【OS】Day21

【Ch18】Paging: Introduction

The operating system takes one of two approaches when solving most any space-management problem. The first approach is to chop things up into variable-sized pieces, as we saw with segmentation in virtual memory.

Unfortunately, this solution has difficulties. When dividing a space into different-size chunks, the space itself can become fragmented, and thus allocation becomes more challenging over time.

Thus, it may be worth considering the second approach: to chop up space into fixed-sized pieces. In virtual memory, we call this idea paging.

Instead of splitting up a process's address space into some number of variable-size logical segments(e.g., code, heap, stack), we divide it into fixed-sized units, each of which we call a page. Correspondingly, we view physical memory as an array of fixed-sized slots called page frames; each of these frames can contain a single virtual-memory page.

18.1 A Simple Example and Overview

Let's look at an example. The figure below presents an example of a tiny address space, only 64 bytes total in size, with four 16-byte pages(virtual pages 0, 1, 2, and 3).

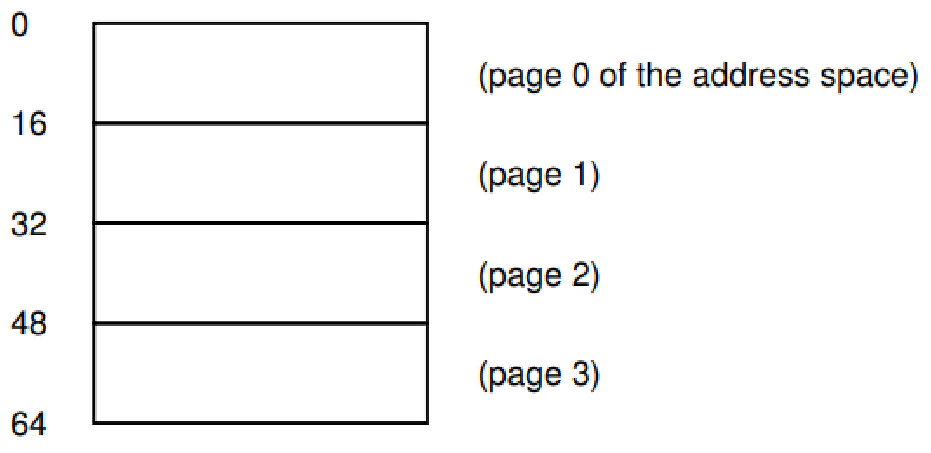


Figure 18.1: **A Simple 64-byte Address Space**

Physical memory, as shown below, also consists of a number of fixed-sized slots, in this case eight page frames.(making for a 128-byte physical memory).

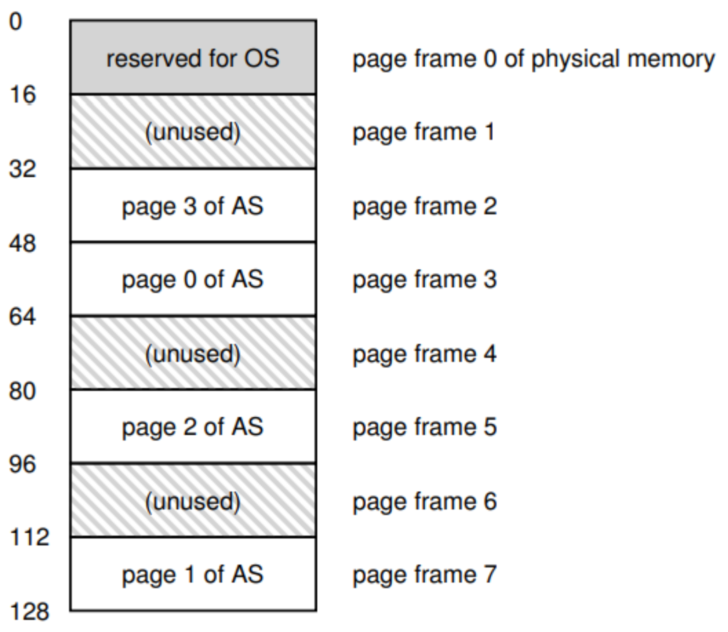


Figure 18.2: **A 64-Byte Address Space In A 128-Byte Physical Memory**

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

Paging has a number of advantages over our previous approaches:

- Flexibility: with a fully-developed paging approach, the system will be able to support the abstraction of an address space effectively, regardless of how a process uses the address space
- Simplicity of free-space management: For example, when the OS wishes to place our tiny 64-byte address space into our eight-page physical memory, it simply finds four free pages; perhaps the OS keeps a free list of all free pages for this, and just grabs the first four free pages off the list.

To record where each virtual page of the address space is placed in physical memory, the operating system usually keeps a per-process data structure known as a page table.

The major role of the page table is to store address translations for each of the virtual pages of the address space, thus letting us known where in physical memory each page resides.

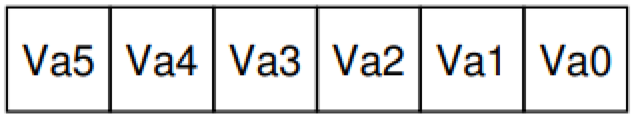
It is important to remember that this page table is a per-process data structure(an exception is the inverted page table). If another process were to run in our example above, the OS would have to manage a different page table for it.

Now, we know enough to perform an address-translation example. Let's imagine the process with that tiny address space(64 bytes) is performing a memory access:

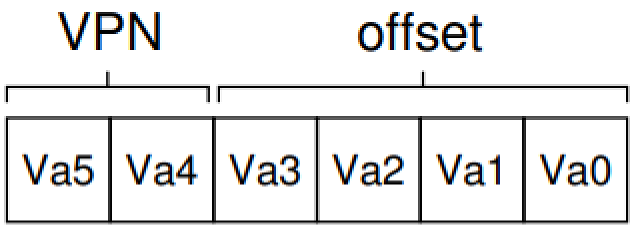
```
movl <virtual address>, %eax
```

To translate the 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:

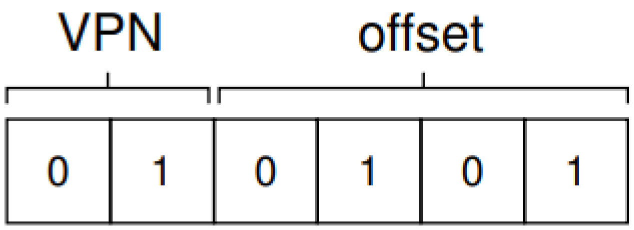


The page size is 16 bytes in a 64-byte address space; thus we need to be able to select 4 pages, and the top 2 bits of the address do just that. Thus, we have a 2-bit virtual page number(VPN).

When a process generates a virtual address, the OS and hardware must combine to translate it into a meaningful physical address. For example, let us assume the load above was to virtual address 21:

```
movl 21, %eax
```

Turning "21" into binary form, we get "010101", and thus we can examine this virtual address and see how it breaks down into a virtual page number and offset:



We can see that the virtual address "21" is on the 5th bytes of virtual page 01.

With our virtual page number, we can now index our page table and find which physical frame virtual page 1 resides within.

In the page table above the physical frame number(PFN) is 7(binary 111). Thus, we can translate this virtual address by replacing the VPN with the PFN and then issue the load to physical memory.

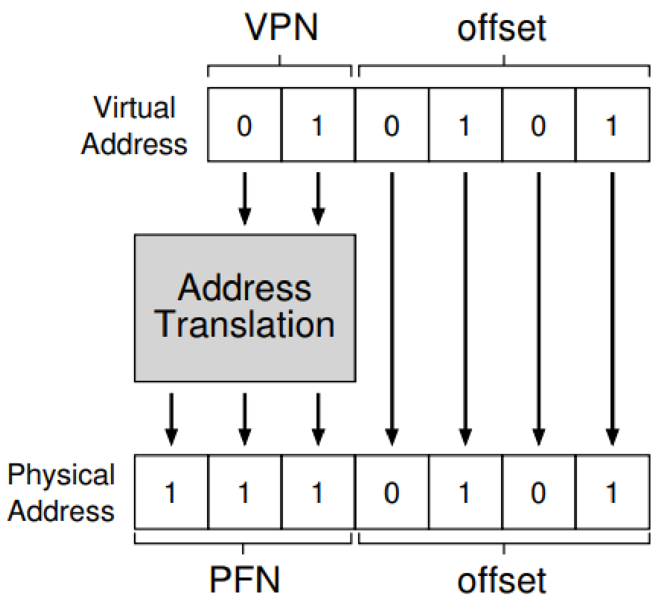


Figure 18.3: **The Address Translation Process**

Note that the offset stays the same(i.e. it is not translated) because the offset just tells us which byte within the page we want. Our final physical address is 1110101(117 in decimal)