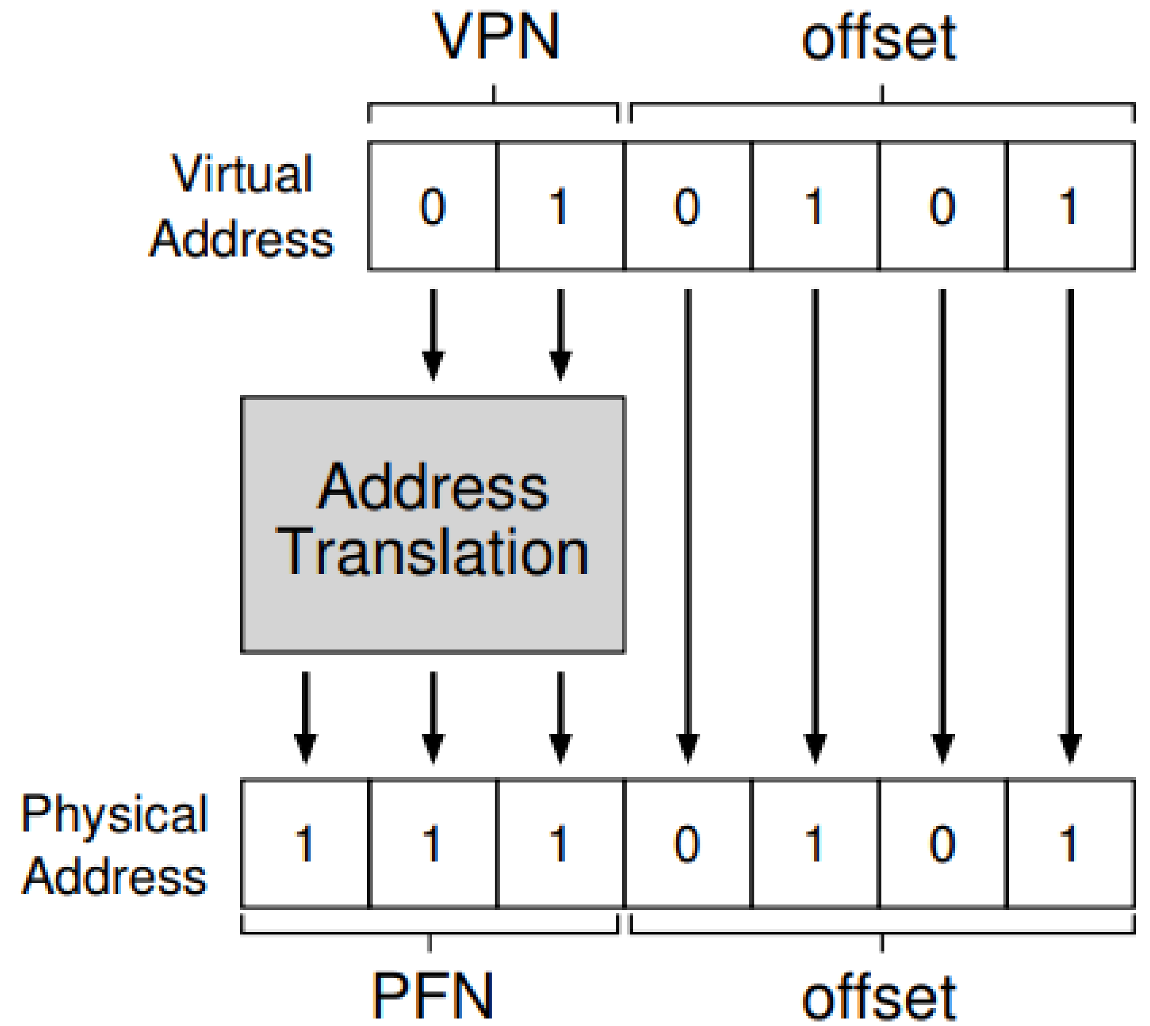


Figure 18.3: **The Address Translation Process**

Where Are Page Tables Stored?

- Page tables can get very large compared to base/bounds pairs.
- Imagine 32-bit address space with 4KB pages.
  - 20-bit VPN + 12-bit offset
  - That's 2^{20} translations per process!
  - If each entry was 4 bytes, that's 4MB per process needed for page tables.
- Thus, we don't store the page table in the MMU but somewhere in physical memory.

What's Actually In The Page Table?

- For now, we will use a simple **linear page table**.
- This is just an array where at index VPN, there is a page table entry (PTE) that contains:
  - The Physical Frame Number (PFN)
  - Valid bit
  - Protection bits (RWX)
  - Present bit (It's in physical memory. More on this later.)
  - Dirty bit (It has been modified.)
  - Reference/Accessed bit (Is it being used?)

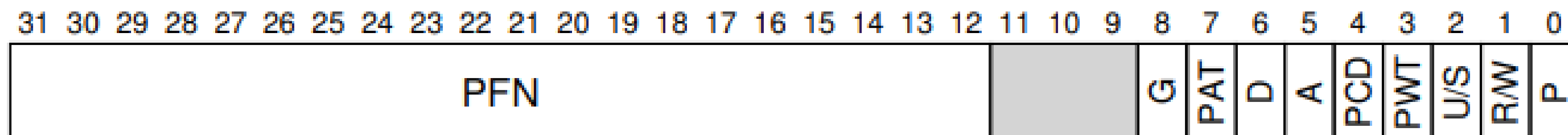


Figure 18.5: **An x86 Page Table Entry (PTE)**

Paging: Also Too Slow

- Suppose this simple instruction is run
  - Movl 21, %eax
- Must translate VA 21 to a PA. Needs the page table for this.
- Suppose the hardware has a single **page-table base register**.
- With that, the hardware can basically do the following:
  - VPN = (VirtualAddress & VPN_MASK) >> SHIFT
  - PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))
  - offset = VirtualAddress & OFFSET_MASK
  - PhysAddr = (PTEAddr->PFN << SHIFT) | offset
- This requires an extra memory access for every memory access!

Paging: Also Too 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 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**

Paging: Also Too Slow

- Thus, we have two major issues we must resolve.
  - Memory usage
  - Extra memory references

Paging: Also Too Slow - Example

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

```
1024 movl $0x0,(%edi,%eax,4)
1028 incl %eax
1032 cmpl $0x03e8,%eax
1036 jne  0x1024
```

Paging: Also Too Slow - Example

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

- Per iteration of the loop:
  - 4 instruction fetches
  - 1 explicit memory update
  - 5 page table accesses
  - Totaling 10 memory accesses per iteration!

```
1024 movl $0x0,(%edi,%eax,4)
1028 incl %eax
1032 cmpl $0x03e8,%eax
1036 jne  0x1024
```