OSTEP Chapter 18

ECE 3600, Fall 2022

Table of Contents

1. Paging

page = fixed-size unit of virtual memory

page frame = fixed-size unit of physical memory

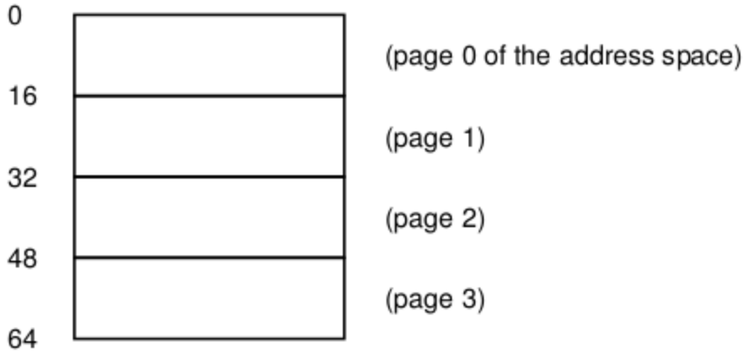


Figure 18.1: **A Simple 64-byte Address Space**

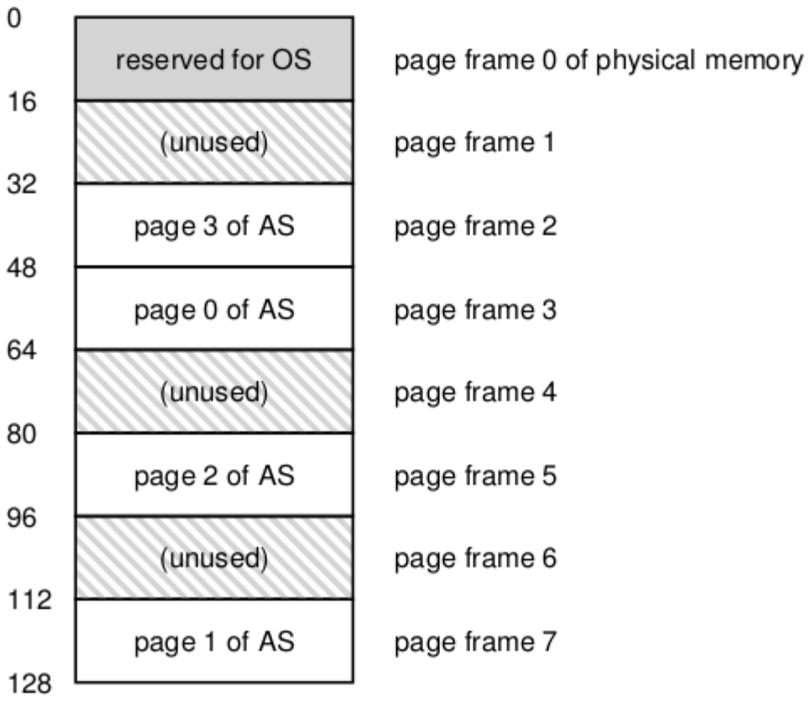
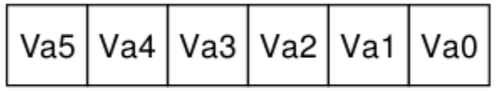


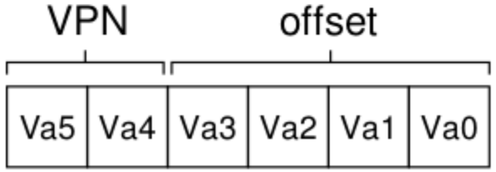
Figure 18.2: **A 64-Byte Address Space In A 128-Byte Physical Memory**

2. Virtual Page Number

To **translate** this virtual address that the process generated, we have to first split it into two components: the **virtual page number (VPN)**, and the **offset** within the page. For this example, because the virtual address space of the process is 64 bytes, we need 6 bits total for our virtual address ($2^6 = 64$). Thus, our virtual address can be conceptualized as follows:



In this diagram, Va5 is the highest-order bit of the virtual address, and Va0 the lowest-order bit. Because we know the page size (16 bytes), we can further divide the virtual address as follows:



3. Virtual Address Translation

PFN = physical frame number

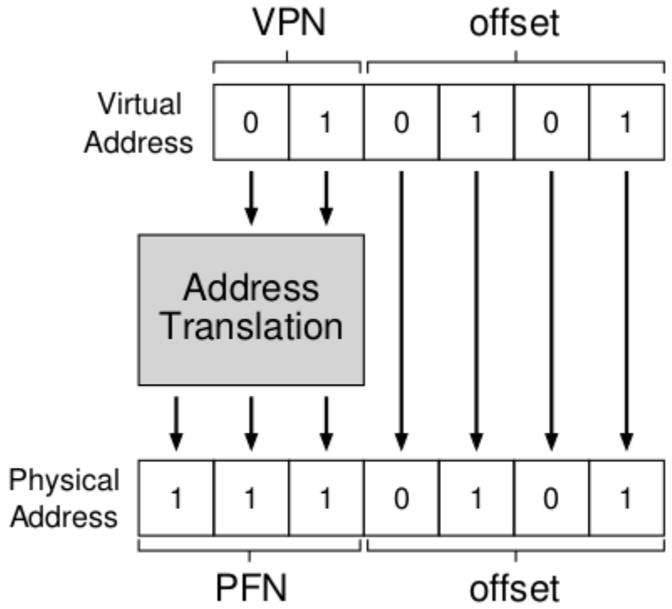


Figure 18.3: **The Address Translation Process**

4. Linear Page Table

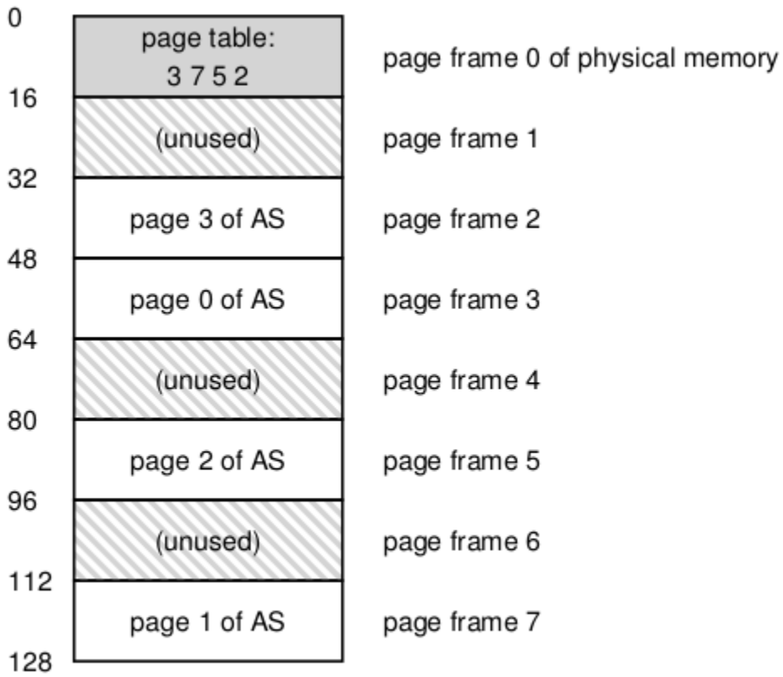


Figure 18.4: **Example: Page Table in Kernel Physical Memory**

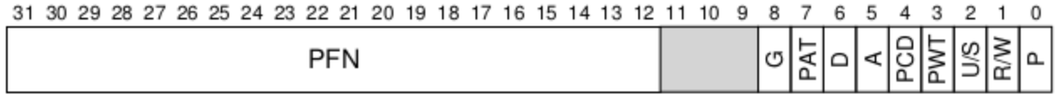


Figure 18.5: **An x86 Page Table Entry (PTE)**

5. Translation Protocol

PTBR = page-table base register

```
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**

specify VPN_MASK, SHIFT, OFFSET_MASK, and PFN_SHIFT : _____

6. Code Example

From the book, Chapter 18 pages 9-10:

```
int array[1000]; // 1000 = 0x3e8
...
for (i = 0; i < 1000; i++)
  array[i] = 0;

gcc -o array array.c -Wall -O
objdump -d array

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

Compare using gcc-8.2.0 on Linux x86_64:

```
int main(void) {
  int array[1000];
  for( int i = 0; i < 1000; ++i) array[i] = 0;
  return array[10];
}

gcc -m32 -S array.c

array.s:
...
        movl    $0, -4(%ebp)
        jmp     .L2
.L3:
        movl    -4(%ebp), %eax
        movl    $0, -4004(%ebp,%eax,4)
        addl    $1, -4(%ebp)
.L2:
        cmpl    $999, -4(%ebp)
        jle     .L3
```

7. Memory Trace

Assume: virtual address space size = 64KB, page size = 1KB
linear (array-based) page table at physical address = 1KB (1024)
Code: VA = 1024 --> VPN = 1 --> PFN = 4
array: VA = 40000 --> VPN = 39, 40, 41, 42 --> PFN = 7, 8, 9, 10

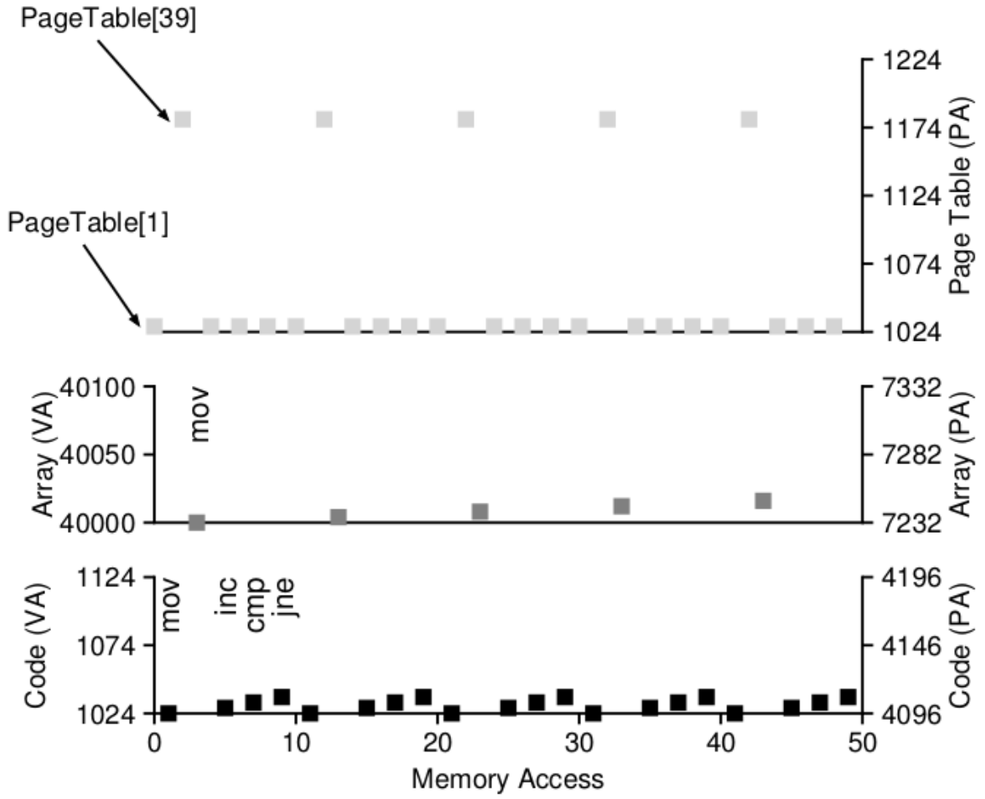


Figure 18.7: **A Virtual (And Physical) Memory Trace**

verify PA 7232 = _____

8. Exercises

Exercises from the book using paging-linear-translate.py:

```
args = -P 1k -a 1m -p 512m -v -n 0
ARG address space size 1m
ARG phys mem size 512m
ARG page size 1k
size =  1024

args = -P 1k -a 2m -p 512m -v -n 0
ARG address space size 2m
ARG phys mem size 512m
ARG page size 1k
size =  2048
```