Chapter 18 Paging: Introduction

Challenging with variable-sized segments

- Leads to external fragmentation
- Allocation becomes more difficult over time
  - Requires allocation strategies

New approach:

- Chop up space into fixed-sized chunks
- In virtual memory, this is called **paging**.

Instead of variable-sized logical segments for code, heap, stack…

- Divide virtual address space into fixed-sized units: **page**.
- Can view the physical memory as *an array of fixed-sized slots*: **page frames**.
- *Each frame can contain a single virtual memory page*.

Q: How can we virtualize memory with pages, and avoid the problems with segmentation?

18.1 A simple Example and Overview

Tiny address space:

- 64 bytes in total
- Four 16-byte pages
- Virtual pages: 0, 1, 2, 3

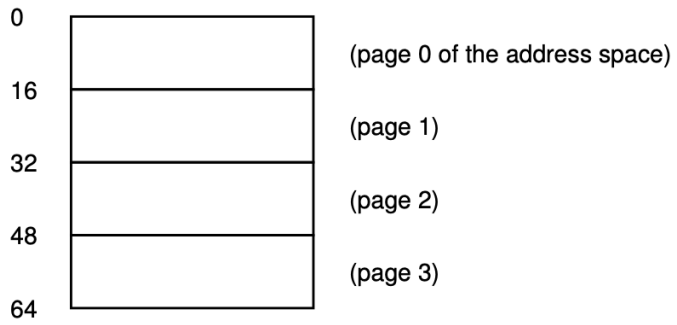


Figure 18.1: **A Simple 64-byte Address Space**

Real address spaces:

- 32 bits = 4 GB
- 64 bits = 16 exabytes
- 48 bits = 280 terabytes

x84-64 / amd64 and ARMv8 only has 48 bits of virtual addresses.

Example Fig. 18.2:

- Placement of virtual pages is flexible
- No assumptions about heap / stack growth
- Simplicity of free-space management
  - OS simply finds four free pages

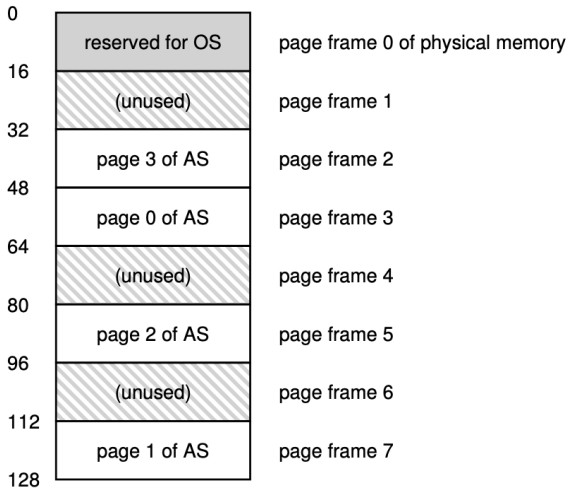


Figure 18.2: **A 64-Byte Address Space In A 128-Byte Physical Memory**

The OS is responsible for mapping

- **pages** from a process's (virtual) address space **to**
- **page frames** in physical memory
- We write:
  - VP *x* -> PF *y*
  - Meaning: Virtual Page *x* maps to Page Frame *y*

In Fig. 18.2 the OS has placed

- VP 0 -> PF 3
- VP 1 -> PF 7
- VP 2 -> PF 5
- VP 3 -> PF 2
- Page frames 1, 4, and 6 are currently free!

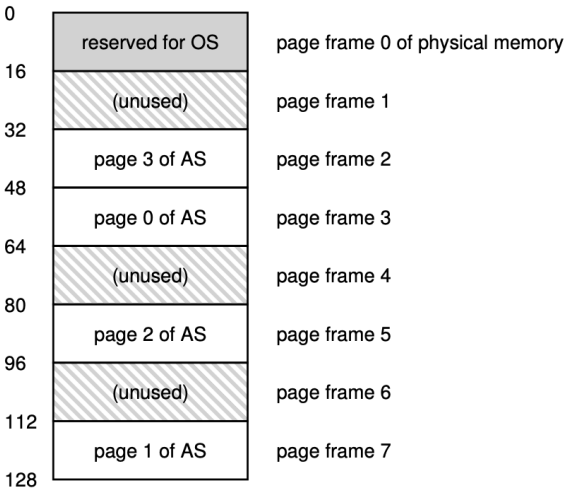


Figure 18.2: **A 64-Byte Address Space In A 128-Byte Physical Memory**

Another process would have a different mapping.

- VP 0 -> PF 1 (currently unused)
- VP 1 -> PF 6 (and so on…)

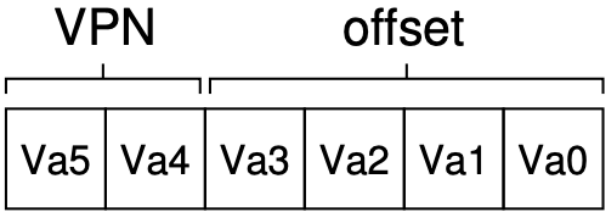
Example. Address-translation with our 64-byte address space

movl <virtual address>, %eax

Load data from <virtual address> into register eax.
(ignore the prior instruction fetch).

Split <virtual address> into two components:

- Virtual page number (VPN)
- Offset



Since our virtual address space is 2^6 = 64 bytes, we need 6 bits for the virtual address.

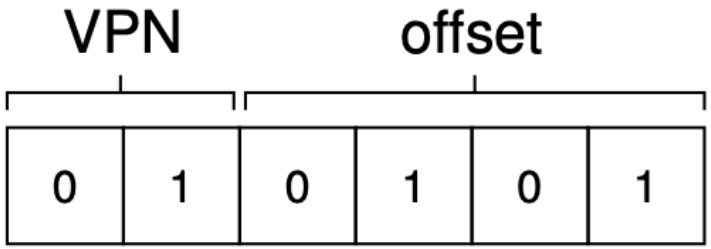
- Page size: 2^4 = 16 bytes
  - Need **four** bits for the offset
- Address space: 64 bytes
- 64/16 = 4 = $2^6/2^4$ = 2^2 pages
  - Need **two** bits for the VPN

Example. Concrete translation

```
movl 21, %eax
```

When process generates *virtual address* 21, OS and HW must

- Translate “21” into VPN + offset
- 21 = 01 0101
  - Virtual page number: 01 = 1
  - Offset: 0101 = 5
  - (fifth byte in the page)
- Translate VPN to Physical Frame Number (PFN)
  - Use a Page Table:
  - VPN 1 —> PFN 7
    - Recall: 0b111 = 7



Final physical address is 111 0101 (117 decimal)

Confirm with Fig 18.2:

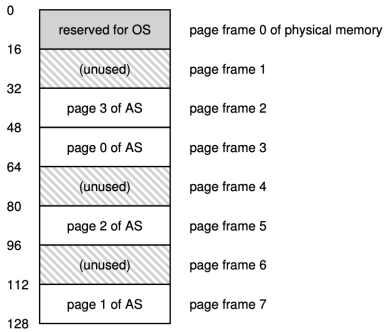


Figure 18.2: **A 64-Byte Address Space In A 128-Byte Physical Memory**

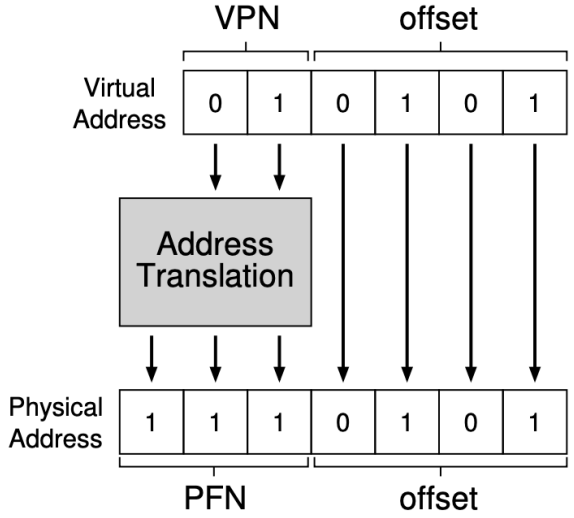


Figure 18.3: **The Address Translation Process**

Q: *Where to store the **virtual page** to **page frame** mapping?*

OS keeps a *per-process* data structure known as a **page table**.

Page table store **address translations** for each virtual page of the address space.

18.2 Where are Page Tables Stored?

Page tables can get big. Much bigger than a small segment table with base/bounds pairs…

Example. 32-bit address space, with 4 KB pages (common because size lines up with disk block sizes)

Virtual address space splits

- 20-bit VPN
- 12-bit offset (4 KB)

Q: *How much space is needed to store page tables?*

20-bit VPN — this implies 2^{20} translations (1048576 translations = 1 million)

- Assume we need 4 bytes per page table entry (PTE)
  - Translation + other useful stuff
- 4 MB for each page table ; one per process
- 100 processes —> 400 MB of memory just for address translations!!

Because page tables are so big: no special on-chip HW.
Instead we store page for each process in the memory itself.

Fig 18.4 shows page table in physical memory.

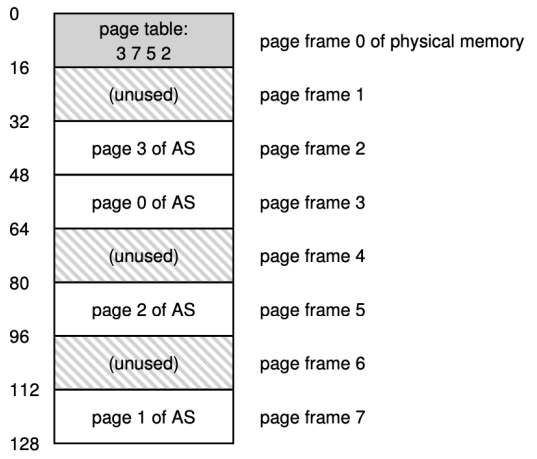


Figure 18.4: **Example: Page Table in Kernel Physical Memory**

18.3 What's Actually in the Page Table?

For now consider a linear page table, or an array that the OS indexes into using the VPN.

VPN -> PTE

Page Table Entry bits:

- **Valid bit**: indicate whether or not the translation is valid
  - Is the VPN valid?
  - Useful to mark all unused space in-between heap and stack as invalid
  - If process tries to access an invalid VPN —> kernel trap —> termination of proc
- **Protection bits**: indicate whether the page can be
  - Read from
  - Write to
  - Execute from
  - Violation: kernel trap
- **Present bit**: indicate whether page is in physical memory or on disk
  - (more on this later…) allows to run proc with address spaces larger than physical memory…
- **Dirty bit**: indicate whether the page has been modified since it was loaded (into memory)
- **Reference bit**: track if a page has been accessed
  - Useful for determining which pages are popular (and should be kept in memory)
  - (important for page replacement)

18.4 Paging: Too Slow?

Consider our example:

```
movl 21, %eax
```

To fetch the data at virtual address 21:

- System must first fetch page table entry from the process's page table (*in memory*)
- Perform the translation: translate virtual address 21 to physical address 117
- Then load data from physical address 117 (*in memory*)

HW must know where the page table is for the currently running proc:

- Assume a single **page-table base register** with the physical starting location of the page table
- To find location of PTE, HW must:

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

Example. Consider virtual address 21 = 01 0101

```
VPN_MASK = 11 0000      (extract the VPN bits from the virtual address)
SHIFT = 4               (number of bits in the offset)
sizeof(PTE) = 4         (size of the page table entry)

        01 0101     (Virtual Address)
AND     11 0000     (VPN_MASK)
        01 0000
>> 4    00 0001     (SHIFT >> 4)  = Virtual Page 1
```

We then use virtual page 1 as an index into the array of the PTEs (page table).

```
PTEAddr = PageTableBaseRegister + (1 * 4)
```

Knowing the physical address, the HW can fetch the PTE from memory

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

OFFSET_MASK = 001111 (extract the offset from the virtual address)

Compute the PhysAddr by shifting the PFN bits to the location of the VPN and bitwise OR with the offset.

Example:

	01 0101	(virtual address)
AND	00 1111	(OFFSET_MASK)
	00 0101	(Offset)
	000 0111	(PFN = 7 — obtain by reading the PTE above)
<< 4	111 0000	(Physical Address without the offset)
OR	000 0101	(Offset)
	111 0101	(Physical Address = 117)

Physical memory address have 7 bits: 2^7 = 128 addresses.

```
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**

Problem:

Paging requires performing one extra memory reference

- Fetch translation from page table (in memory)
- Then fetch actual data from memory

Memory references are costly!

Will slow down processing by a factor of 2x or more…

18.5 A Memory Trace

```
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
```

Fig 18.7 show how these instructions access memory.

X-axis show the number of memory accesses per loop (5 first iterations)

- 10 memory access per iteration
- Breakdown: four instruction fetches, one data move, five page table lookups

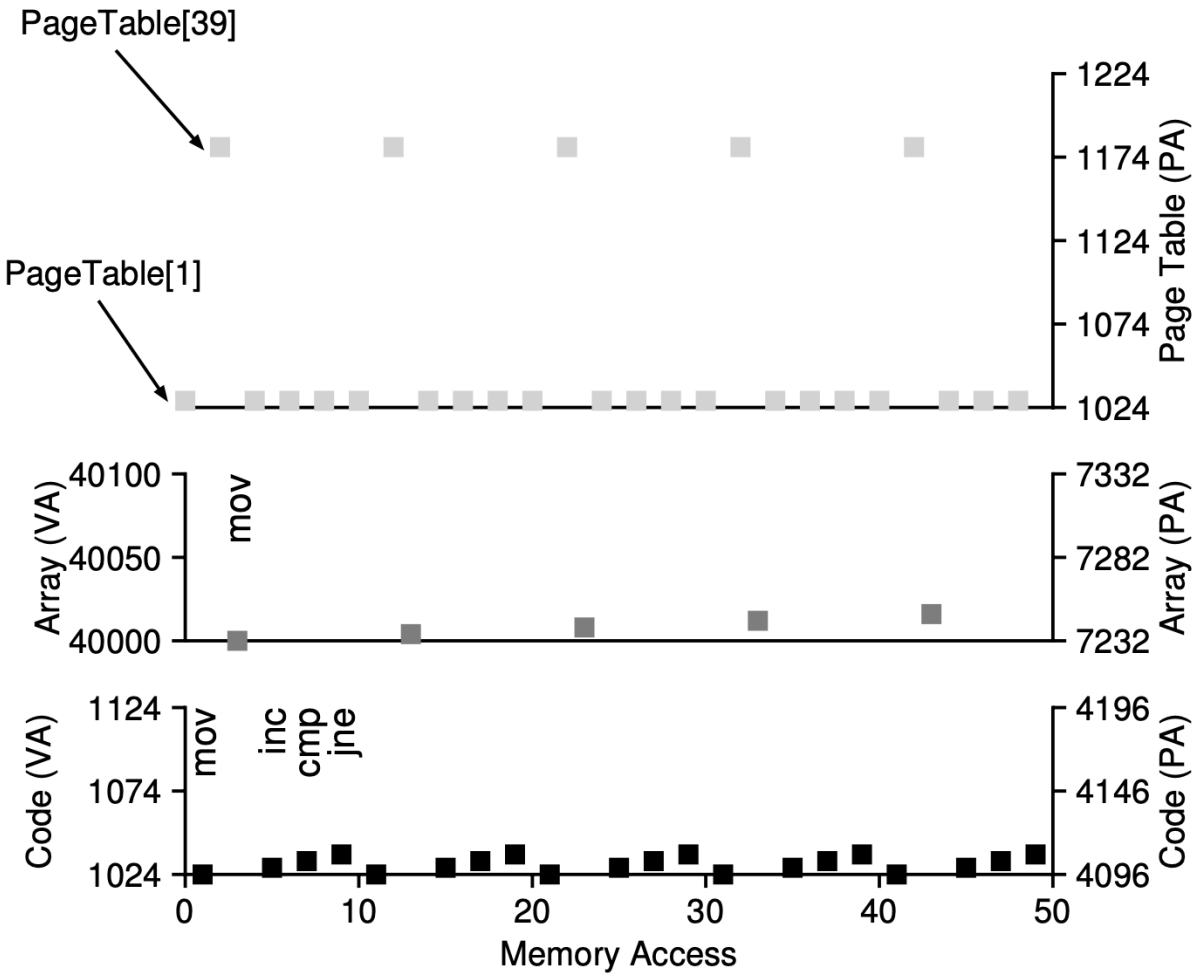


Figure 18.7: **A Virtual (And Physical) Memory Trace**