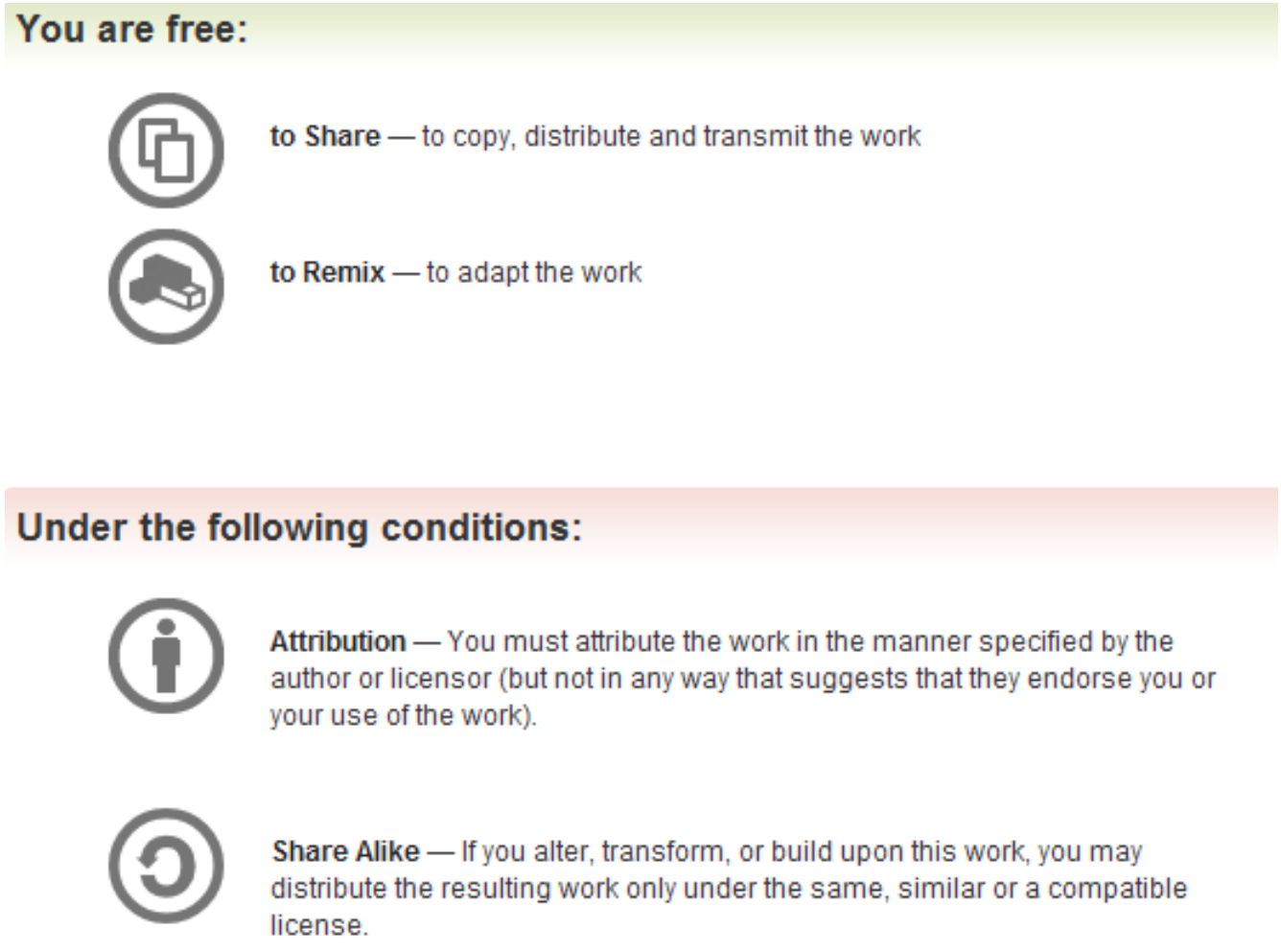
Intermediate x86
Part 2

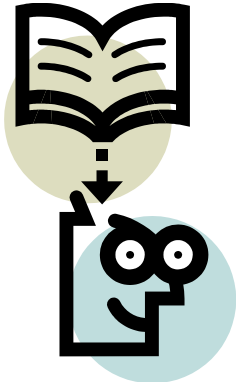
Xeno Kovah – 2010

xkovah at gmail

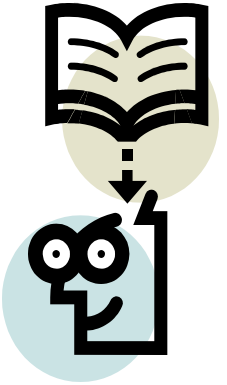
All materials are licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/





Paging



Previously we discussed how segmentation translates a logical address (segment selector + offset) into a 32 bit linear address.

- When paging is disabled, linear addresses map 1:1 to physical addresses.
- When paging is enabled, a linear address must be translated to determine the physical address it corresponds to.
- It's called "paging" because physical memory is divided into fixed size chunks called pages.
- The analogy is to books in a library. When you need to find some information first you go to the library where you look up the relevant book, then you select the book and look up a specific page, from there you maybe look for a specific specific sentence …or "word"? ;)
- The internets ruined the analogy!

Notes and Terminology

- All of the figures or references to the manual in this section refer to the Nov 2008 manual (available in the class materials). This is because I think the manuals <= 2008 organized and presented much clearer than >= 2009 manuals.
- When I refer to a “frame” it means “a page-sized chunk of physical memory”
- When paging is enabled, a “linear address” is the same thing as a “virtual memory address” or “virtual address”

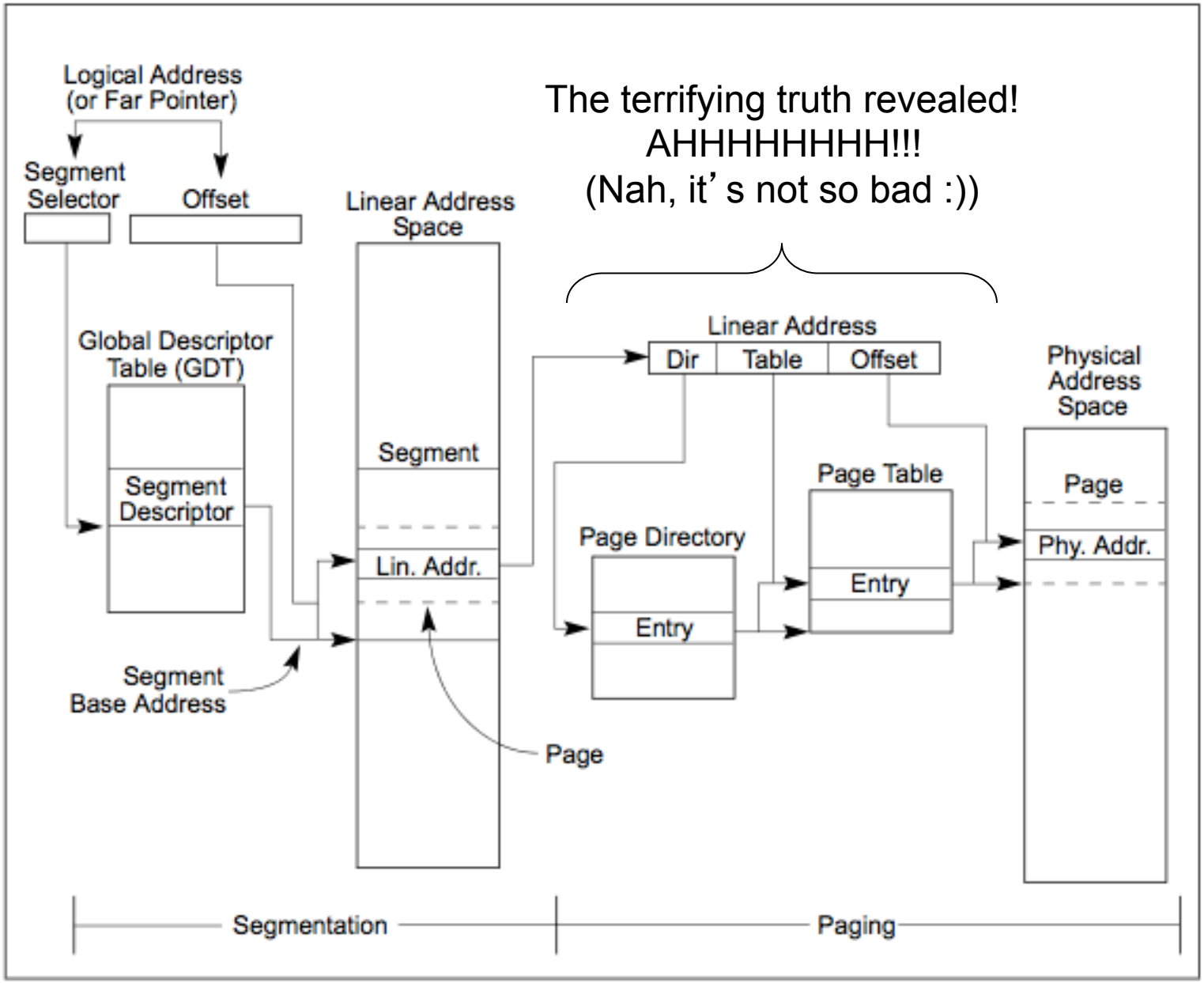


Figure 3-1. Segmentation and Paging

Virtual Memory

- When paging is enabled, the 32 bit linear address space can be mapped to a physical address space less than 32 bits. Whippersnappers need to understand that for most of history it was cost-prohibitive to have 4GB of RAM.
- Paging makes memory access virtual in that no longer does the linear address correspond to the exact same physical address. Low addresses can map to high addresses, or low addresses, or no addresses.
- Remember I said I will use “linear address” and “virtual address” interchangeably

Ways to translate from linear to physical memory

- Traditional
  - 32 bit linear address to 32 bit physical address space, 4KB pages
  - 32 bit linear address to 32 bit physical address space, 4MB pages
- Physical Address Extension (PAE) allows you to address 64GB of RAM (2^36 bytes)
  - 32 bit linear address to 36 bit physical address space, 4KB pages
  - 32 bit linear address to 36 bit physical address space, 2MB pages
- Page Size Extension (PSE-36) is another way to address 2^36 bytes, but we ignore it because I don't know that anyone uses it

Paging and the Control Registers

("<Anyone> and The Control Registers" could make a good band name)

- There are 5 Control Registers (CR0-CR4) which are used for paging control as well as enabling/disabling other features.
- CR0 has the PG bit which if set to 1 enables paging. It also contains the PE bit which if set to 1 enables protected mode (PG requires PE being set)
- CR1 is reserved and not used for anything (lame)
- CR2 stores the linear address that caused a page fault (discussed later)

Paging and the Control Registers 2

- CR3 pointer to page directory. AKA Page-Directory Base Register (PDBR).
- CR4
  - PAE - Physical Address Extension - enable/disable the ability to address 2^36 bytes of physical memory (64 GB) instead of just 2^32
  - PSE - Page Size Extentions - Enable/disable the use of large (4MB or 2MB) pages (Don't confuse with that PSE-36 thing I told you about a little while ago.)
  - PGE - Page Global Enable - Enable/disable global page feature.
- We'll mention all of the CR4 flags of interest again when we get to them later.

Accessing Control Registers

- Accessed with their own MOV instructions (only allowed from ring 0)
  - MOV CR, r32
  - MOV r32, CR
- Different opcodes than normal MOVs, no fancy r/m32 form. Only register to register

32bit to 32 bit, 4KB pages

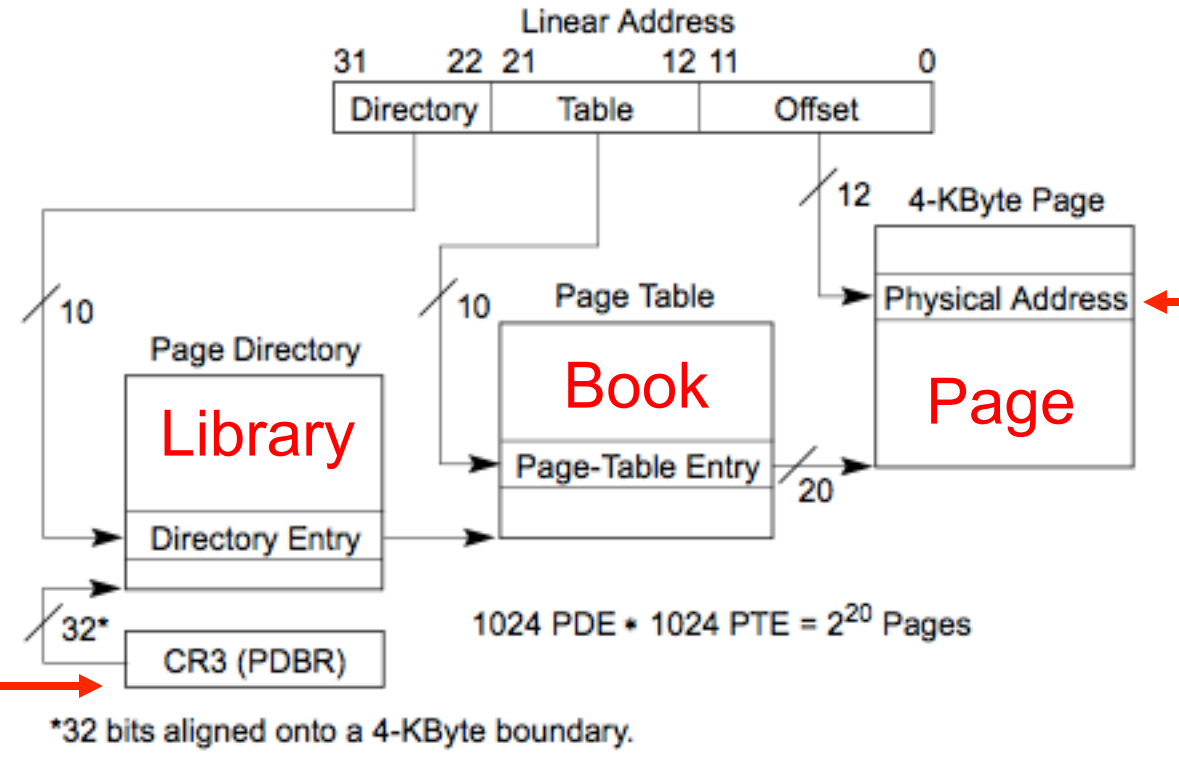


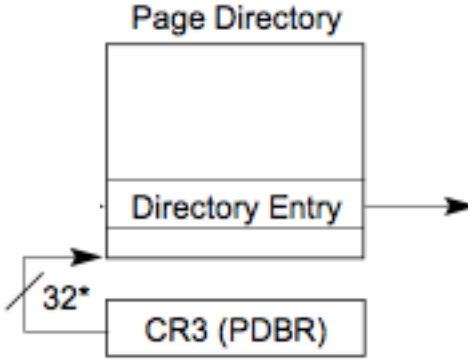
Figure 3-12. Linear Address Translation (4-KByte Pages)

Location in the page

The nerd who always knows where the library is. e.g. me ;)

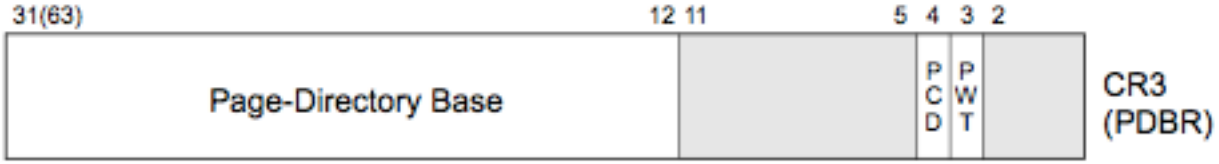
- This is the process by which linear addresses are translated to physical addresses.
- The OS sets the tables up, and the hardware automatically traverses them whenever you try to access a linear (virtual) address.

Stop! Break it down! What's going on here? CR3

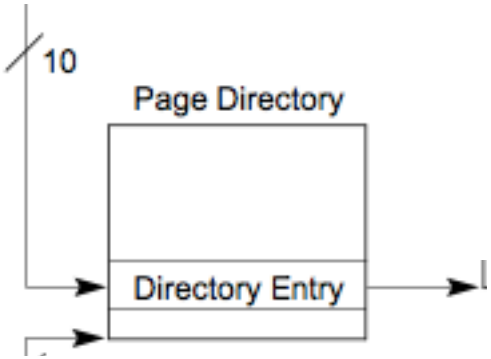


*32 bits aligned onto a 4-KByte boundary.

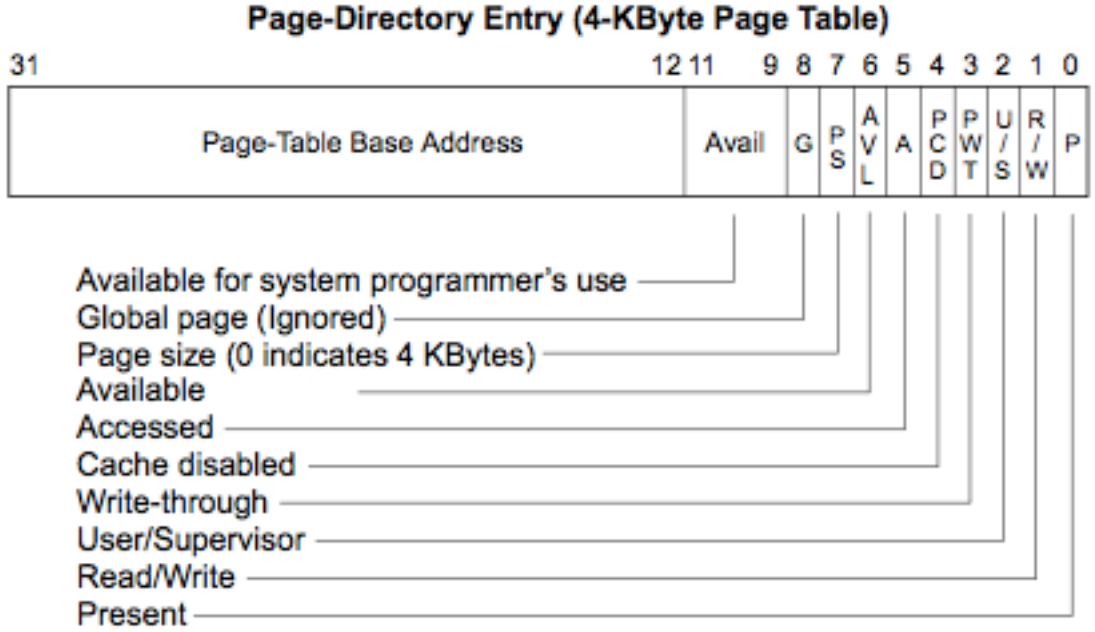
- We start at CR3, which always should point to the current process' page directory
  - That's right, I said current process'. As in, each process can get it's own page directory, and thus it's own view of memory.
- Note the * text in the picture says the address is aligned on a 4KB boundary. This is because the address is actually only specified in the upper 20 bits of CR3, and then the bottom 12 bits (2^12 = 4KB) of the address are assumed to be 0. (That leaves the bottom 12 bits of CR3 available for misc usage, and indeed there are a couple flags for caching which we can ignore)
- Annoying part: The value in CR3 is a physical address



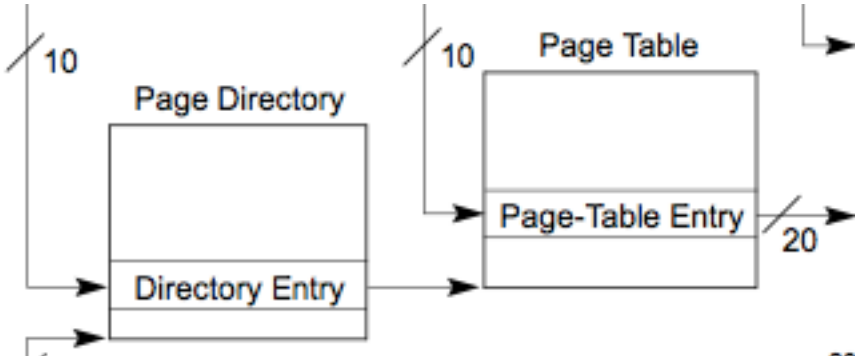
What's going on here?: Page Directory Entry



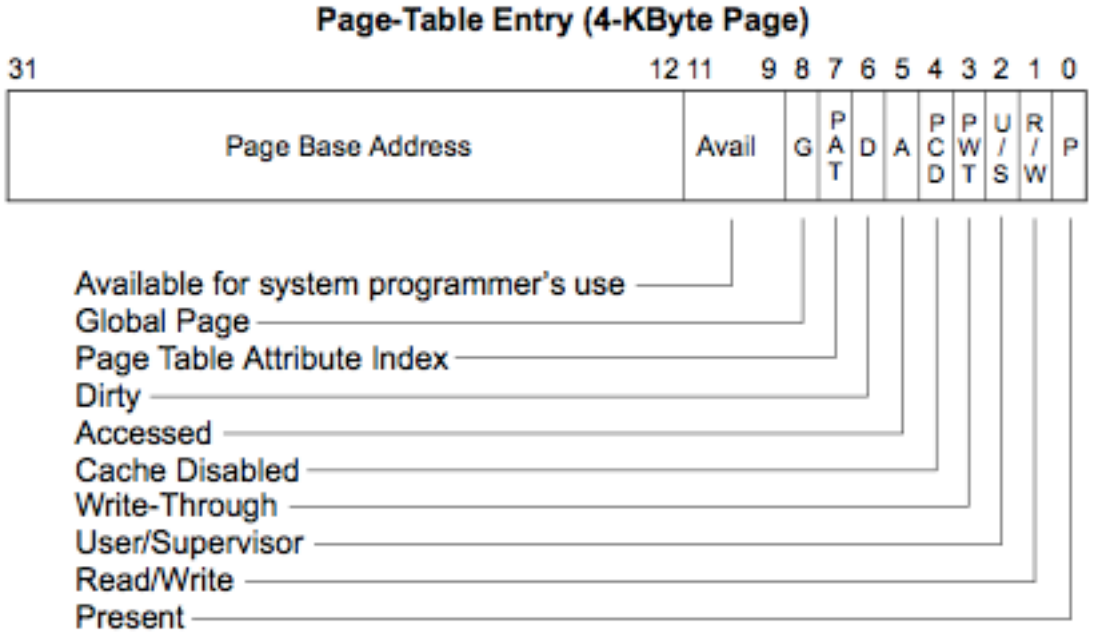
- As we saw before (cue backtrack), the most significant 10 bits of the linear address are used as an index into the Page Directory, to point at a Page Directory Entry (PDE). The PDE in turn specifies the physical address of a Page Table.
- The PDE has a special format we'll talk about in a bit.



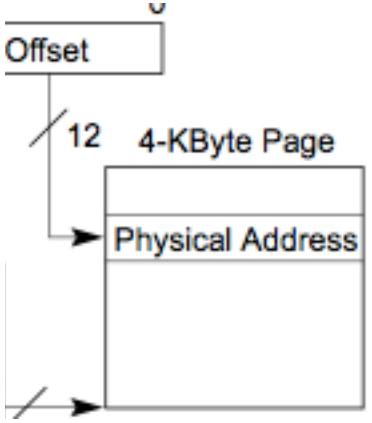
What's going on here?: Page Table Entry



- The middle 10 bits of the linear address are used as an index into the Page Table, to point at a Page Table Entry (PTE). The PTE in turn specifies the physical address of a Page.
- The PTE has a special format (very similar to the PDE) we'll talk about in a bit.



What's going on here?: Page



- Once you have a PTE pointing a the physical address for a page of memory, the last 12 bits of the linear address are used as an offset into that page. It is this location which ultimately represents the physical address where information is stored whenever the virtual address is utilized.
- There's nothing particularly special about any given page. They're just chunks of RAM (4KB big in this case) being organized by the OS. Note that the OS needs to keep track of which physical pages are currently being used.
- That's where "The Other Virtual Memory" comes into play. If the system runs out of physical memory, it can start using storage on the hard drive as fake RAM aka virtual memory.
  - Accessing the HD is about 100000 times slower than accessing RAM

32bit to 32 bit, 4KB pages

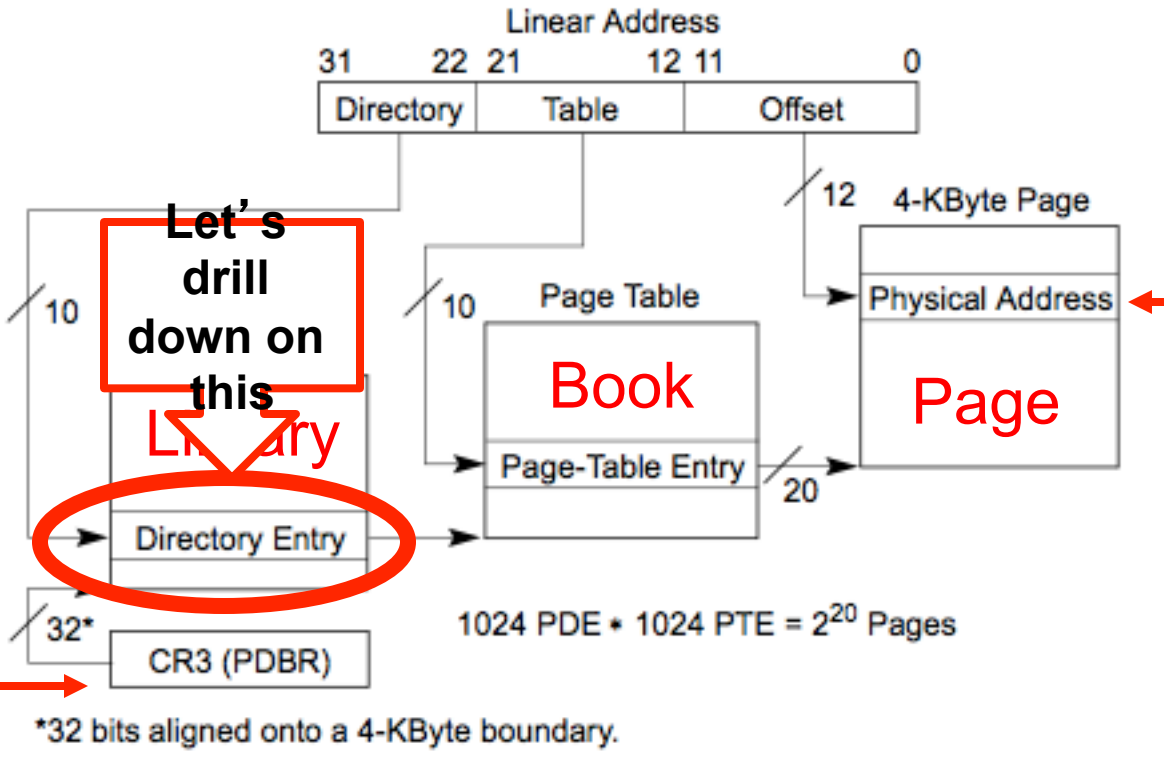
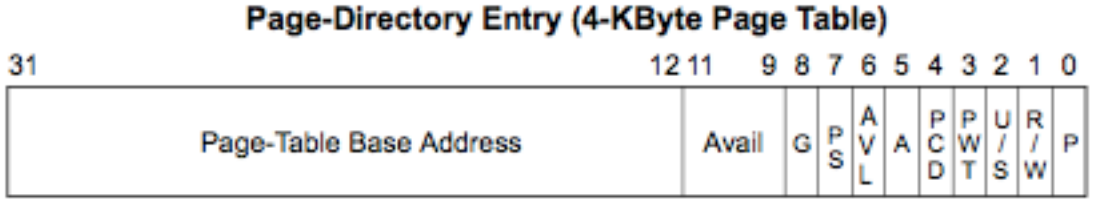


Figure 3-12. Linear Address Translation (4-KByte Pages)

The nerd who always knows where the library is. e.g. me ;)

- This is the process by which linear addresses are translated to physical addresses.
- The OS sets the tables up, and the hardware automatically traverses them whenever you try to access a linear (virtual) address.

Page Directory Entry (PDE) Fields



- **P (Present) Flag** - If set to 1 the pointed to page table is present in physical memory. If 0, either the linear address space is empty (the default starting state) or the memory has been paged to file. If 0, bits 1-31 are available for software-defined purposes.
- **R/W (Read/Write) Flag** - If 1, the pages pointed to by the page directory are read/write. If 0, read-only. (This flag overrides the equivalent flag in the PTE).s
- **U/S (User/Supervisor) Flag** - If 1, pages can be accessed by CPL == 3 (User mode), if 0, pages can only be access by CPL < 3 (Supervisor mode).
- U/S and R/W flags interact with the CR0.WP (Write Protect) Flag in the following way: If CR0.WP == 1, even supervisor mode code cannot write to a read-only page. If CR0.WP == 0, supervisor code can write wherever it wants.

Bold are the more important fields

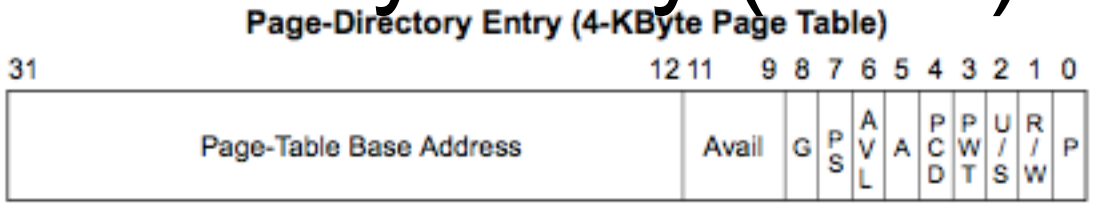
Table 4-3. Combined Page-Directory and Page-Table Protection

Page-Directory Entry		Page-Table Entry		Combined Effect	
Privilege	**Access Type**	**Privilege**	**Access Type**	**Privilege**	**Access Type**
User	Read-Only	User	Read-Only	User	Read-Only
User	Read-Only	User	Read-Write	User	Read-Only
User	Read-Write	User	Read-Only	User	Read-Only
User	Read-Write	User	Read-Write	User	Read/Write
User	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	User	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write

NOTE:

* If CR0.WP = 1, access type is determined by the R/W flags of the page-directory and page-table entries. IF CR0.WP = 0, supervisor privilege permits read-write access.

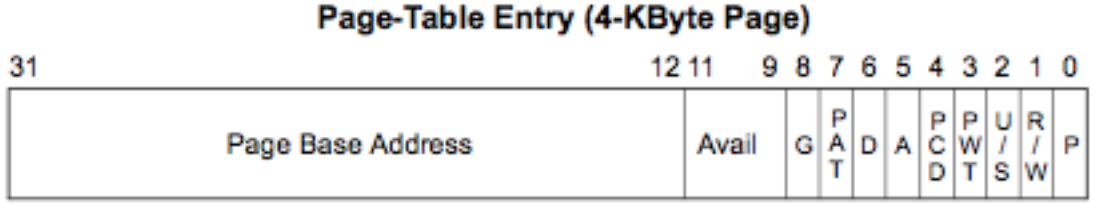
Page Directory Entry (PDE) Fields 2



- Page-level write-through (PWT) flag & Page-level cache disable (PCD) flag - Control caching aspects, but we're not getting into data caching in this class.
- A (Accessed) Flag - If 1, the page table pointed to has been accessed (read/written to). Generally initialized to 0 by memory management software (i.e. not automatically set by hardware on access.)
- AVL & Avail - Bits available for memory management software use.
- **PS (Page Size) flag** - Set to 0 for 4KB pages (therefore PDE points to PTE). Set to 1 for large page (4 MB), and the PDE points directly at a large page. For the case we're covering right now (32bit-32bit-4KB pages) it MUST be set to 0.
- **G (Global) Flag** - Translations between a linear and physical address is cached in the Translation Lookaside Buffer (TLB - talked about later). If set to 1, this cached translation will not be flushed if CR3 is changed to point at a different Page Directory. If set to 0, this translation will be flushed from the TLB as normal, when CR3 changes. We'll revisit this later.
- **Page-Table Base Address** - Upper 20 bits of the ***physical address*** for the base of a Page Table. The bottom 12 bits of the physical address are assumed to be 0s. (I.e. the page table must start on a 12-bit (4KB) aligned address).

Bold are the more important fields

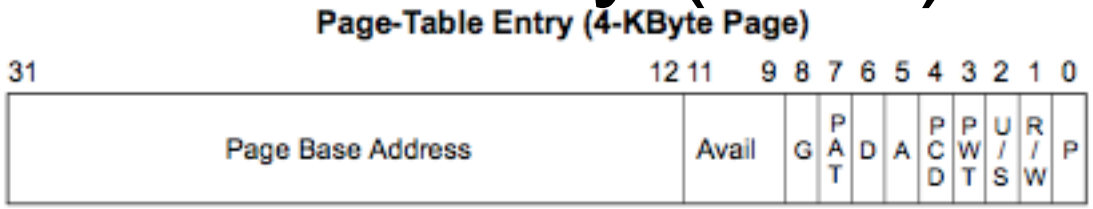
Page Table Entry (PTE) Fields



- **P (Present) Flag** - If set to 1 the pointed to page is present in physical memory. If 0, either the linear address space is empty (the default starting state) or the memory has been paged to file. If 0, bits 1-31 are available for software-defined purposes.
- **R/W (Read/Write) Flag** - If 1, the pages pointed to are read/write. If 0, read-only. (Still subject to the Page Directory permissions, per table 4-3.)
- **U/S (User/Supervisor) Flag** - If 1, pages can be accessed by CPL == 3 (User mode), if 0, pages can only be access by CPL < 3 (Supervisor mode).
- U/S and R/W flags interact with the CR0.WP (Write Protect) Flag in the following way: If CR0.WP == 1, even supervisor mode code cannot write to a read-only page. If CR0.WP == 0, supervisor code can write wherever it wants.
- All these flags are used exactly the same way as in the PDE!

Bold are the more important fields

Page Table Entry (PTE) Fields 2



- Page-level write-through (PWT) flag & Page-level cache disable (PCD) flag - Control caching aspects, but we're not getting into caching in this class.
- A (Accessed) Flag - If 1, the page pointed to has been accessed (read/written to). Generally initialized to 0 by memory management software (i.e. not automatically set by hardware on access.)
  D (Dirty) Flag - If 1, the page pointed to has been written to. Generally initialized to 0 by memory management software (i.e. not automatically set by hardware on access.)
- PAT (Page Attribute Table) Flag - Extends PWT and PCD capabilities, and thus we ignore it in this class.
- Avail - Bits available for memory management software use.
- **G (Global) Flag** - Translations between a linear and physical address is cached in the Translation Lookaside Buffer (TLB - talked about later). If set to 1, this cached translation will not be flushed if CR3 is changed to point at a different Page Directory. If set to 0, this translation will be flushed from the TLB as normal, when CR3 changes. We'll revisit this later.
- **Page Base Address** - Upper 20 bits of the ***physical address*** for the base of a Page. The bottom 12 bits of the physical address are assumed to be 0s. (I.e. the Page must start on a 12-bit (4KB) aligned address).

Lab: WinDbg & !pte command

- !pte takes a VA and prints out the page directory and page table information
- To start with the simple case, we want to force our system to boot in non PAE mode (PAE is talked about later). So reboot the system and select the “NO PAE + DEBUG” entry at the bootloader screen
- We will talk about what we’re going to achieve while it’s rebooting

Getting concrete (overshoes)

- There is no fixed way that virtual memory has to be mapped to physical memory. Different OSes use different conventions for how they organize their virtual address space, some have the kernel at low addresses, some have it at high. Some map kernel memory direct to physical memory, some don't. In all cases, certain areas often are used for certain things simply by having all the code follow a specific convention.

Windows PDE/PTE organization

- By convention, the Page Directory (physical memory pointed to by CR3) is mapped to 0xC0300000. To find the PDE for a given Virtual Address (VA), you compute:
  - 0xC0300000 + (upper 10 bits of VA) * sizeof(PDE)
  - sizeof(PDE) = 4 bytes
- By convention, the Page Tables will be mapped into memory starting at 0xC0000000. To find the PTE for a given VA, you compute:
  - 0xC0000000 + (upper 10 bits of VA) * PAGE_SIZE + (middle 10 bits of VA) * sizeof(PTE)
  - PAGE_SIZE = 0x1000 (4096), sizeof(PTE) = 4
- http://msdn.microsoft.com/en-us/library/cc267483.aspx

Example by hand:

- GDT Base = 0x8003f000
- Binary = 1000 0000 0000 0011 1111 0000 0000 0000
- Grouping = (1000 0000 00)(00 0011 1111) (0000 0000 0000)
- Upper 10 bits = 0x200, Middle 10 bits = 0x3F, Bottom 12 = 0
- PDE = 0xC0300000 + 0x200 * 4 = 0xC0300800
  - If you read the 4 bytes at 0xC0300800 you will get the PDE which will have all the bits which were previously specified in those big tables
- PTE = 0xC0000000 + 0x200 * 0x1000 + 0x3F * 4 = 0xC02000FC
  - If you read the 4 bytes at 0xC02000FC you will get the PTE which will have all the bits which were previously specified in those big tables

How do you implement a convention?

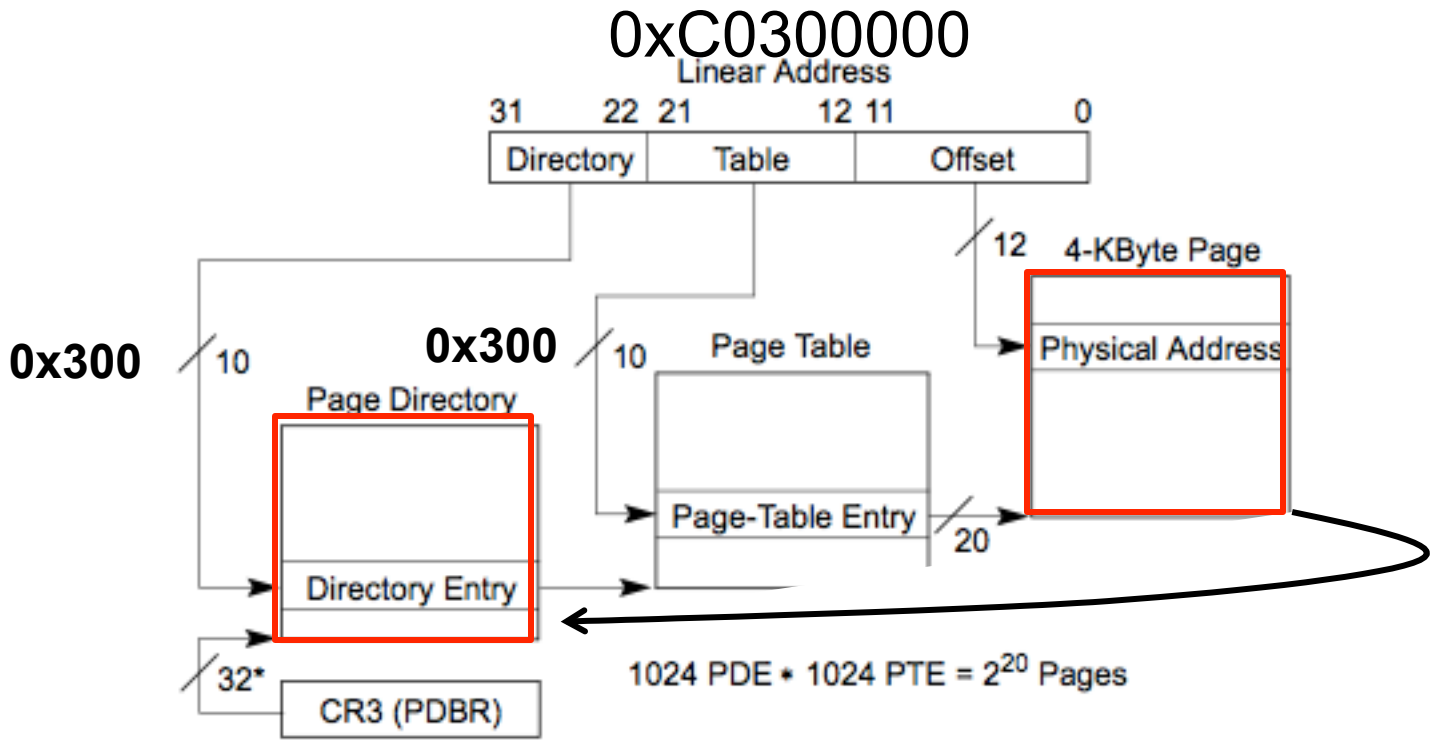


Figure 3-12. Linear Address Translation (4-KByte Pages)

Not presently here

- Both the PDE and PTE have the 0th bit as the Present Bit.
- What happens a program attempts to access memory, and during translation between the linear and physical memory the hardware comes across a PDE or PTE with the present bit set to 0?
- It invokes a Page Fault #PF (IDT[14]). The OS-supplied page fault handler then determines whether it can recover from the fault.
- When a Page Fault occurs, the address which was being attempted to be accessed is automatically put into the CR2 register.
- Some causes of page faults
  - Page is paged out ("swapped") to disk (recoverable)
    - FYI, page file organization is OS-specific
  - Automatic stack growth (recoverable)
  - Attempts to write to read-only memory (recoverable if memory is intended to be copy-on-write)
  - No valid linear-to-physical translation (unrecoverable)
  - User code accessing memory marked as supervisor (unrecoverable*) (* = see slide notes)

32 bit to 32 bit 4MB pages

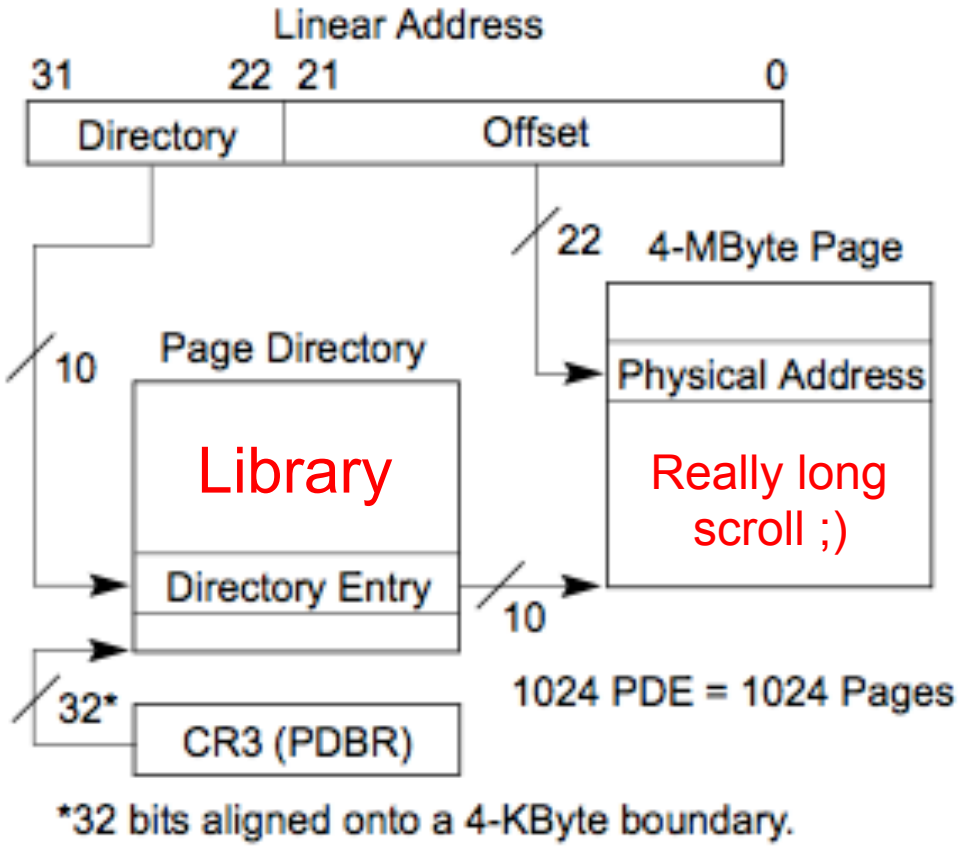


Figure 3-13. Linear Address Translation (4-MByte Pages)

32-32-4MB PDE

- Fields except for page base address interpreted the same way. (Since you can see the D and PAT bits, you can see that it's kind of combining fields from both the PDE and PTE)
- Most significant 10 bits specify 4MB-aligned page start (other bits assumed to be 0)
- **Page Size (PS) Flag** - MUST be set to 1 to have 4MB page, and for the 1 to mean anything, the CR4 PSE (Page Size Extensions) flag must be set to 1.

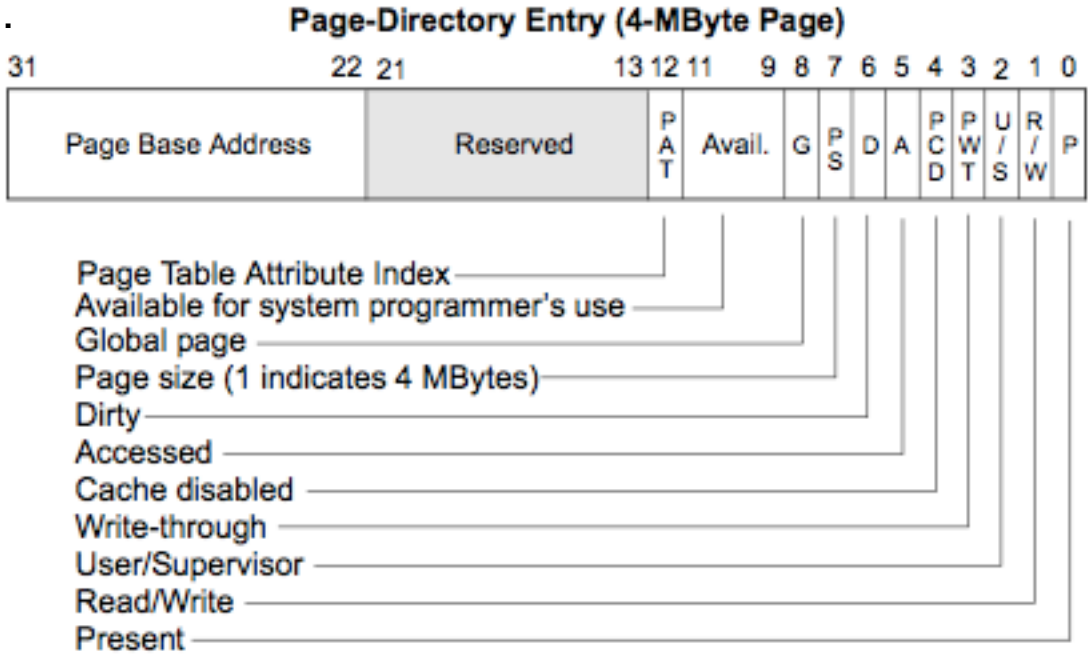


Figure 3-15. Format of Page-Directory Entries for 4-MByte Pages and 32-Bit Addresses

Why would I want ginormous pages?

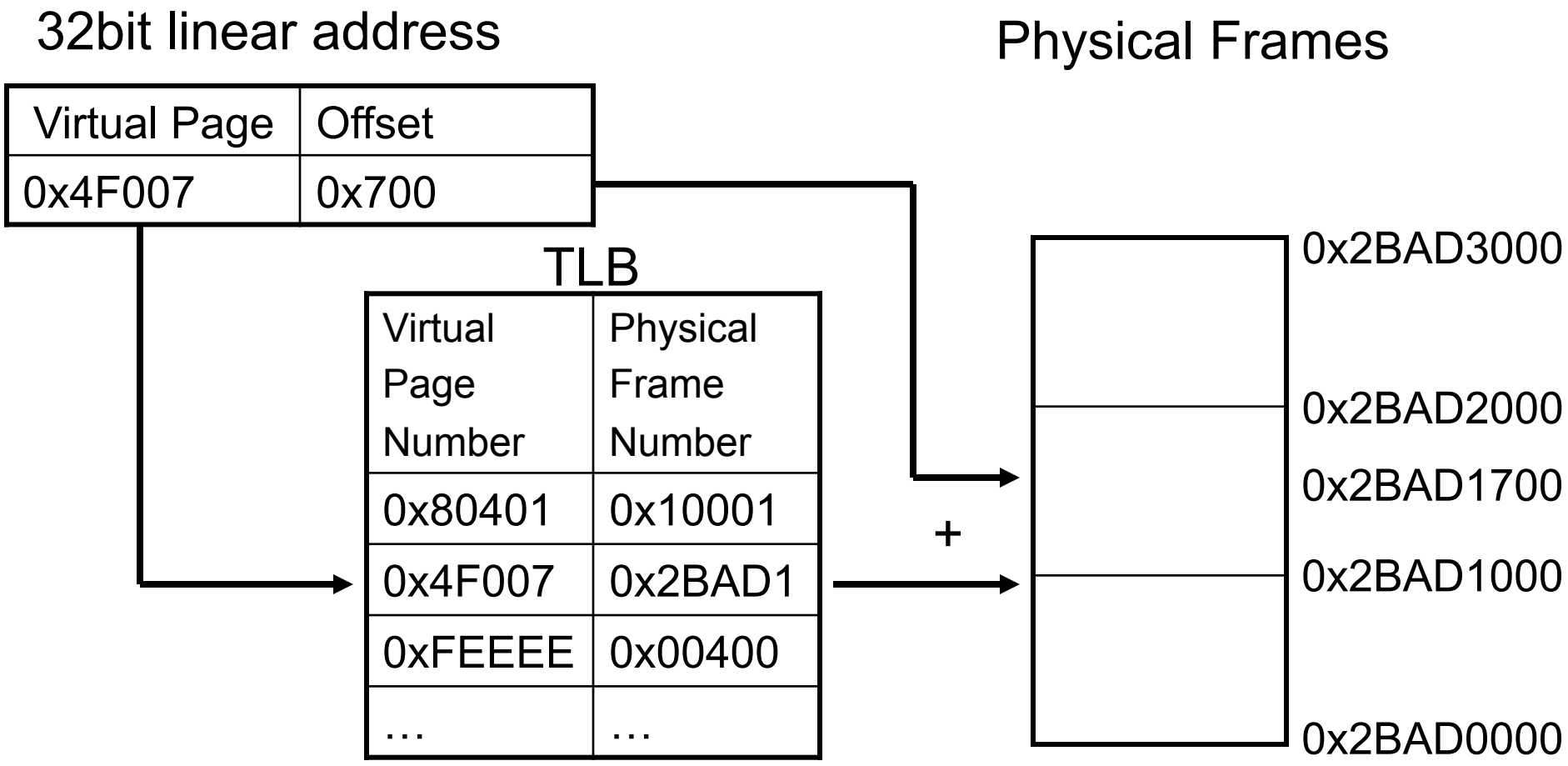
- From the Nov 2008 edition of Intel Vol 3a, section 3.7.3:
- "When the PSE flag in CR4 is set, both 4-MByte pages and page tables for 4-KByte pages can be accessed from the same page directory. If the PSE flag is clear, only page tables for 4-KByte pages can be accessed (regardless of the setting of the PS flag in a page-directory entry)."
- "A typical example of mixing 4-KByte and 4-MByte pages is to place the operating system or executive's kernel in a large page to reduce TLB misses and thus improve overall system performance."
- "The processor maintains 4-MByte page entries and 4-KByte page entries in separate TLBs. So, placing often used code such as the kernel in a large page, frees up 4-KByte-page TLB entries for application programs and tasks."
- OK, I can't defer it any longer, let's look at the TLB.

Translation Lookaside Buffer (TLB)

- An in-CPU cache which stores translations between linear addresses and physical pages.
- The idea being that memory accesses will be faster when the hardware does not have to walk from CR3 to the page directory and possibly to the page table.
- By caching a map which describes which linear page corresponds to which physical page, the hardware can just use the least significant x bits which specifies the offset into the page.
  - (where x depends on what type of paging layout you're using - in canonical 4KB pages x = 12, in 4MB pages x = 22)
- This is similar to the way that the "hidden" part of a segment register stores the information from a segment descriptor, so that it doesn't have to go back and walk the GDT. Except it's more complicated in that the TLB is a real cache and follows layout like other types of caches. (4-way set associative if you know what that means, if not, it's quite alright.)

TLB as a grey box

(not actually how it looks, just to give you something to visualize)



Result: Data stored at virtual memory address 0x4F007700 is stored at physical memory address 0x2BAD1700 without consulting the page tables

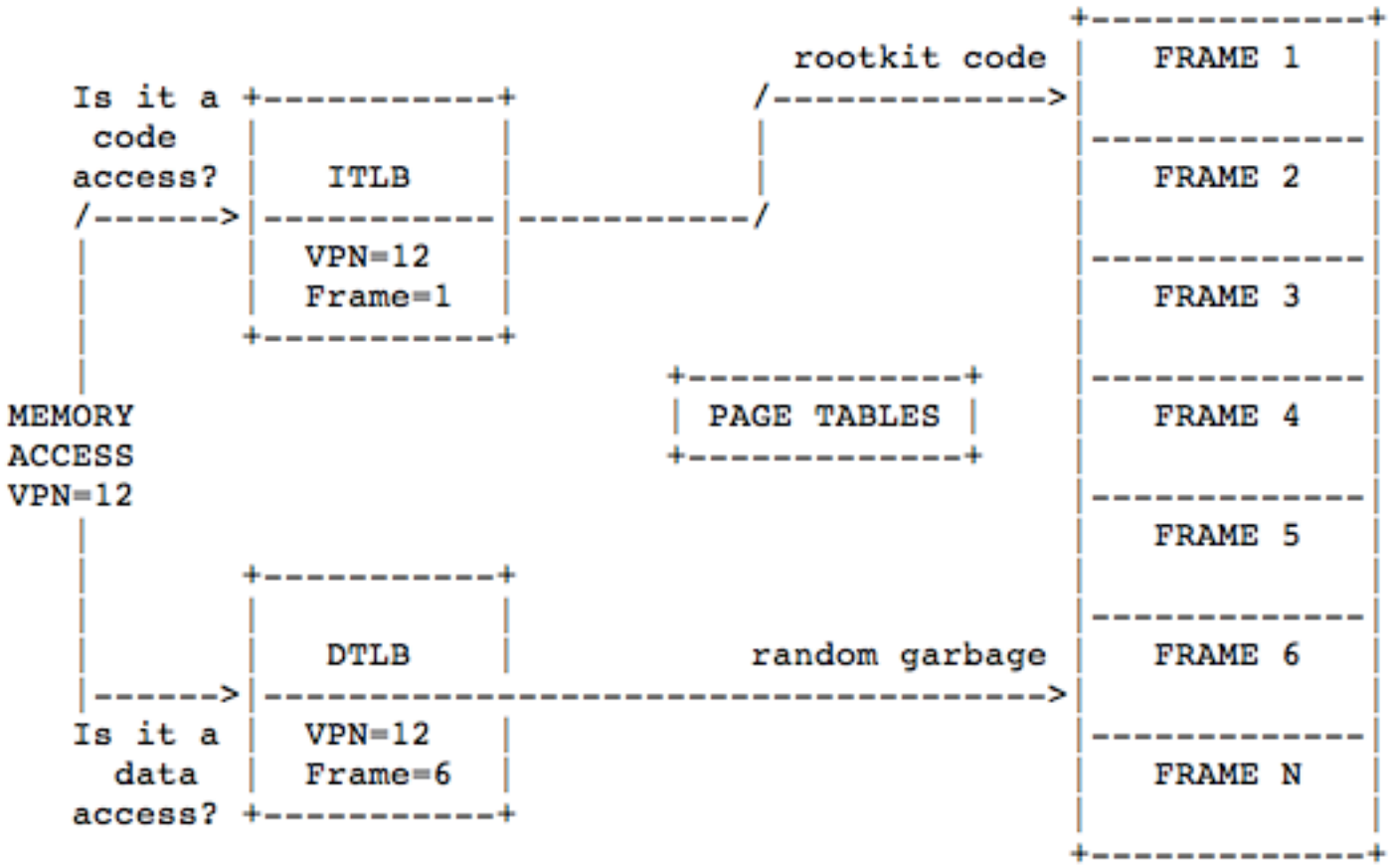
More about the TLB

- Whenever CR3 is set to a new value, all TLB entries which are not marked as global are flushed. (Only ring 0 can mov a value to CR3.)
- Ring 0 code can also use the INVLPG instruction to invalidate the TLB cache entry for a specified virtual memory address.
- There are actually multiple TLBs. Generally there are four:
  - Data TLBs: one for caching 4KB translations, one for large (2/4MB) pages
  - Instruction TLBs: one for 4KB, one for large (2/4MB) pages
- Number of entries in the cache differs between chip microarchitectures and revisions. Example for Core 2 Duo (T7400 - Merom) in my MacBook Pro:
  - DTLBs - 256 4KB entries, 32 4MB entries
  - ITLBs - 128 4KB entries, 8 4MB entries

Shadow Walker Rootkit

- https://media.defcon.org/dc-13/video/2005_Defcon_V81-Sherri_Sparks,_Jamie_Butler-Shadow_Walker.m4v
- http://www.phrack.com/issues.html?issue=63&id=8
- TLB manipulation for fun and profit – hides a page of memory from a memory scanner
- Exploits the fact that there can be a different translation for the same virtual memory address stored in the ITLB vs the DTLB
- In order to allow itself to execute normally but fool a scanner, it needs to differentiate execute accesses (caused by itself), from read/write accesses (caused by a scanner). In order to do this, it sets the present bit to 0 for the page it is executing on, and then replaces the page fault handler(PFH).
- The modified PFH then examines the EIP which was pushed with the fault state. If the EIP is within the hidden page, then it is execution by the rootkit, and the page fault handler should map the virtual memory to the correct frame (thus filling the ITLB). If the EIP is anywhere else, then it is some outside entity and the page should be mapped to any random frame (thus filling the DTLB.)

ASCII Art of Dooooom!



[Figure 5 - Faking Read / Writes by Desynchronizing the Split TLB]

Defeating Shadow Walker

- Check the IDT page fault handler entry to ensure it has not been pointed at a different location than the expected OS page fault handler.
- Integrity check the memory for the legitimate page fault handler to ensure it isn't subject to an inline hook.
- Manually flush the TLB before scanning memory.
- Map each physical page of memory to a target virtual memory page and read the page looking for whatever you were looking for in the first place.
- Profile paging performance.

Physical Address Extention (PAE)

- A hardware-supported way to address more than 4GB of RAM by just rewriting your memory manager (as opposed to a full system rewrite for 64bit support)
- Windows uses a different kernel when PAE is being used - ntkrnlpa.exe (as opposed to ntoskrnl.exe)
  - If you have enabled DEP (Data Execution Prevention) on a Windows system, it requires PAE, and thus it will load this alternate kernel.
  - DEP is enabled (for system services) by default post XPSP2, therefore your OS is probably running in PAE mode most of the time.

PAE - 32 bit to 36 bit, 4KB pages

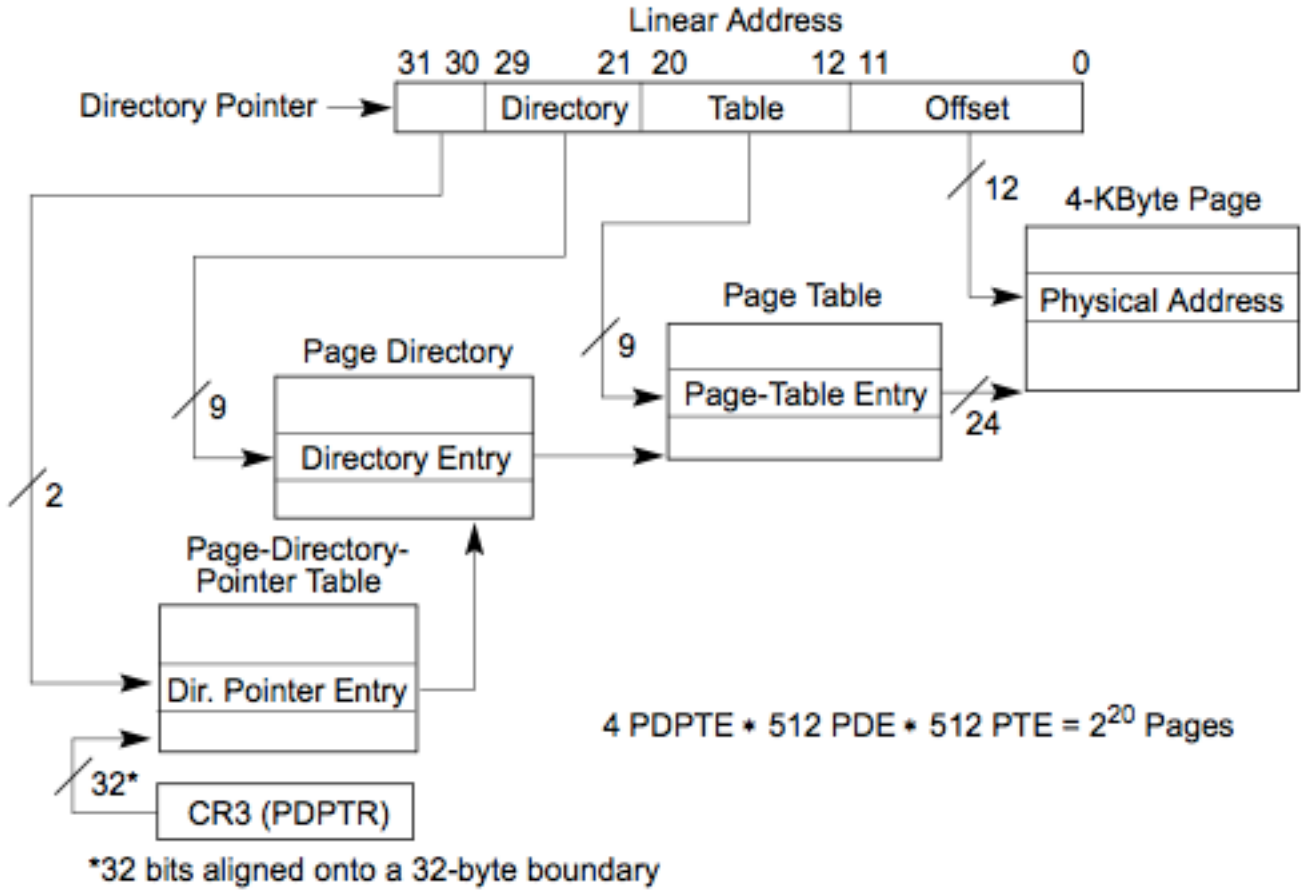


Figure 3-18. Linear Address Translation With PAE Enabled (4-KByte Pages)

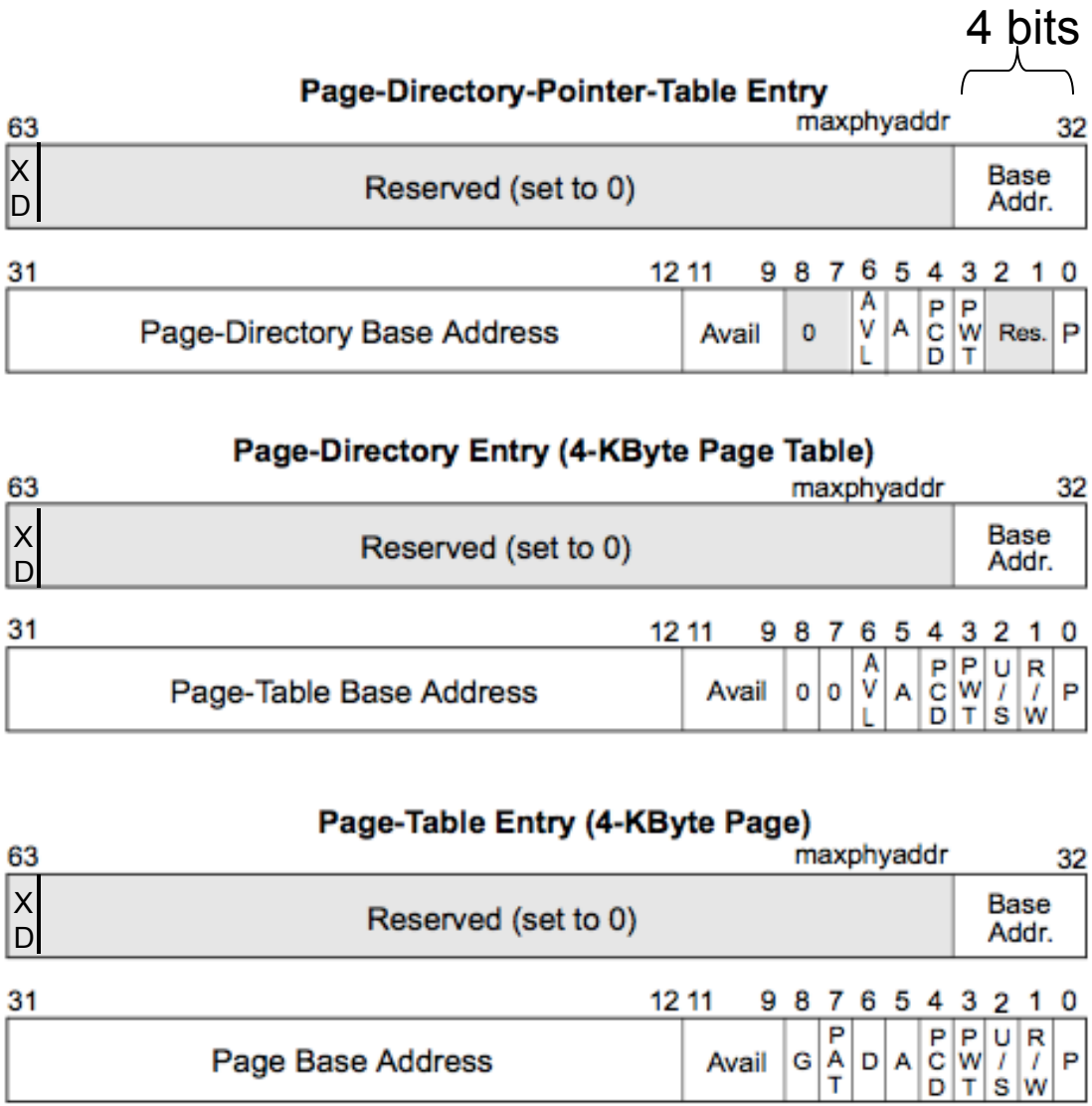
Do the math

- Uhhh…Doesn't look like traversing through those tables lets a single process access 64GB of memory…
- That's right! You didn't really think there was going to be a one-to-one mapping from a small set (32 bit linear addresses) to a big set (36 bit physical addresses) did you? (I did before I paid attention ;))
- It doesn't really help an individual process that much, it just helps out the system as a whole, because the memory manager has a larger list of free frames that it can pull from, so that it doesn't have to page stuff out as often due to lack of spare space.

Do the math 2

- Then what was the point of changing the page table layout?
- 1. If the OS wanted to allow processes to access more than 4GB of memory, it could support swapping out entries in the Page Directory Pointer Table
  - This would avoid changing CR3 and the consequent invalidating of TLB entries
- 2. There weren't 4 extra bits available in the existing PDEs and PTEs.
- 3. Room to support new features (like NX/XD talked about later.)

PAE Entry Formats



NOTE!
Bit 63 is the XD bit, it's just not shown on these old pictures (and like I said I dislike the new format)

Figure 3-20. Format of Page-Directory-Pointer-Table, Page-Directory, and Page-Table Entries for 4-KByte Pages with PAE Enabled

Windows PAE PDE/PTE organization

- By convention, the Page Directory (physical memory pointed to by CR3) is mapped to 0xC0600000. To find the PDE for a given Virtual Address (VA), you compute:
  - 0xC0600000 + (upper 11 bits of VA) * sizeof(PDE)
  - sizeof(PDE) = 8 bytes
  - NOTE: This means Windows is not even using the Page Directory Pointer Table!
- By convention, the Page Tables will be mapped into memory starting at 0xC0000000. To find the PTE for a given VA, you compute:
  - 0xC0000000 + (upper 11 bits of VA) * PAGE_SIZE + (middle 9 bits of VA) * sizeof(PTE)$_{42}$
  - PAGE_SIZE = 0x1000 (4096), sizeof(PTE) = 8

Example by hand:

- GDT Base = 0x8003f000
- Binary = 1000 0000 0000 0011 1111 0000 0000 0000
- Grouping = (1000 0000 000)(0 0011 1111) (0000 0000 0000)
- Upper 11 bits = 0x400, Middle 9 bits = 0x3F, Bottom 12 = 0
- PDE = 0xC0600000 + 0x400 * 8 = 0xC0602000
  - If you read the 8 bytes at 0xC0601000 you will get the PDE which will have all the bits which were previously specified in those big tables
- PTE = 0xC0000000 + 0x400 * 0x1000 + 0x3F * 8 = 0xC04001F8
  - If you read the 8 bytes at 0xC02000FC you will get the PTE which will have all the bits which were previously specified in those big tables

PAE - 32 bit to 36 bit, 2MB pages

- Left as an exercise to the reader ;)
- (used to map ntkrnlpa.exe pages)

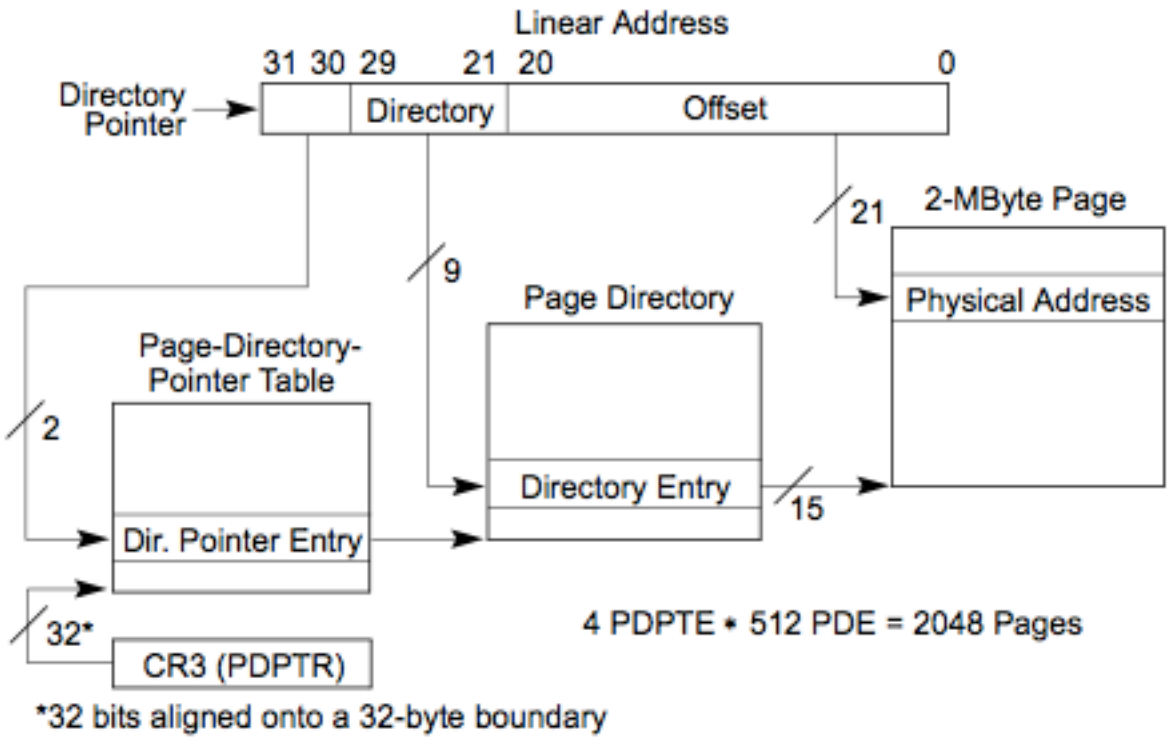


Figure 3-19. Linear Address Translation With PAE Enabled (2-MByte Pages)

NX/XD Bit

- Originally the NX (No Execute) bit was an AMD extension. Intel picked it up as well, as the XD (eXecute Disable) bit. It's more commonly referred to as the NX bit though.
- The bit implements a "W^X" (write XOR execute) policy whereby memory can either be writable or executable but not both.
- Remember that segmentation has a way of preventing memory from being executable, but everyone uses page level protections rather than segmentation, so this became necessary to prevent code from executing in ostensibly-data-only memory regions such as stack or heap.
- Intended for preventing some exploits.

NX/XD Bit 2

- Any pages which have a PTE, PDE, or PDPTE with the XD = 1, are non-executable (higher granularity specifications of XD override finer granularity).
- Attempts to execute from non-executable pages results in a page fault
- MS refers to the utilization of XD as Data Execution Prevention (DEP) or "Hardware DEP". "Software DEP" refers to MS's Structured Exception Handler sanity checking ("SafeSEH") and has nothing to do with XD.
- It's all well and good until someone uses a different type of exploit to turn off DEP ;) http://www.uninformed.org/?v=2&a=4 (But I'll let the exploit class talk about that)

Lab: Investigating NX bits

- I couldn't find any pages in VMWare which had NX active. Fail.
- I suspect VMWare Server 1.x doesn't support exporting the NX bit to the VM
- Here's some "proof" from someone else's site that if XD is enabled, you should see the MSBit set to 1
- kd> !pte 0xFFDF0400
  VA ffdf0400
  PDE at 00000000C0603FF0 PTE at 00000000C07FEF80
  contains 0000000000127063 contains 0000000000152163
  pfn 127 ---DA--KWEV pfn 152 -G-DA—KWEV <- Executable bit is set
  kd> !pte 0x7FFE0400
  VA 7ffe0400
  PDE at 00000000C0601FF8 PTE at 00000000C03FFF00
  contains 000000003D283867 contains **8**000000000152005
  pfn 3d283 ---DA--UWEV pfn 152 -------UR-V <- Executable bit is not set
- From http://www.harmonysecurity.com/blog/2009/11/implementing-win32-kernel-shellcode.html

The ol’ switcharoo

- In a typical protected mode OS, processes have isolated (“protected” ;)) memory spaces, and threads within the same process share the memory space.
- But how are these memory spaces achieved? Clearly not through segmentation.
- When the kernel is context switching between processes it makes sure each gets a CR3 value which points to a different Page Directory (or Page Directory Pointer Table in PAE.)
- And the kernel and all modules which operate in the kernel all have the same view of memory, because they’re all using the same CR3 value.
- This is why if someone can load a kernel module/driver, they then have free reign to scribble over the kernel or other modules. BUT, because the CR3 for userspace processes will not be the same as for the kernel, it actually takes some Legwork for an attacker (or defender) in kernel space to influence the memory of userspace processes.

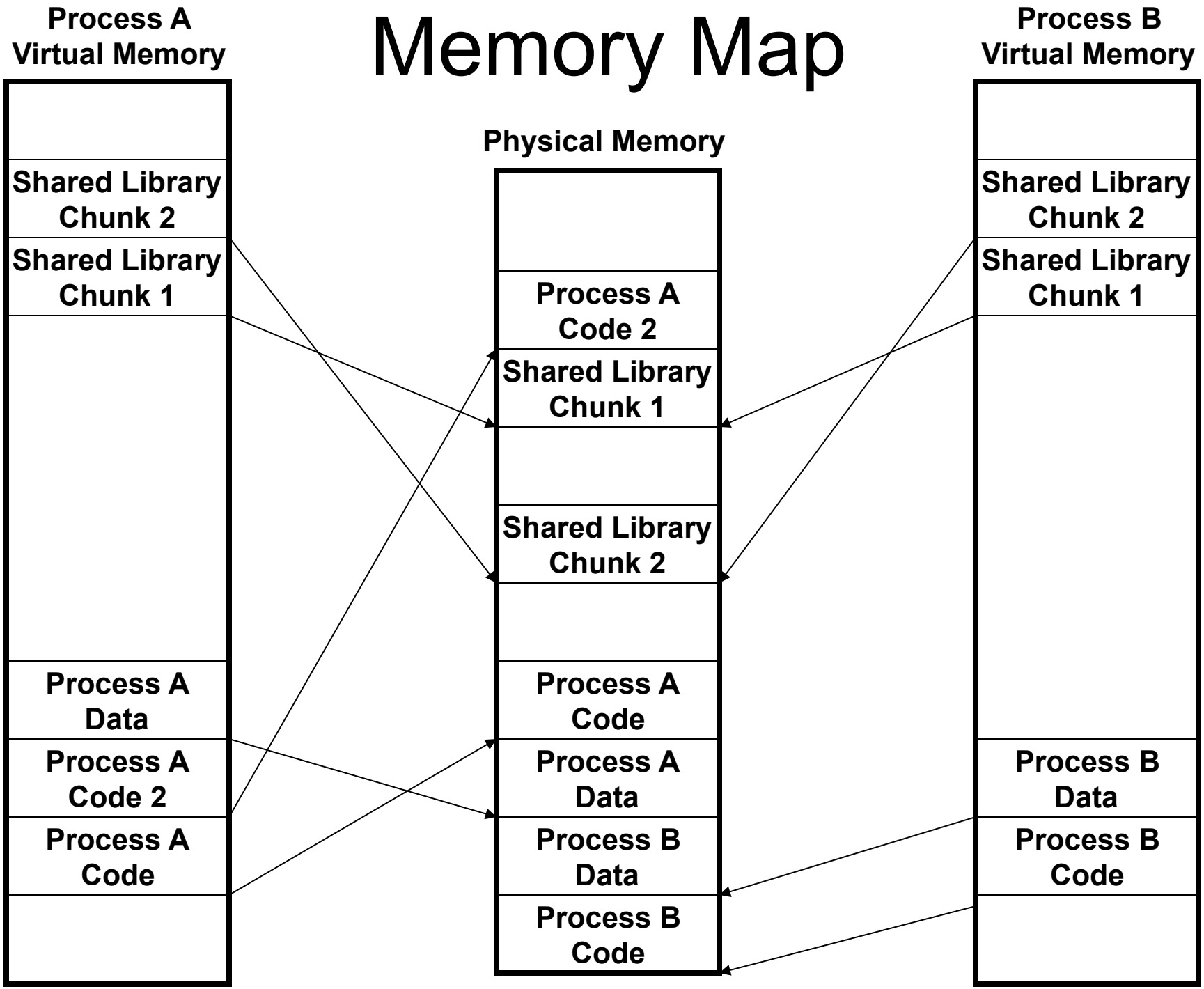
Sharing memory

- What about when you don't want there to be strict isolation between memory spaces?
- Shared Libraries - The whole point of Dynamic Link Libraries (DLLs - Windows) and Shared Objects (*nix), is that you don't want to build a library into each executable which uses it, because then you have multiple copies of the exact same code stored in physical memory, which wasteful. Therefore it is desirable to have the library stand alone and map that single physical memory region corresponding to the library into multiple processes' virtual address spaces.
  - For reasons discussed in a future class it is desirable, *but not required*, to map the library into the same virtual address range in all processes. Not required because you can't prevent the loading of a necessary library just because some other jerk library already took its desired virtual memory location.

Sharing Memory 2

- Inter-Process Communication (IPC) - If two processes need to send large amounts of data between each other, it could be achieved by simply mapping a fixed size chunk of physical memory into their virtual memory spaces, and then having the processes agree on a communications protocol.
  - Remember though that userspace processes can't affect page tables directly, there must be some OS-supported way which they must leverage. E.g. VirtualAlloc()/VirtualProtect() on Windows or mmap()/mprotect() on *nix. (Or some more user-friendly IPC mechanism.)
  - A separate technology, Direct Memory Access - DMA, operates on the same premise that having hardware devices talk to each other through chunks of RAM can be quite fast. "DMA: the Fire in your Wire" - Max Dornseif

Memory Map


Process A
Virtual Memory
Shared Library
Chunk 2
Shared Library
Chunk 1
Process A
Data
Process A
Code 2
Process A
Code
Physical Memory
Process A
Code 2
Shared Library
Chunk 1
Shared Library
Chunk 2
Process A
Code
Process A
Data
Process B
Data
Process B
Code
Process B
Virtual Memory
Shared Library
Chunk 2
Shared Library
Chunk 1
Process B
Data
Process B
Code

Memory Sharing

- Through the eyes of paging structures
- (NOTE: I doctored this image slightly, also, pretend task is the same as process)

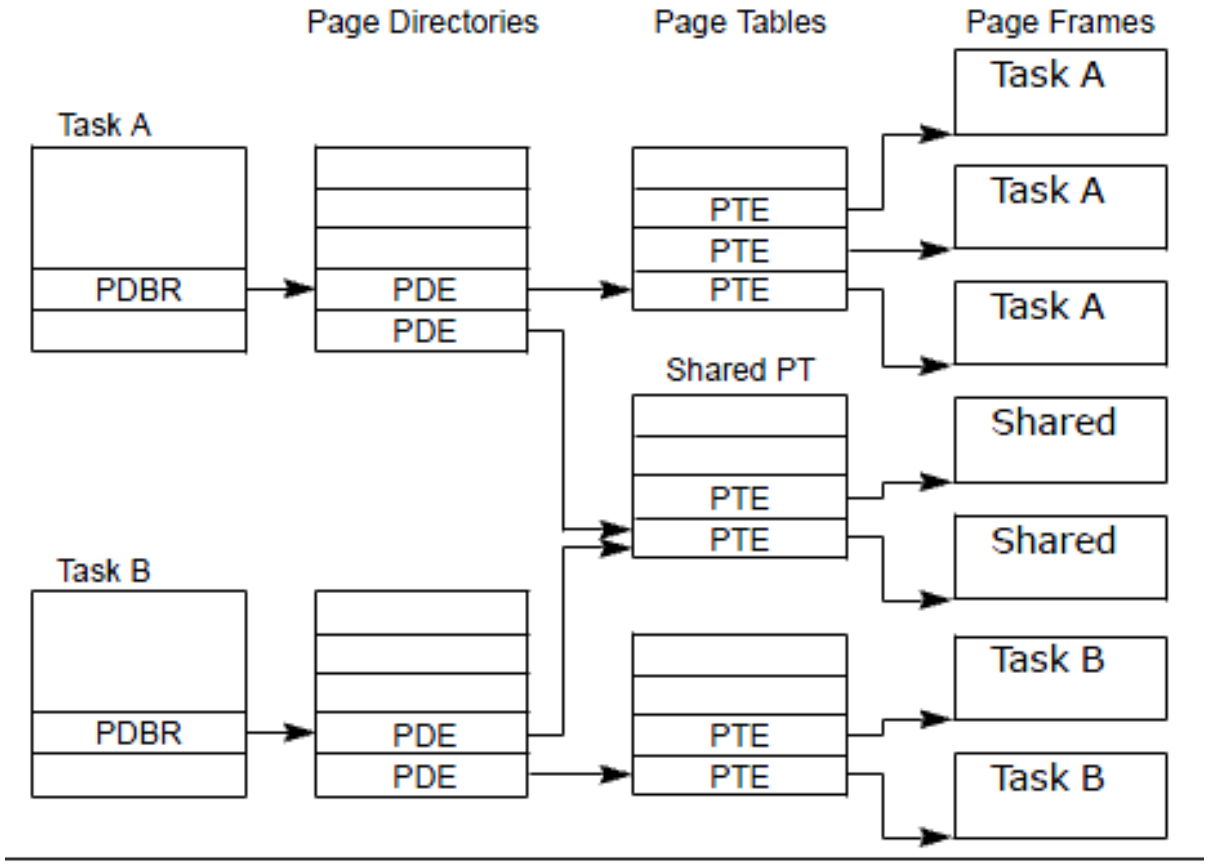


Figure 6-9. Overlapping Linear-to-Physical Mappings

Misc Instructions Picked Up Along The Way

- MOV CR, r32 – (CR = One of the Control Registers)
- MOV r32, CR – (CR = One of the Control Registers)
- INVLPG – Invalidate page entry in TLB