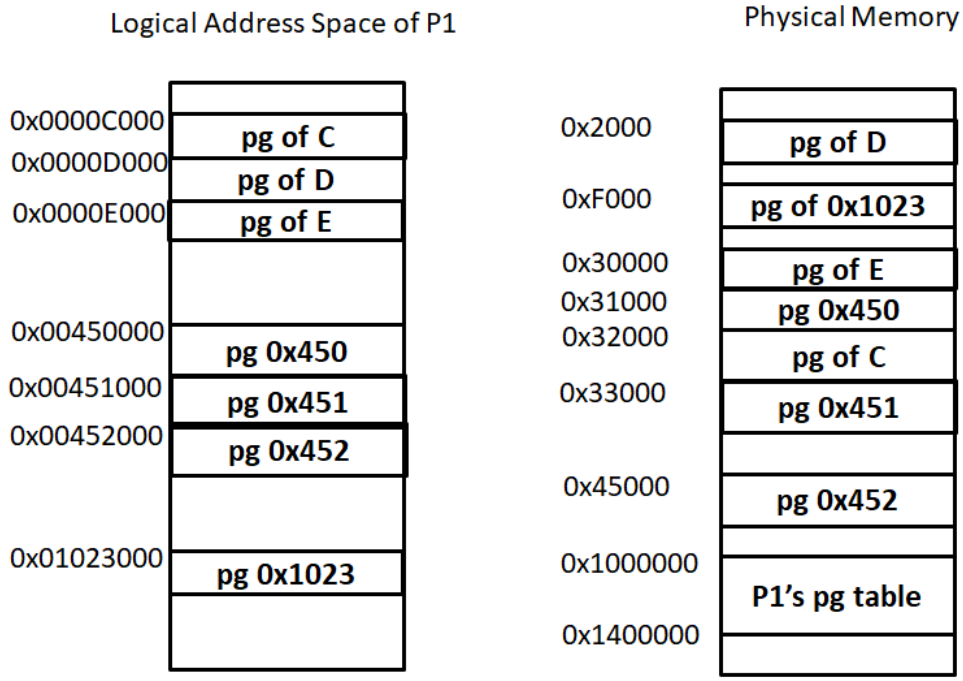
OS Paging

September 7, 2023

1 Example

Consider the following logical address space of process P1 (so P1 doesn't have any other allocated memory other than what is shown) and the contents of the physical memory space.



For this example, we assume the following

1. Size of logical address = 4 bytes = 32 bits
2. Size of physical address = 4 bytes = 32 bits
3. Size of page = 4KB = 2^{12} bytes.

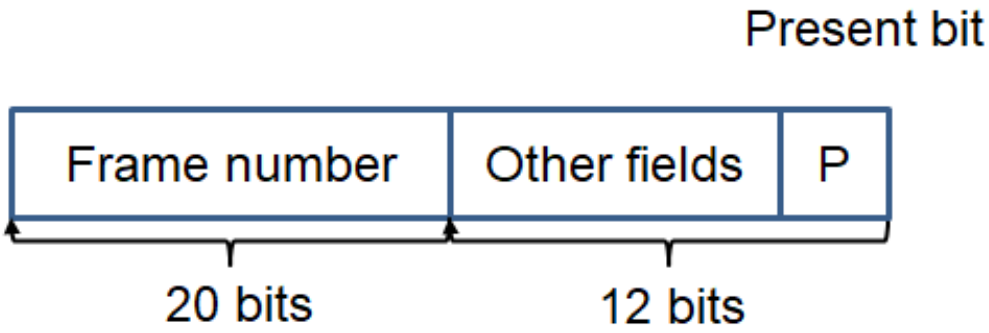
We must distinguish betweeen 2 different things.

1. Deciding which page is in which frame
2. Finding which page is in which frame.

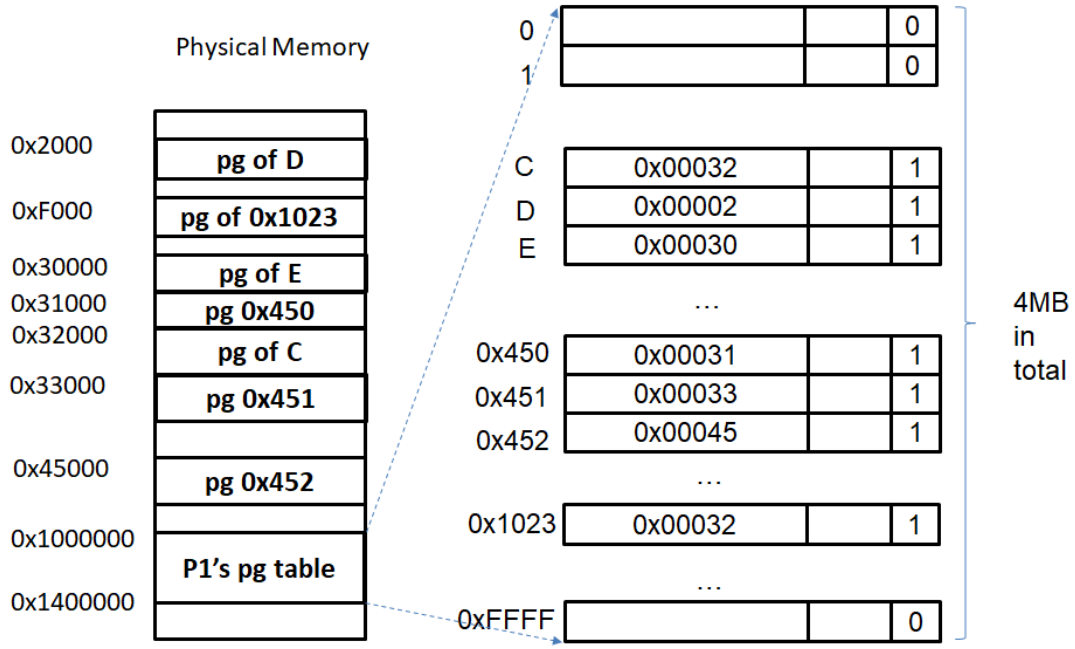
The main job of a MMU is the latter and NOT the former. So finding is the job and not deciding. A page can be stored in any frame. The main thing here is to clarify what the MMU (a piece of hardware) is supposed to do.
We now give example as to what the MMU should be able to do:

1. When given a logical address within page C, say 0xC236, the MMU should be able to find the corresponding address 0x32236.
2. When given a logical address within page B, say 0xB444, the MMU should be able to do a page fault indicating that MMU is unable to find a corresponding address for 0xB444 in the physical memory.

The main way by which the MMU performs these task is by looking up the page table. The page table should be thought of as a static array containing Page Table Entries (PTE). The following diagram shows a picture of a page table entry



Every page has it's own page table entry. Given a 32 bit logical address space where each page is 4KB, there are 2^{20} pages. You must work this out for yourself. We show the zoomed out picture of the page table in this case. Note that only the page table entries of page 0xC, 0xD, 0xE, 0x450, 0x451, 0x452 and 0x1023 have present bit set to 1.



Following the MMU operation described in the lecture slides, when P1 is running, the page table register will have an address of 0x1000000. For example, the logical address provided is 0x451052. The offset will be 0x052 while the page number is 0x451. Therefore, we need

to look up the 0x451 page entry which would have an address of 0x1000000 + 0x1144 = 0x1001144. (This is pointer arithmetic 0x451 × 4 = 0x2144.) Then, we will read the page table entry that will give us present bit as 1 and frame number as 0x33. However, so far, this is the 1-level paging scheme. It has the following disadvantages:

1. Page tables must be stored in physical memory and takes up 4MB of contiguous space. The OS needs to find 1K contiguous frames to store the page table. (How did I get the number 1K?)

2. Page tables then are not scalable in terms of memory usage. 1024 processes will require close to 4GB of physical memory space.

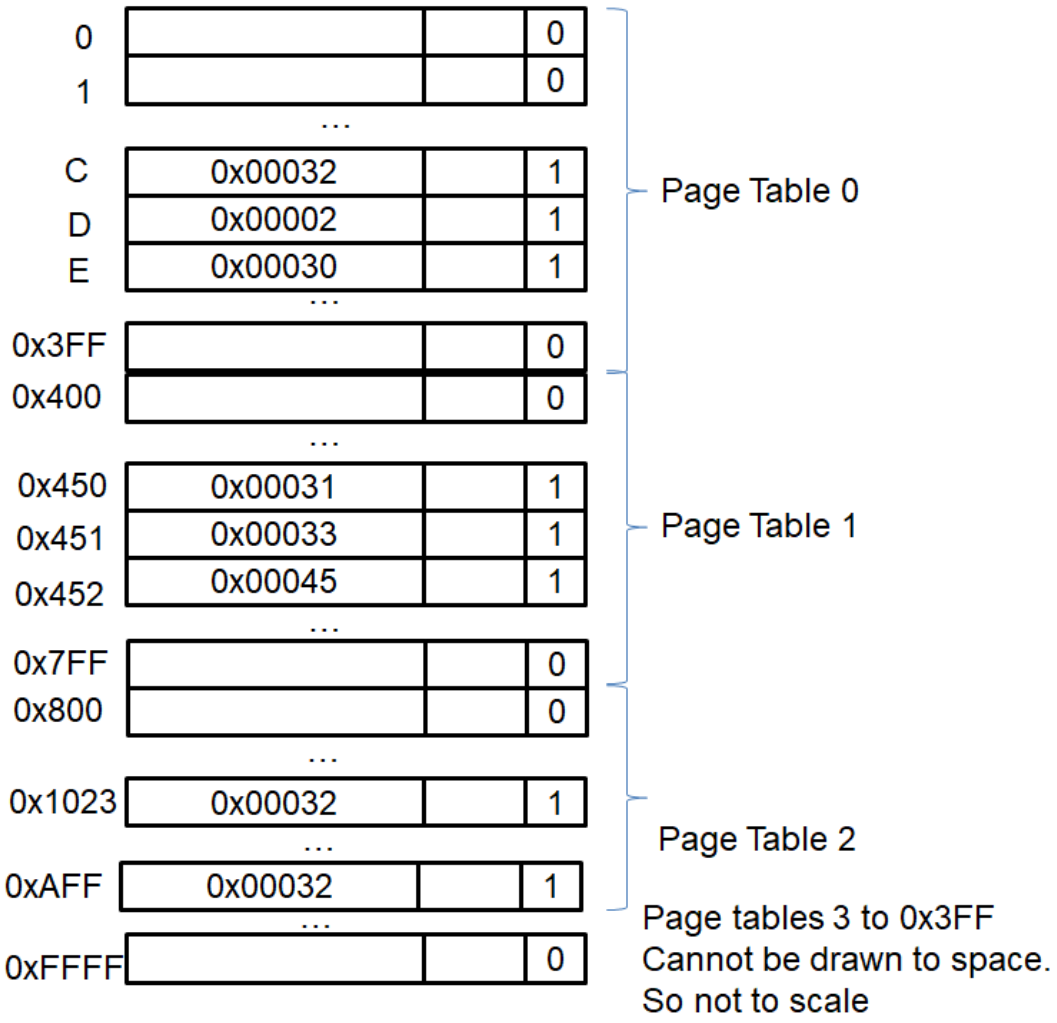
3. Page tables has tremendous internal fragmentation. According to the example above, the whole page table is 4MB. But actually only 28 bytes store useful frame numbers. The rest of the (2^{20}-7) page table entries are just storing rubbish values with present bit set to 0.

2-level paging over come the problem by doing 2 things:

1. Break down the original page table into 4KB chunks. This way, if all the page table entries in one 4KB chunk contains entries that say that present bit is 0, we don't have to store it in physical memory.

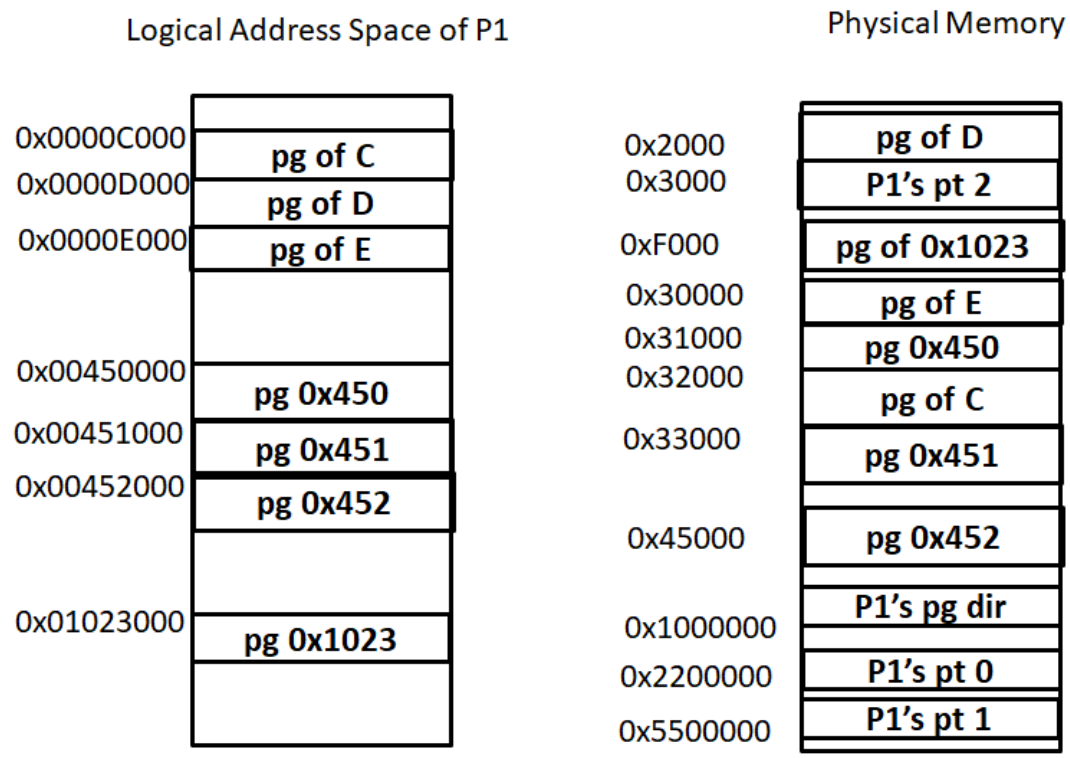
2. Because we are having 4KB chunks of the original page table possibly NOT in memory, we use a page directory to keep track whether each chunk is in memory or not.

Let's look closer at the original page table of P1. Every of the 4KB chunk of the original page table, we call "smaller" page tables. (Yeah, in 2-level paging each of these 4KB is a page table, but really it's a slice of the original 1-level page table.) So 1 original page table (4MB in size) is divided into 1024 "smaller" page tables. The following diagram shows how the original page table is divided up.



So, from the diagram above, we can see that only page tables 0,1 and 2 contains non-zero entries for the present bit. All the other 4KB page tables have page table entries containing 0 for the present bit.

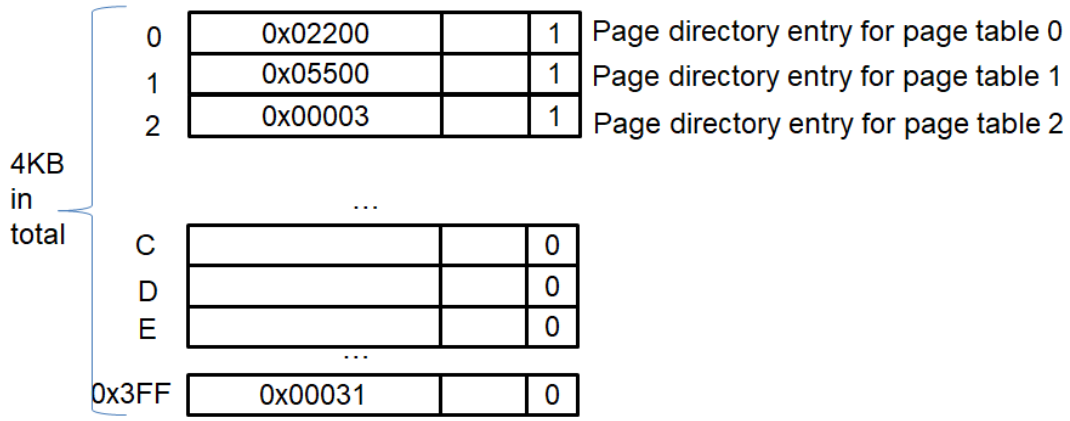
In 2-level paging, we don't have to store the page table 0x3 to page table 0x3FF in memory. Rather, we use another data structure, the page directory to keep track of this fact. NOTE: a page directory entry has exactly the same structure as a page table entry, only difference is that for the page table entry, the frame number stores where the page is found, while for the page directory, it stores where the 4KB "smaller" page table is found. The following shows the change in the physical memory when we use 2-level paging.



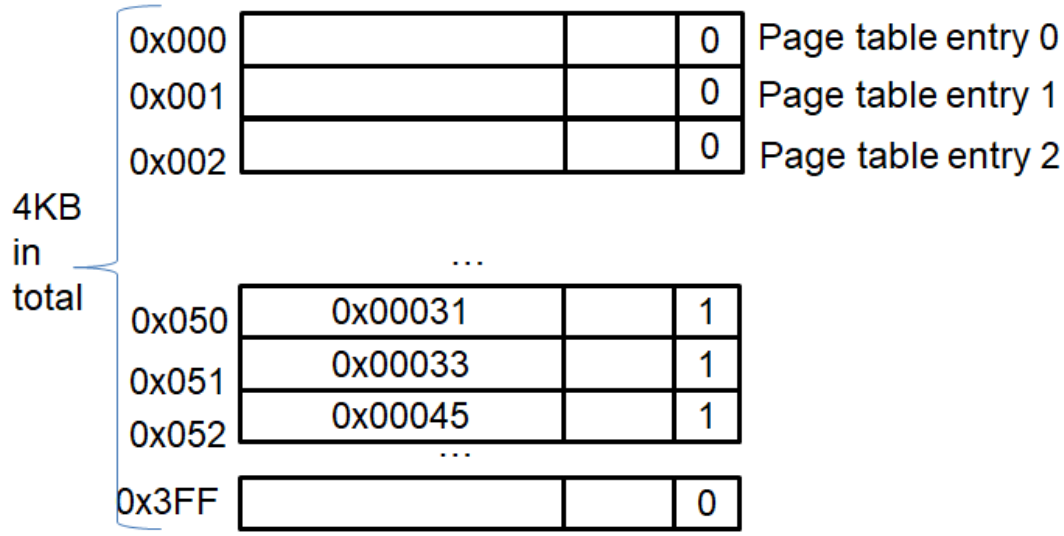
We make a few observations:

1. The size of the page directory is 4KB. How did we derive this? Well, the 4MB page table is divided into 1024 4KB page tables. So each of these "smaller" page tables has one entry in the page directory. Since each page directory entry is 4B in size, we have 4B $\times$ 1024 = 4KB.
2. We only store the page table 0, 1 and 2 in memory.
3. So the usage of physical memory is down from 4MB to 16KB.
4. Furthermore, the page tables do not have to be contiguous. We will show how this is done.

The following diagram shows the contents of the page directory table stored in address 0x1000000.



Therefore, in 2-level paging MMU, we store a page-directory-register that contains the address 0x1000000. When a logical address is provided, we use the first 10 bits to look up the page directory. For example, the physical address of the page directory entry for page table 1 is found in 0x1000004. When 32-bit logical address 0x450882 (0b0000 0000 0100 0101 0000 1000 1000 0010) is provided, the first 10 bits of this is 0x1. We multiply 1 by 4 in 12 bits and append to the frame number to derive 0x1000004. Then we read the page directory entry to find that page table 1 is found at in frame 0x05500. But the next 10 bits is 0x50, so we look up the 0x50 entry of page table 1. Recall that page table 1 is the following:



And the 0x050 entry of page table 1 is truly the page table entry of 0x450 page! Hence, we take the frame and concatenate with the offset, we get 0x00031882, which is indeed a physical address of page 0x450 that is stored in frame 0x31.