Segmentation

- Programming with multiple segments is hard
- The virtual address space is limited to the size of RAM
- Paging mitigates the problems with the segmentation

Paging

- The virtual address space is the same as physical address space
- The virtual address space and physical address space are divided into 4 KB blocks called pages
- The paging hardware maintains a mapping from a virtual page number (VPN) to the physical page number (PPN)

Paging

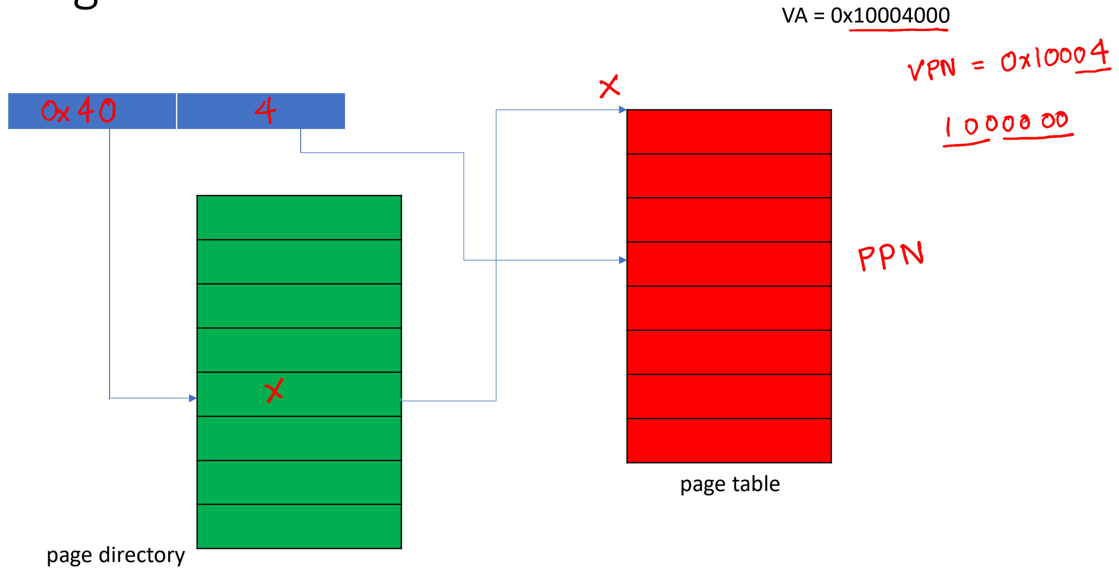
- Total number of virtual or physical pages are 2^{20} (4 GB/ 4 KB)
- The page table contains VPN to PPN mappings
- If the page table is implemented using a one-dimensional array, then 2^{20} contiguous entries are needed in the page table
- A two-dimensional page table doesn't require large contiguous space

Page table

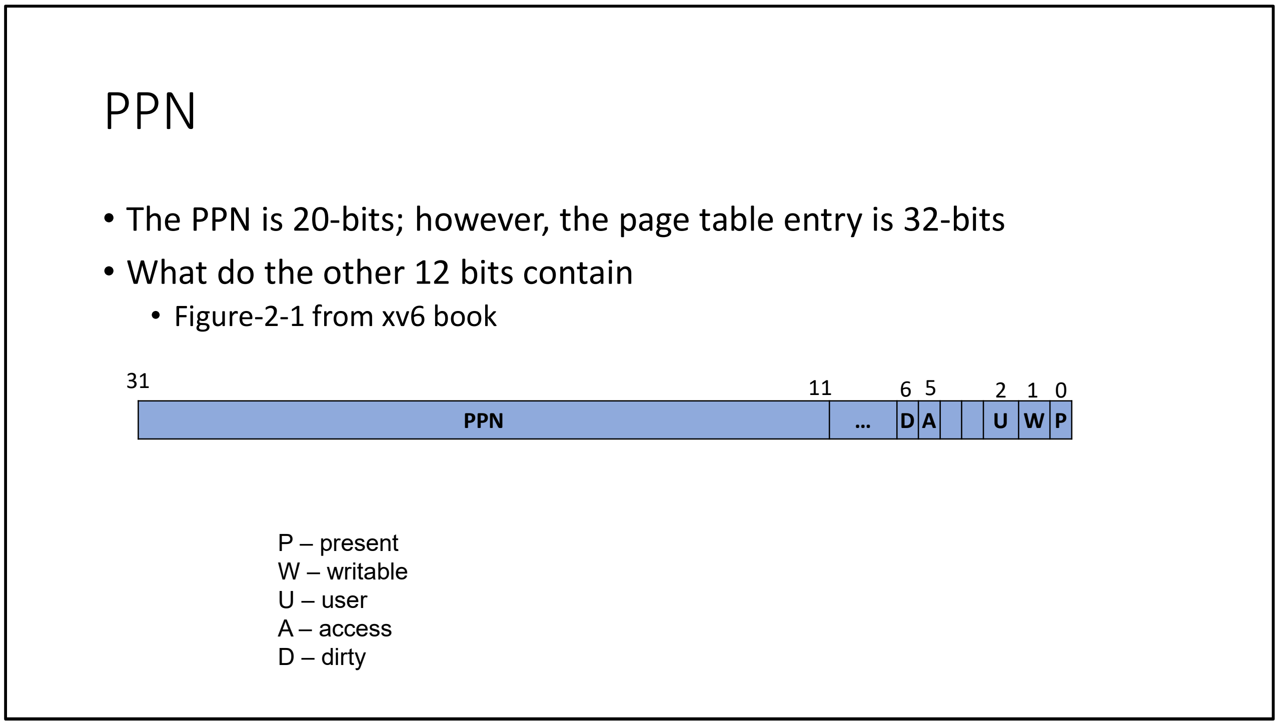
- two-dimensional page tables
  - The top 10-bits of the VA are used to index in a page directory
  - page-directory contains the physical addresses of a page table
  - The next 10-bits (after top 10 bits) are used to index in the page table
  - The corresponding entry in the page table contains the physical address

- Read Section-4.3 from Intel manual-3

Page table

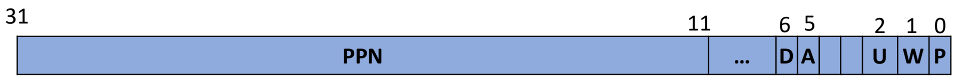


The top 20 bits of the VA (0x40 in this example) are indexed in the page directory to find the physical address of the page table. The next 10 bits (after the top 10 bits, 4 in this example) are indexed in the page table to find the physical address corresponding to the virtual page. The lower 12-bits (offset within a page) are equal in both virtual and physical addresses.



PPN

- The PPN is 20-bits; however, the page table entry is 32-bits
- What do the other 12 bits contain
  - Figure-2-1 from xv6 book



P – present
W – writable
U – user
A – access
D – dirty

Because the page table contains VPN to PPN mappings, only 20 bits are required to store the PPN. However, the page table uses 32-bits to store the PPN. The top 20-bits in a page table entry is PPN, and the rest bits are the other flags. The meaning of some of these flags is given on the next slide. Top 20-bits in the page directory contains the PPN of the page table page, and the lower 12 bits contain flags (some of them are similar to the page table entry).

Page table entry

- The top 20-bits in a page table entry contain the PPN
- The lower 12-bits contain other information about the page
  - e.g., whether page table entry is valid
  - read/write access
  - page was accessed at least once
  - page was written at least once
  - page can be accessed by the user, etc.

Page table

- The OS creates a page table for every process

- How does OS support process isolation?
  - two different page tables do not contain the same physical page

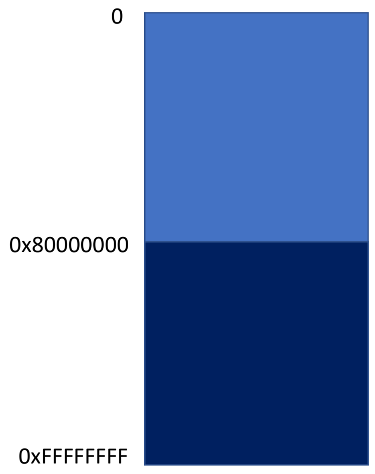
- Where does the page table live?
  - In OS address space (otherwise the process can modify the page table)

Page table

- Who creates the page table?
  - OS
- Who walks the page table?
  - Hardware
- Kernel accesses a virtual address or a physical address
  - virtual
- Can we map the OS pages and user pages inside the same page table?
  - Yes
- Why can’t user access kernel pages?
  - The user flag in the page table entries corresponding to the kernel virtual pages are not set

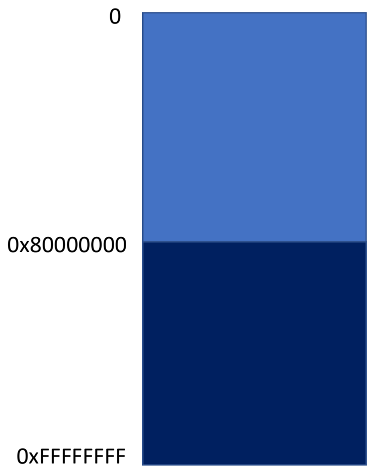
Process address space

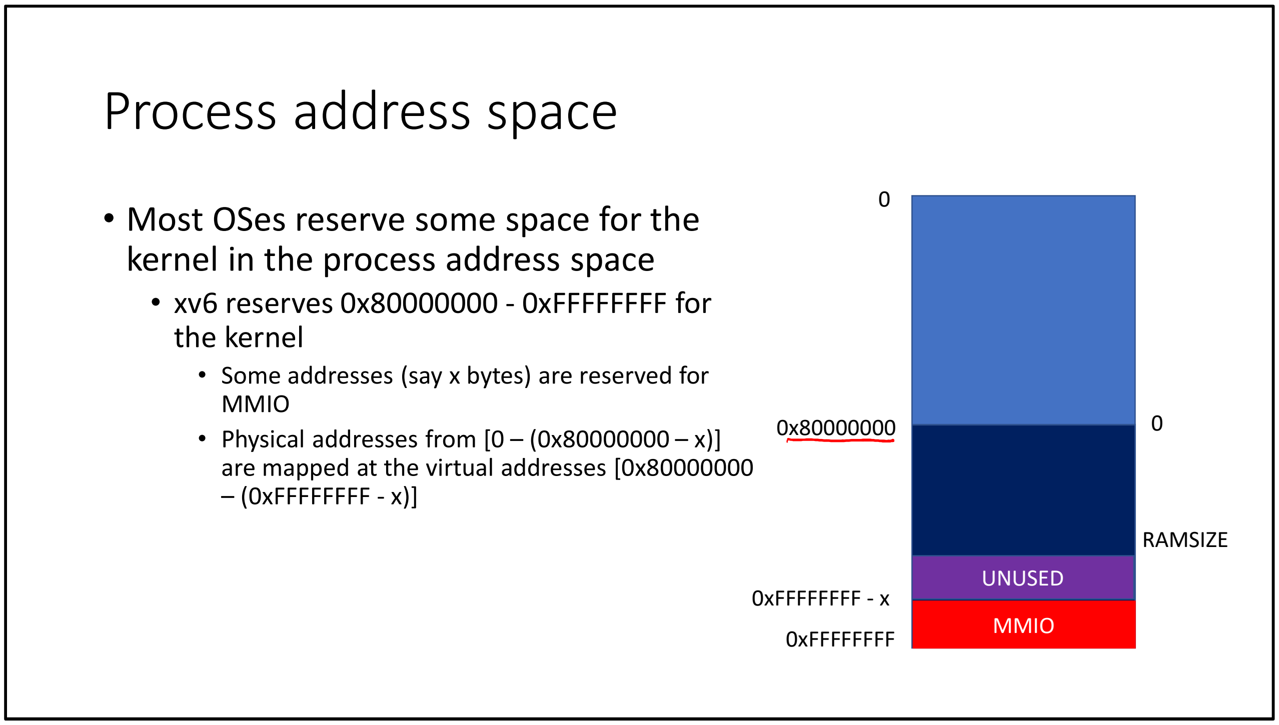
- Most OSes reserve some space for the kernel in the process address space
  - xv6 reserves 0x80000000 - 0xFFFFFFFF for the kernel
  - Linux reserves 0xC0000000 – 0xFFFFFFFF for the kernel
  - Windows also does something similar



Process address space

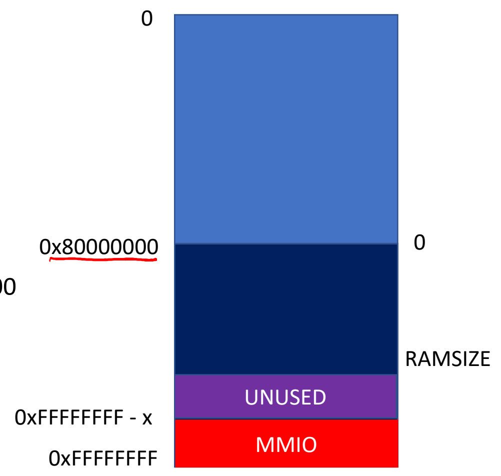
- Most OSes reserve some space for the kernel in the process address space
  - xv6 reserves 0x80000000 - 0xFFFFFFFF for the kernel
  - Linux reserves 0xC0000000 – 0xFFFFFFFF for the kernel
  - Windows also does something similar
- How does OS prevent processes from accessing its memory
  - User/supervisor flag in a page table entry
  - User can access a page only if the user bit in the page table entry is set



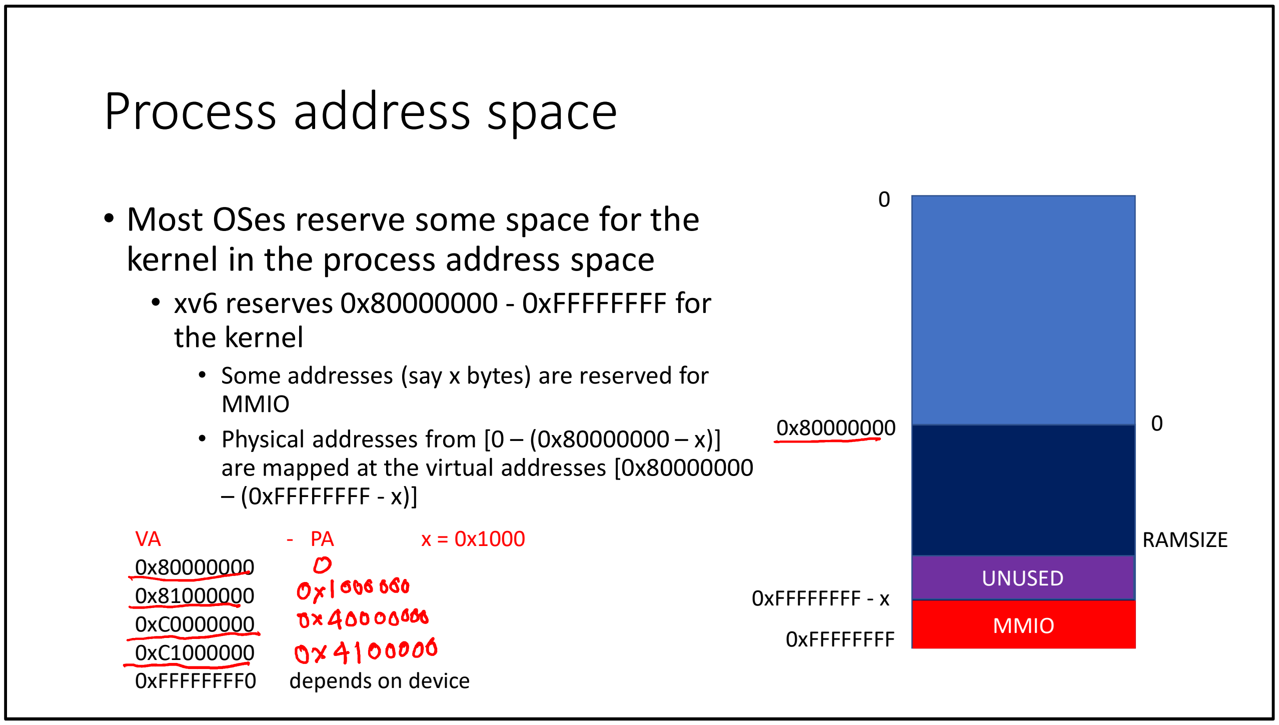


Process address space

- Most OSes reserve some space for the kernel in the process address space
  - xv6 reserves 0x80000000 - 0xFFFFFFFF for the kernel
    - Some addresses (say x bytes) are reserved for MMIO
    - Physical addresses from [0 – (0x80000000 – x)] are mapped at the virtual addresses [0x80000000 – (0xFFFFFFFF - x)]



The virtual addresses in the range 0x80000000 – 0xFFFFFFFF, are reserved for the OS. These pages are protected from the user using the user flag in the page table entry. Some virtual addresses (say x bytes) towards are end of the virtual address space are reserved for the device MMIO. In this example, [0xFFFFFFFF-x, 0xFFFFFFFF] are reserved for MMIO. Physical addresses in the range [0, 0x7FFFFFFF - x] are mapped at the virtual addresses [0x80000000, 0xFFFFFFFF-x]. This scheme makes the address translation (virtual to physical and vice-versa) for kernel addresses straightforward, which can be done merely by an addition/subtraction.

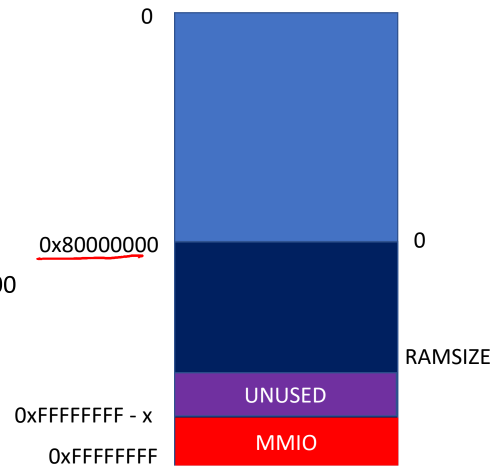


Process address space

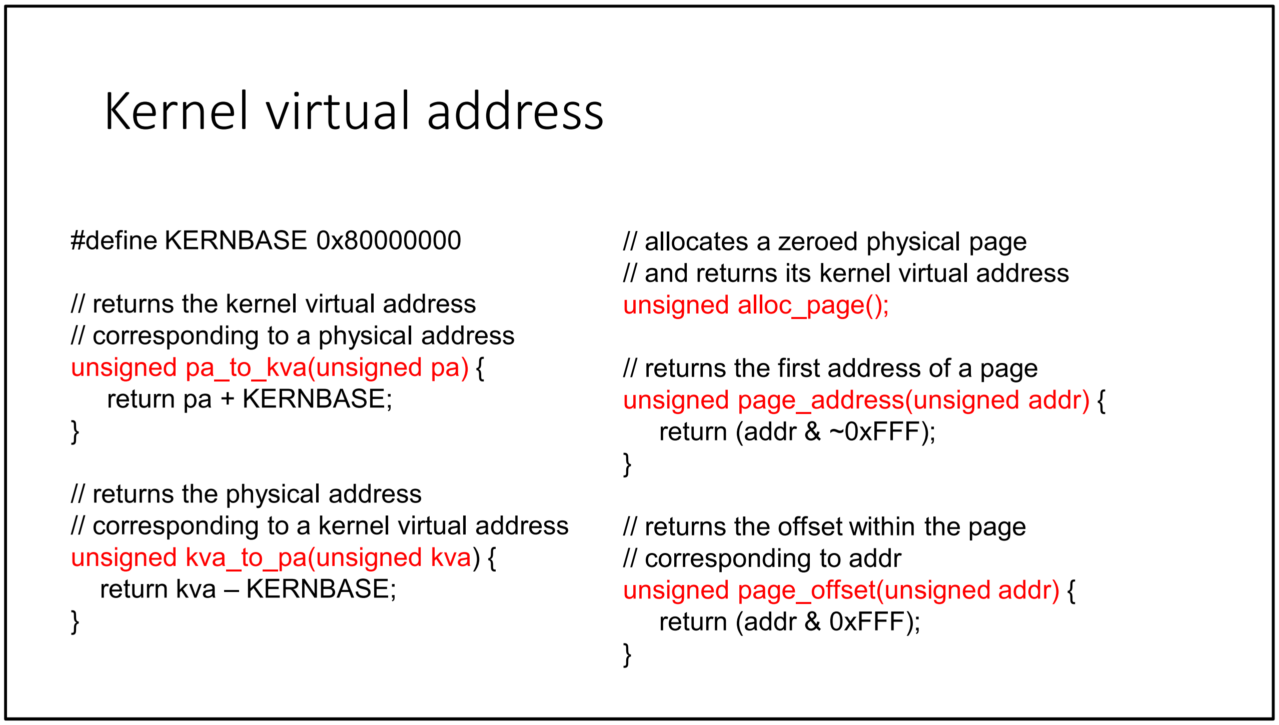
- Most OSes reserve some space for the kernel in the process address space
  - xv6 reserves 0x80000000 - 0xFFFFFFFF for the kernel
    - Some addresses (say x bytes) are reserved for MMIO
    - Physical addresses from [0 – (0x80000000 – x)] are mapped at the virtual addresses [0x80000000 – (0xFFFFFFFF - x)]

x = 0x1000

VA	- PA
0x80000000	0
0x81000000	0x1000000
0xC0000000	0x40000000
0xC1000000	0x41000000
0xFFFFFFFF0	depends on device



To convert a kernel virtual address to the physical address, we just need to subtract 0x80000000 from the kernel virtual address. However, for MMIO pages, this mapping could be different because the MMIO addresses (which are part of physical address space) depends on which devices we are using.

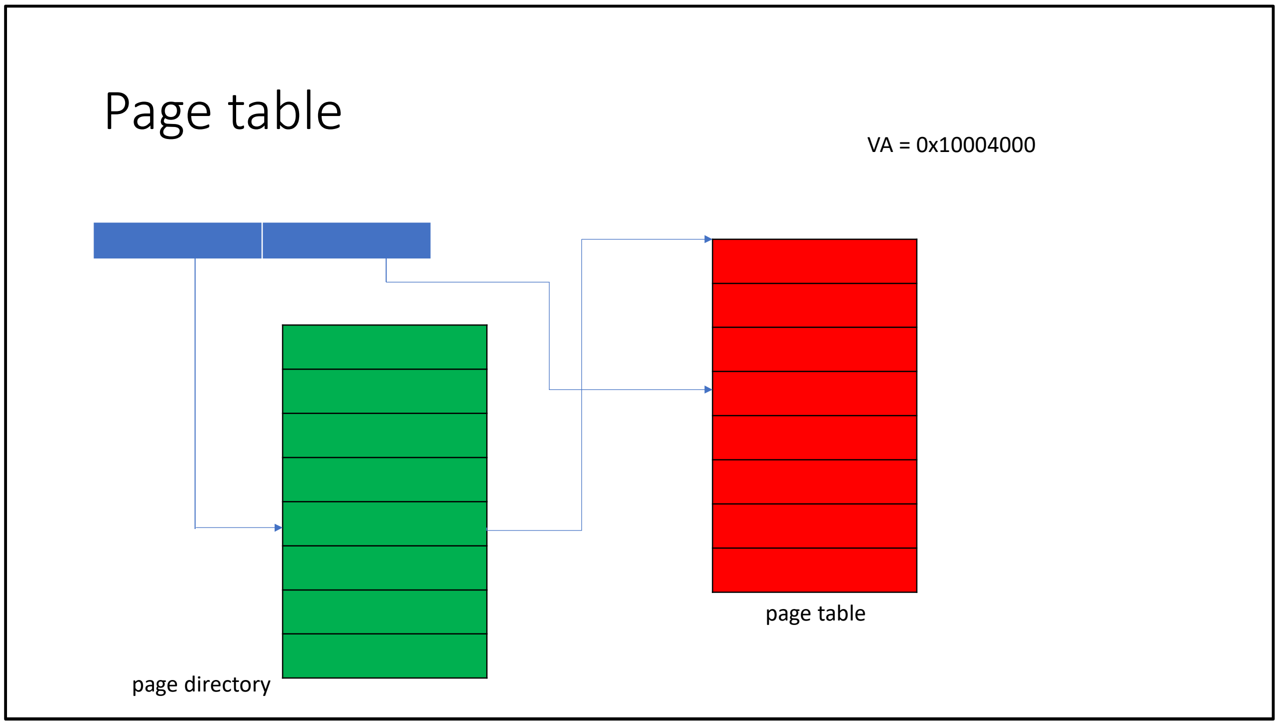


Kernel virtual address

```
#define KERNBASE 0x80000000

// returns the kernel virtual address
// corresponding to a physical address
unsigned pa_to_kva(unsigned pa) {
    return pa + KERNBASE;
}

// returns the physical address
// corresponding to a kernel virtual address
unsigned kva_to_pa(unsigned kva) {
   return kva – KERNBASE;
}
```

```
// allocates a zeroed physical page
// and returns its kernel virtual address
unsigned alloc_page();

// returns the first address of a page
unsigned page_address(unsigned addr) {
    return (addr & ~0xFFF);
}

// returns the offset within the page
// corresponding to addr
unsigned page_offset(unsigned addr) {
    return (addr & 0xFFF);
}
```

Here are some routines that we are going to use in our further discussions on page tables. The way we have mapped the physical pages in the kernel virtual address space, the implementations of these routines are very simple, that otherwise, would have very complicated (requires a page table walk). alloc_page routine allocates a physical page (a physical address aligned to 4 KB) and returns the kernel virtual address of that page.



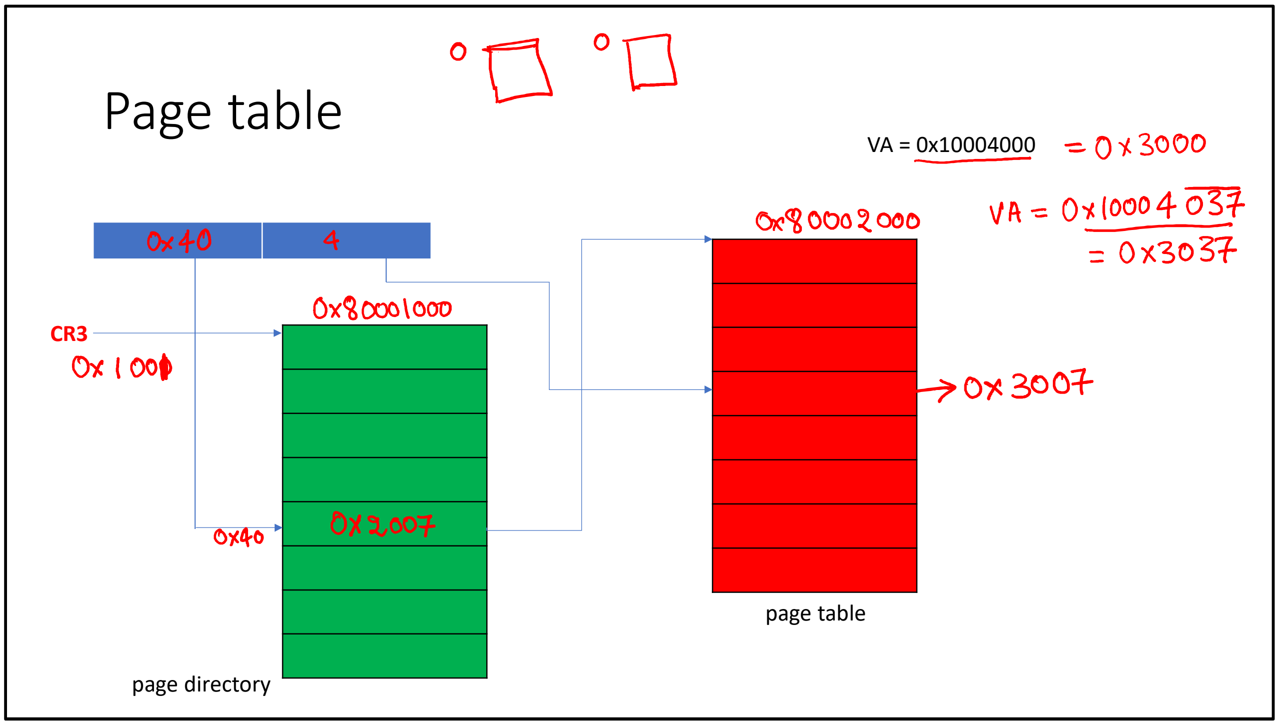
Page table

VA = 0x10004000

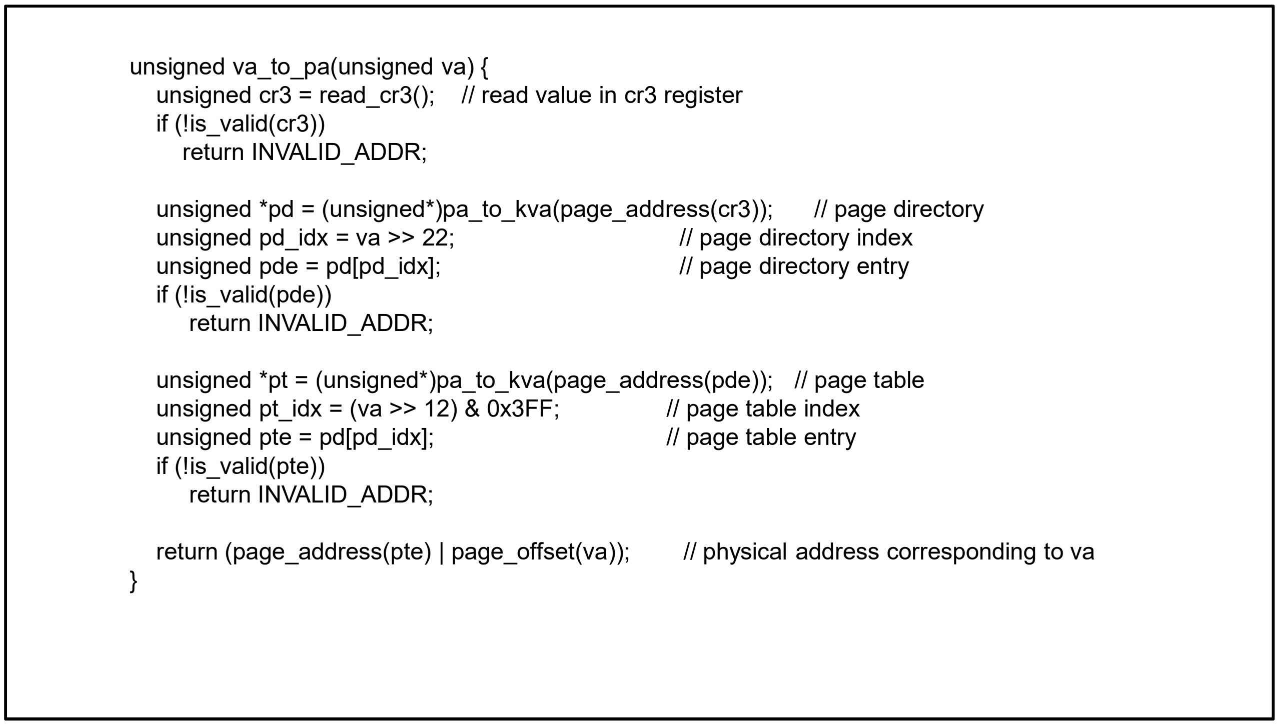
The question is, how does the hardware find the address of the page directory.

Page table

- Where is the base address of the page directory?
  - In the %cr3 register
  - top 20 bits contain the PPN of the page directory
  - lower 12 bits contain other flags (e.g., valid)
  - why %cr3 contains a physical address
    - chicken and egg problem
- How to load the %cr3 register?
  - mov %eax, %cr3   // any GPR can be used instead of %eax
- Why user application can't load a new page table?
  - mov to cr3 is a privilege instruction



This example shows how we can walk the page table in the software to find the physical address. The indices of page directory and page table are 0x40 and 4, respectively. In this example, the cr3 contains 0x1001. Here the last 12-bits of the cr3 are flags, and the top 20-bits are the physical address of page directory. To access the page directory, we need to find the virtual address of the page directory. In the xv6 kernel, the pa to kva (kernel virtual address) translation is done by adding 0x80000000 to the physical address. After obtaining the virtual address of the page directory (0x80001000), we can read the page table PPN (0x2, lower 12 bits are flags) at index 0x40 in the page directory. The virtual address of the page table is computed by adding the 0x80001000 to the physical address of the page table page. After obtaining the page table address (0x80002000), we compute the PPN (3, last 12 bits are flags) from the page table entry at index 4. The physical address corresponding to the VA 0x10004000 is 0x3000. The physical address corresponding to the VA 0x10004037 is 0x3037 (because page offset in the virtual and physical pages are the same).



```
unsigned va_to_pa(unsigned va) {
    unsigned cr3 = read_cr3();    // read value in cr3 register
    if (!is_valid(cr3))
        return INVALID_ADDR;

    unsigned *pd = (unsigned*)pa_to_kva(page_address(cr3));      // page directory
    unsigned pd_idx = va >> 22;                               // page directory index
    unsigned pde = pd[pd_idx];                                // page directory entry
    if (!is_valid(pde))
         return INVALID_ADDR;

    unsigned *pt = (unsigned*)pa_to_kva(page_address(pde));   // page table
    unsigned pt_idx = (va >> 12) & 0x3FF;                    // page table index
    unsigned pte = pd[pd_idx];                               // page table entry
    if (!is_valid(pte))
         return INVALID_ADDR;

    return (page_address(pte) | page_offset(va));        // physical address corresponding to va
}
```

This slide shows the implementation of the va_to_pa routine. va_to_pa takes a virtual address as input and returns the corresponding physical address. First, the physical address of the page directory is computed (by reading cr3 value). If the value in cr3 is valid, the physical address can be computed by resting the lower 12-bits of the cr3 value. To access the page directory, we compute the kernel virtual address of the page directory. We compute the index of the page directory and read the value at that particular index in the page directory. If the value is valid, the first we calculate the physical address of the page table from the page directory entry, and then we compute the corresponding kernel virtual address of the page table. Finally, we read the value at the page table index (computed from the virtual address) in the page table and check its validity. If the entry is valid, then the address of the physical page is computed by resetting the flag bits (lower 12 bits). The physical address is computed after adding the offset in the virtual page to the physical page address.

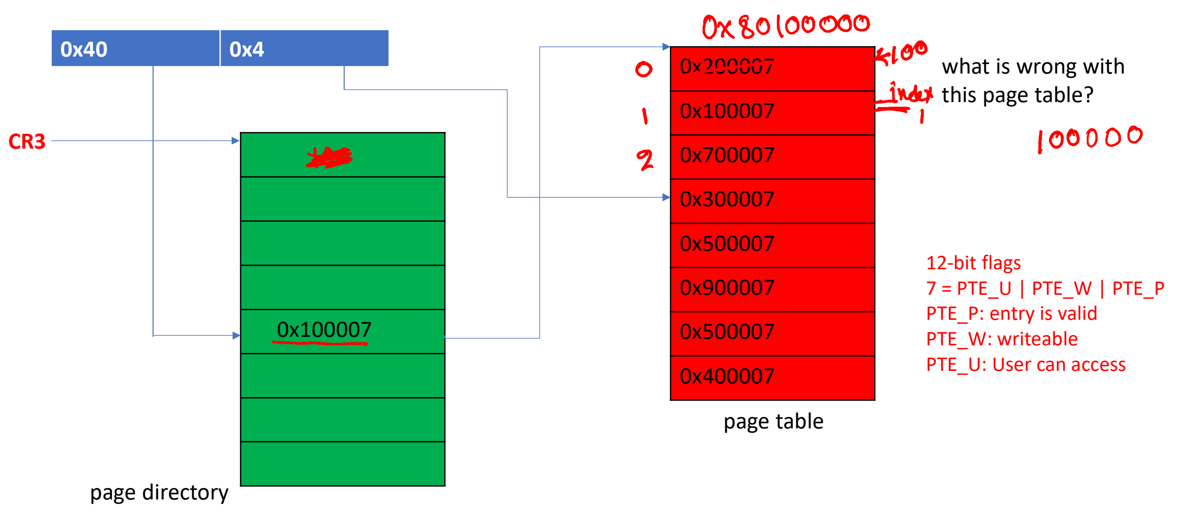
Page table

- Why can a user not modify the page table pages?
  - page table pages are not mapped corresponding to the user virtual pages

Page table

VA = 0x10004000

VA= 0x 10001000 = 100



what is wrong with this page table?

12-bit flags
7 = PTE_U | PTE_W | PTE_P
PTE_P: entry is valid
PTE_W: writeable
PTE_U: User can access

In this example, the physical address (0x1000000) is the address of a page table. Because this physical address is also mapped at index 1, in the page table (which is a user-accessible entry) at index 0x40 in the page directory, the user can overwrite the page table by writing to virtual page 0x10001000. If the users can modify the page table, then they can access any physical address by modifying the page table. The kernel doesn't let this happen by not mapping page table pages in user virtual address space.