Operating systems

Week 6-1

Memory Management

Lesson Preview

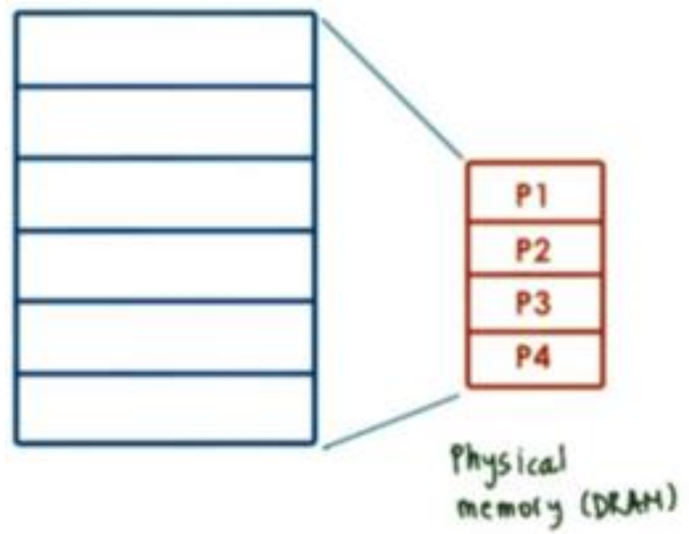
- Memory Management
  - Physical and virtual memory management
  - Review of memory management mechanisms
  - Illustration of advanced services

Memory Management Systems

- Memory management subsystems that are part of operating systems both have some similar types of goals
- MMS use intelligentially sized containers; memory pages or segments
- Not all memory is needed at once in the manner where tasks operate on subset of memory
- Optimized for performance in contrast, reducing the time to access state in memory leading to better performance

Memory management: Goals

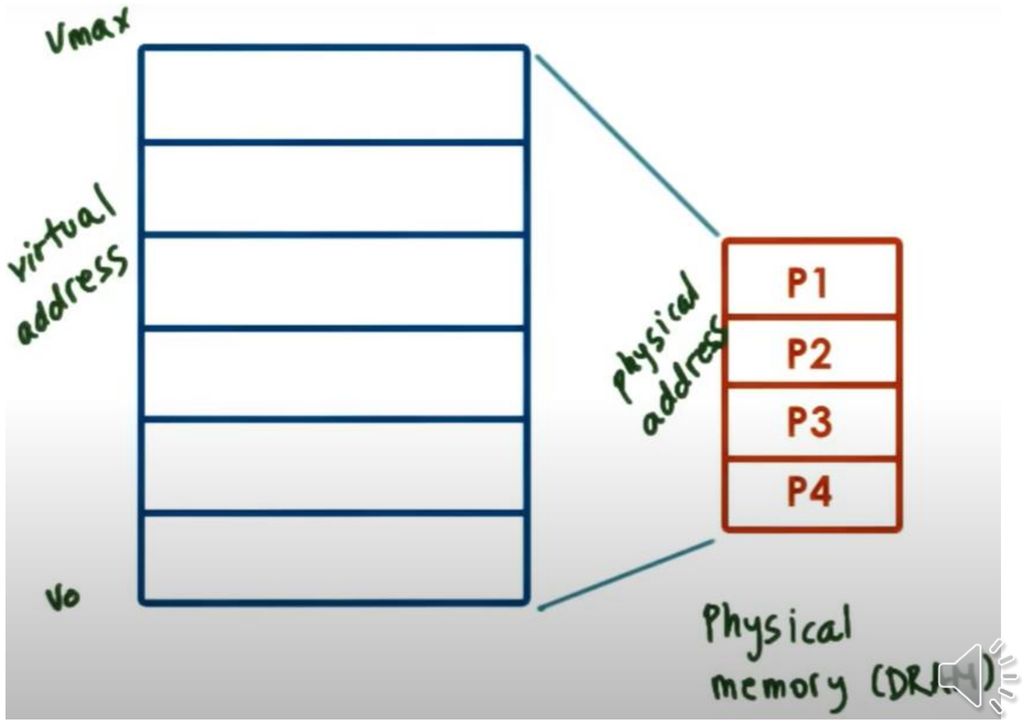
- OS manages physical resources e.g., DRAM
- Decouple the notion of physical memory from the virtual memory that used by the address space in order not to post any limits on the size and the layout of an address space based on the amount of physical memory or how it shared with other processes
- In fact, everything uses virtual addresses, and these are translated to the actual, physical addresses where the state is stored



- The range of the virtual addresses, from V_0 to V_{max}, can be much larger than the actual amount of physical memory
- OS allocate physical memory and also arbitrate how it's being accessed→ mechanisms/algorithms as well data structures
- Note data allocation since p-memory smaller than v-memory

P-memory vs V-memory

- Allocate → allocation, replacement
- Arbitrate→ address translation and validation



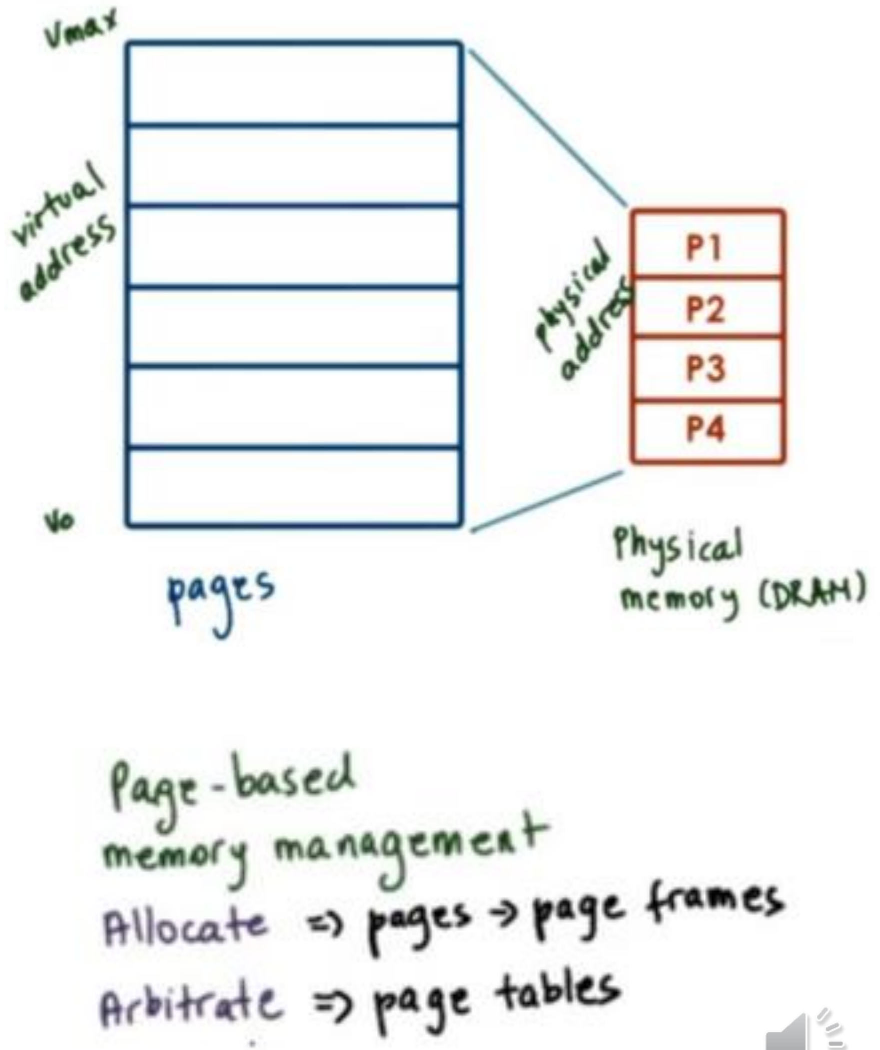
- Allocation → there is some dynamic component to the memory management process that determines when a content should be brought in from disk and then which contents from memory should be stored on disk depending on the kinds of processes that are running

- Arbitration → operating system is able to interpret and verify a process memory access,
  - translate, validate and verify

OS Support for MMS.... Decoupling

- Pages→ fixed size segments divides the v-memory space
- Page frames→ of the same size divides the p-memory

  " not to scale between the two"
- For allocation, the OS is to map pages from the virtual memory into page frames of the physical memory
- Page-based memory management system the arbitration of the access is done via page tables



OS Support for MMS…. Segmentation

- A segment-based memory management approach

- Allocation uses more flexibly-sized segments that can then be mapped to some regions in physical memory as well as swapped in and out of physical memory

- Arbitration of accesses in order to either translate or validate the appropriate access uses segment registers that are typically supported on modern hardware

Hardware Support … Faults

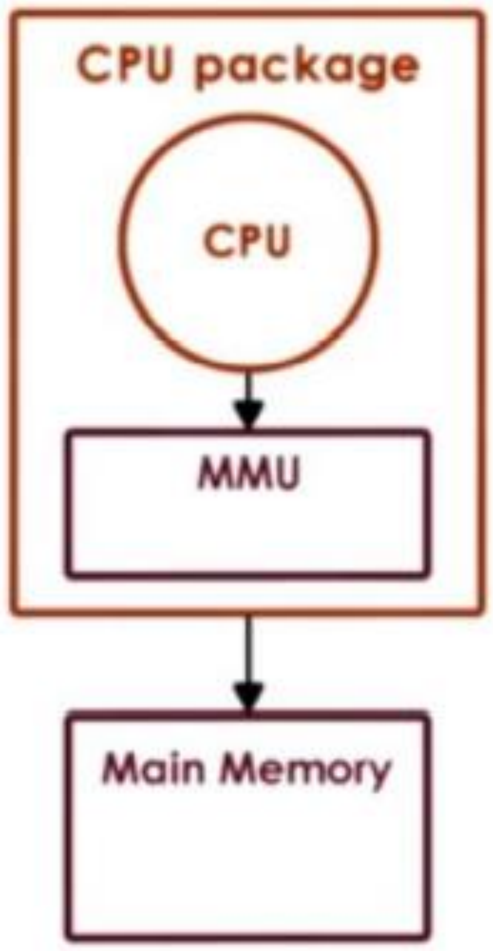
- Every CPU package is equipped with a Memory management unit
- The CPU issues virtual addresses to the Memory Management Unit, and it’s responsible for translating them into the appropriate physical address. Or potentially the MMU can generate a fault
  - Illegal accesses examples:

  - insufficient permissions to perform a particular access

  - override a particular memory location

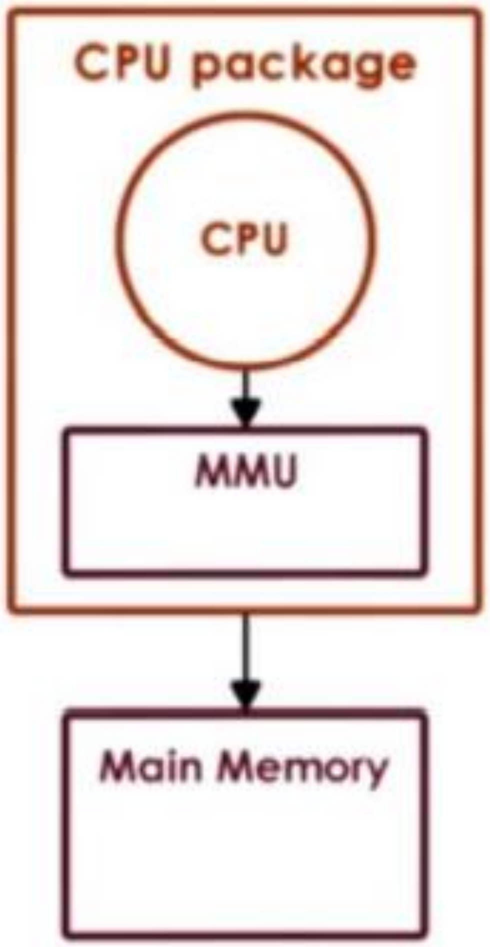
  - the memory address that’s requested hasn’t been allocated at all

  - page that’s being referenced isn’t present in memory and must be fetched from disk



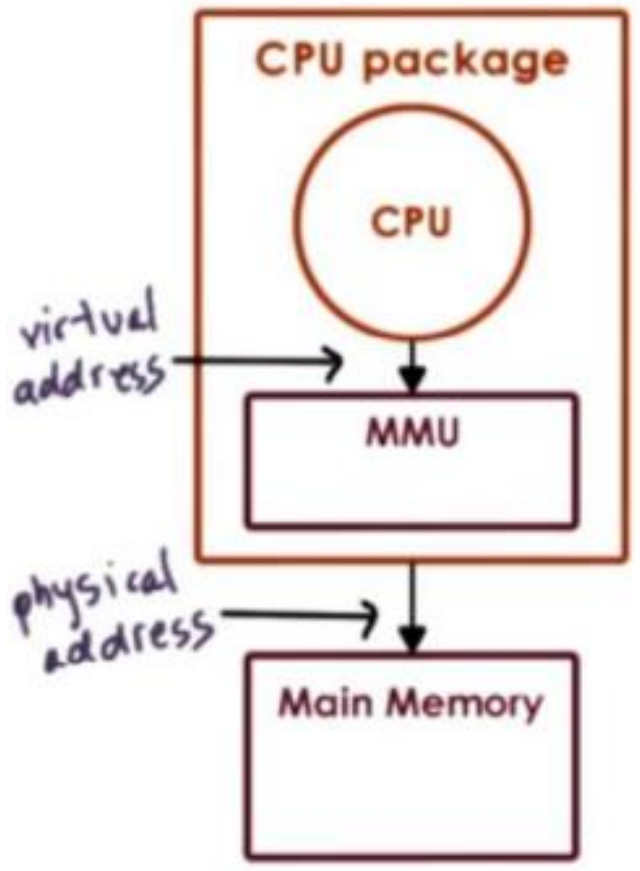
Hardware Support ... Registers

- Another way hardware supports memory management is by using designated registers during the address translation process
  - Page-based→ registers are used to point to the currently active page table
  - Segment-based→ registers are used to indicate the base address of the segment, its limits or over all size and total number of segments ...etc.



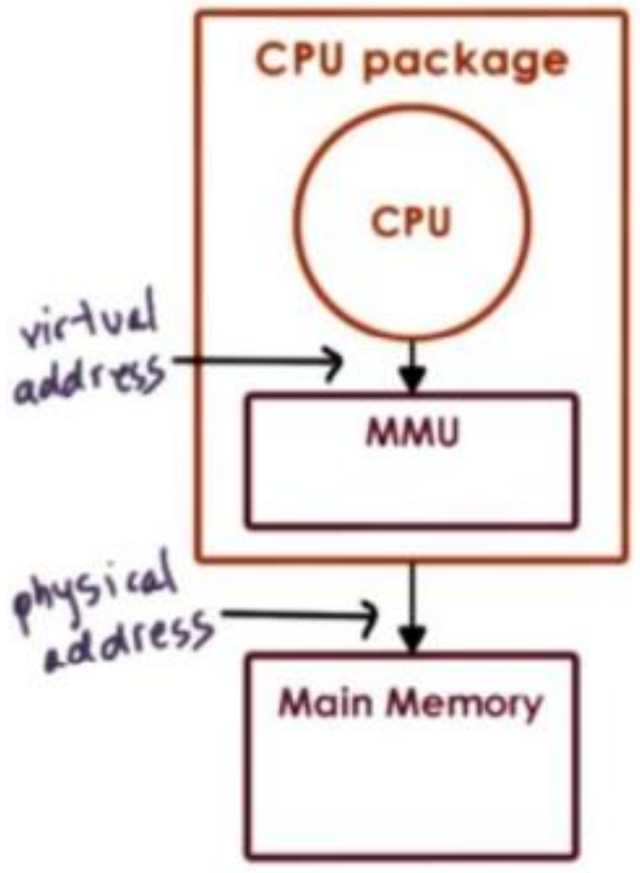
Hardware Support ... Cache

- Since the memory address translation happens on pretty much every reference, most memory management units would integrate a small cache of valid virtual to physical address translations
- This is called the translation lookaside buffer or TLB
- The presence of a TLB will make the entire translation process much faster



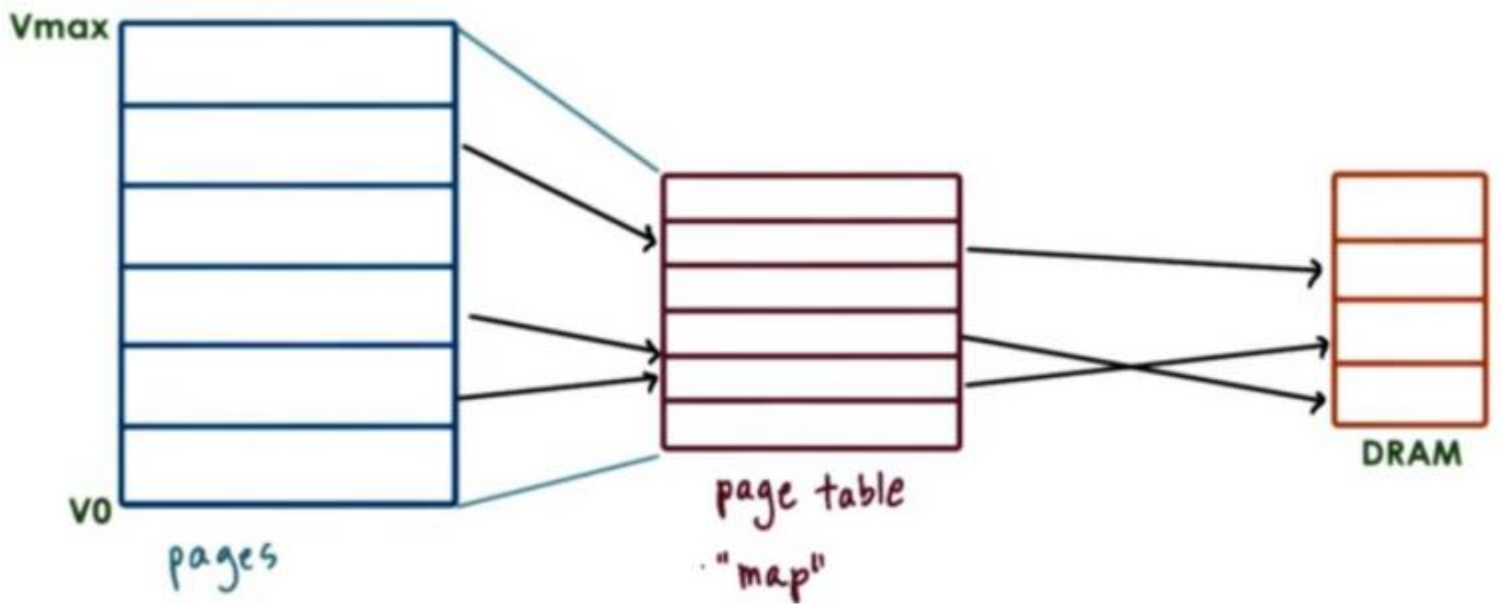
Hardware Support … Translation

- And finally, the actual generation of the physical address from the virtual address, so the translation process, that's done by the hardware
- The OS will maintain certain data structures such as the page table to maintain certain information that's necessary for the translation process however, the actual translation the hardware performs it.
- This also implies that the hardware will dictate what type of memory management modes are supported



Page Tables

- Page tables→ translates the virtual memory addresses into physical memory addresses



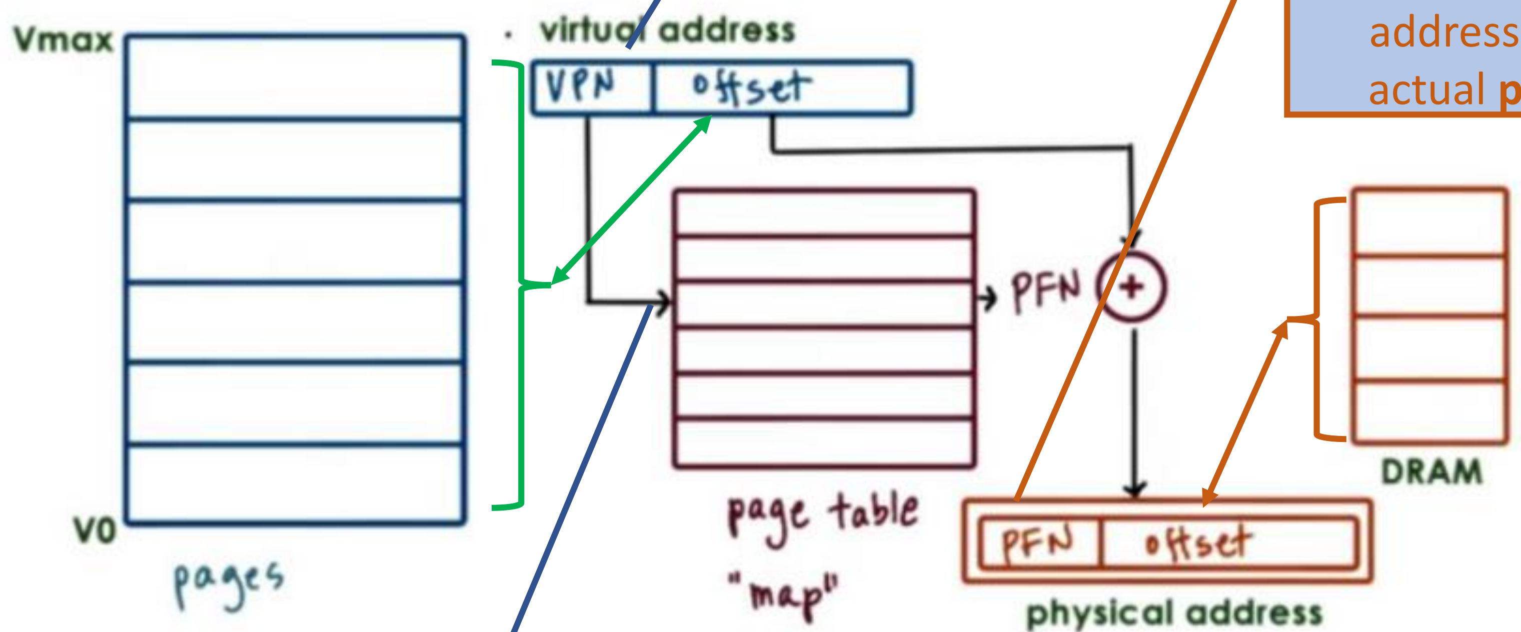
- A map that tells the operating system and the hardware itself where to find specific virtual memory references
- The sizes of the pages of the v-memory, and the corresponding page frames in p-memory are identical, keeping those two same size then we don't have to keep track of the translation of every single individual virtual address

Virtual to Physical Memory Addresses

- Why we don’t have to keep track of the translation of every single individual virtual address based on the fact that sizes of pages and page frames are identical
  - Instead, we can only translate the first virtual address in a page to the first virtual address in a page frame in a physical memory. And then the remaining addresses in the virtual memory page will map to the corresponding offsets in the physical memory page frame
  - As a result, reducing the number of entries we maintain in the page table

1- only the first portion of the virtual address is used to index into the page table. **virtual page number**, and the rest of the virtual address is the actual offset in the v-memory

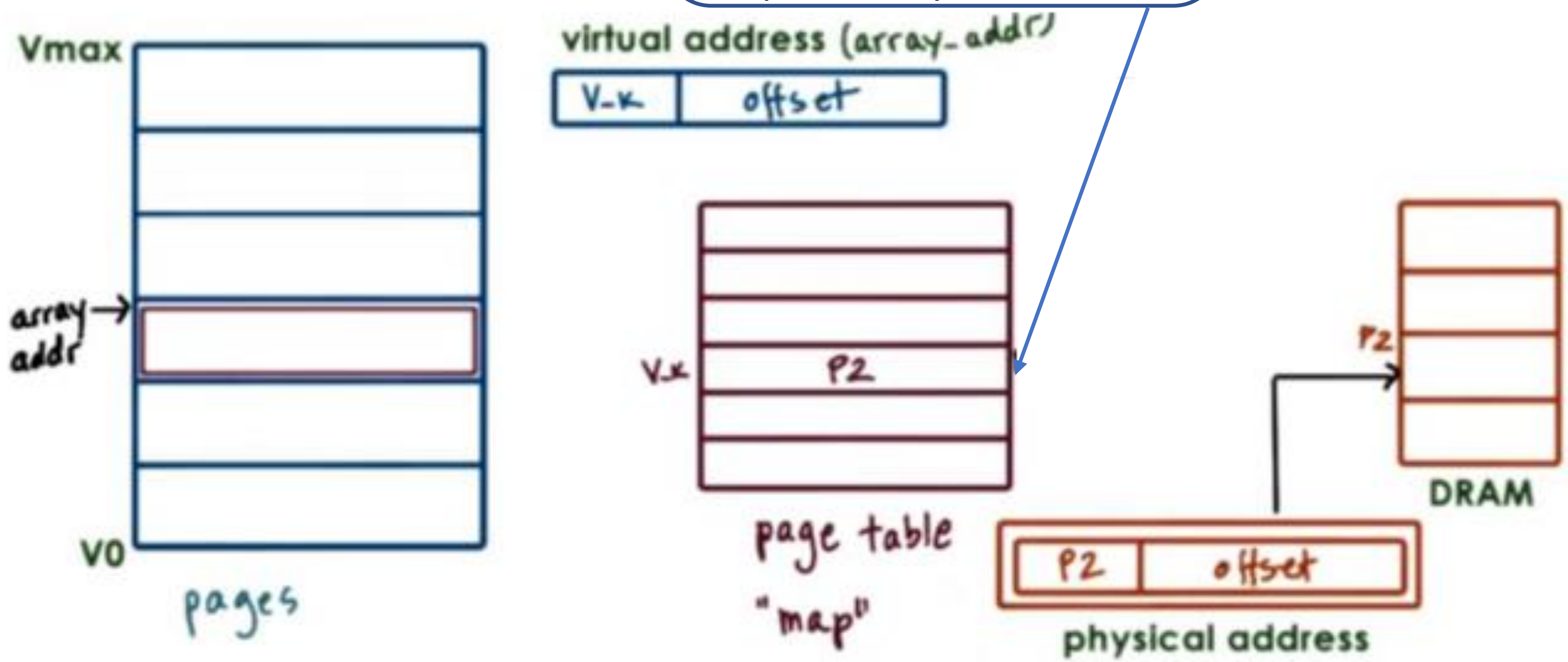
3- This **physical frame number** needs to be sent with the offset (of the DRAM) that's specified in the later part of the physical address to produce the actual **physical address**



2- The **virtual page number** is used as an offset into the page table. And that will produce the **physical frame number**

4- The resulting **physical address** can ultimately be used to reference the appropriate location in physical memory

3- Establish a mapping between the virtual address (**V-K**) and the **offset address** in the v-memory and P2 in the p-memory



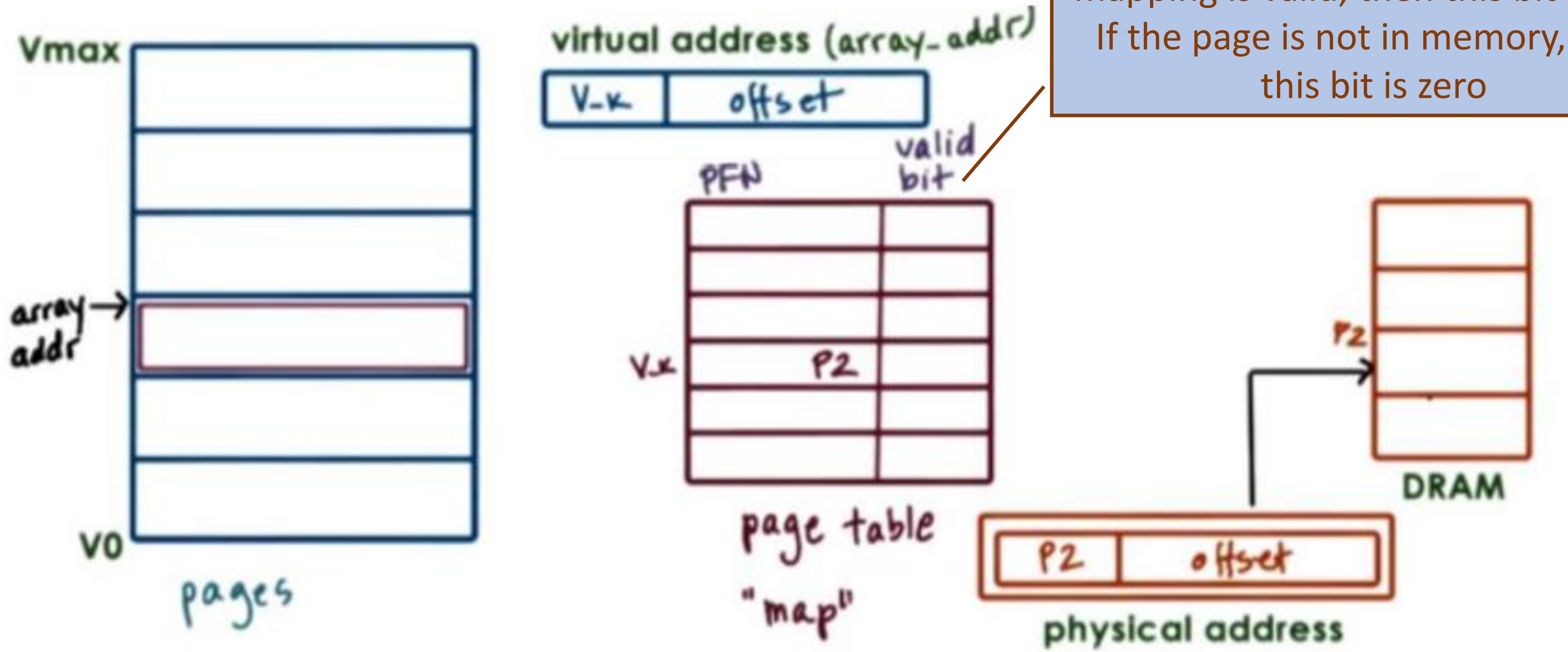
2-there isn't physical memory that corresponds to this range of virtual memory addresses, then use P2 since its free

1- Initialize the array and the array address

Notes ….

- Note, the physical memory for the array in the virtual address space is only allocated when the process is first trying to access it, during this initialization routine
- We refer to this as allocation on **first touch**. The reason for this is that to make sure that physical memory is allocated only when it is really needed, because sometimes programmer may create data structures that they don’t use
- If a process hasn’t used some of its memory pages for a long time, and it’s likely that those pages will be reclaimed. So the contents will no longer be present in physical memory. They will be reclaimed, they will be pushed on desks and probably some other content will find its way into the physical memory

unused pages == reclaimed



If the page is in memory and the mapping is valid, then this bit is one. If the page is not in memory, then this bit is zero

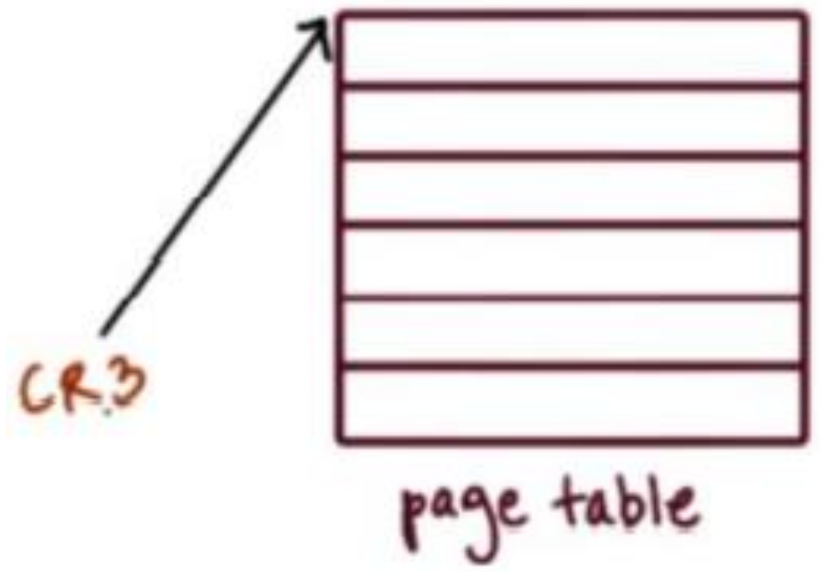
If the hardware determines that the mapping is invalid and false, then control gets passed to the operating system. The OS at that point gets to decide on number of questions:

- Should the access be permitted
- Where exactly is the page located
- Where should it be brought into DRAM

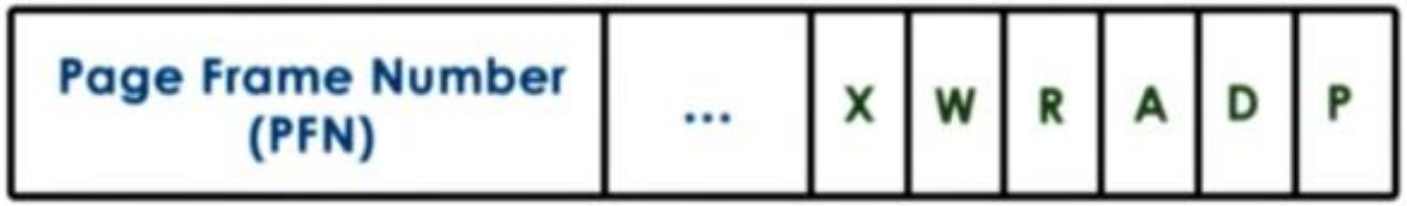
Page Tables

- The hardware assist with page table accesses by maintaining a register that points to the active page table

- On X86 Platforms it uses register CR3, so basically on a context switch update the contents of the CR3 register with the address of the new page table

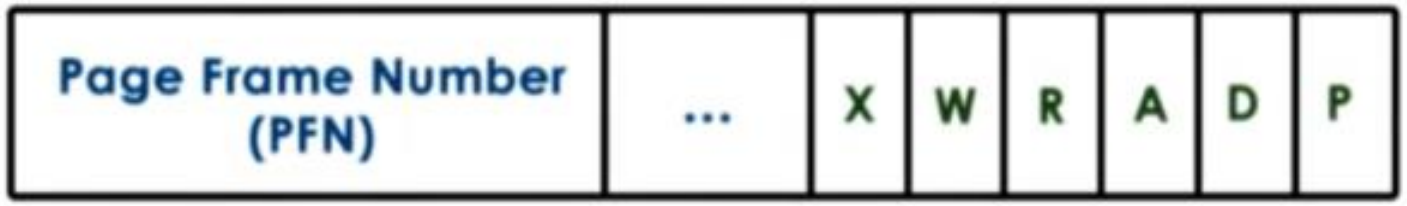


General Page Table Entry



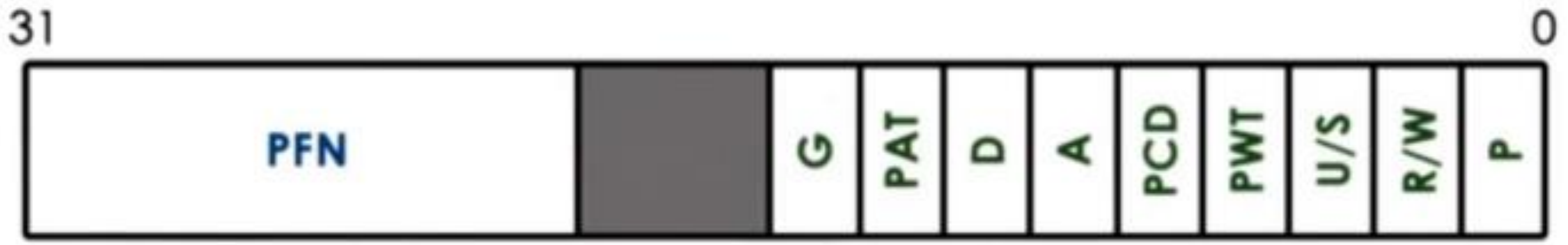
- Every page table entry will have the physical page frame number and that it will also have at least a valid bit
- Valid bit also called present bit, it indicates whether the contents of the virtual memory are present in the physical memory or not
- Dirty bit, which gets set whenever a page is written to. This is useful in file systems, where files are cached in memory. And then we can detect using this dirty bit which files have been written to and need to be updated on disk

General Page Table Entry… continued



- Accessed bit, this can keep track of in general whether the page has been accessed for read or for write
- Other useful information that can be maintained as part of the page table entry also would include certain protection bits. Whether a page can be only read or also written to, or maybe some other operation is permissible (R, W, X)

Pentium x86

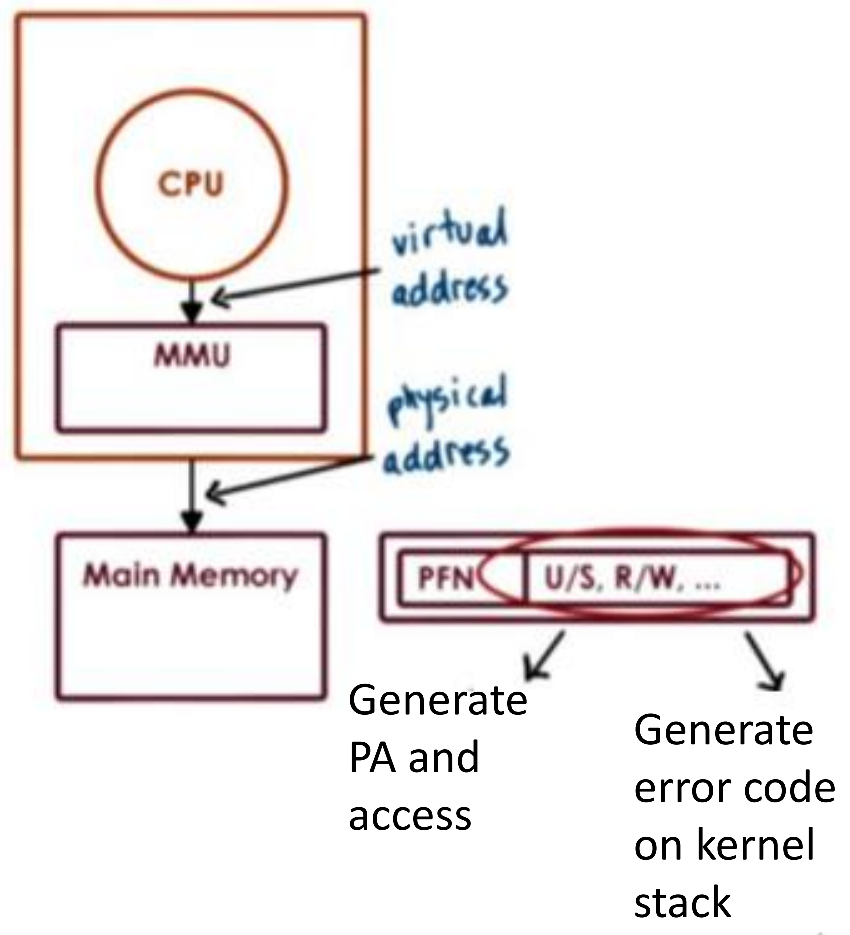


- Flags present, dirty, and accessed, have identical meaning as in the generic page table entry
- The bit read/write, it’s a single bit that indicates permission
  - 0 = read only
  - 1 = 1 read and write
- U/S bit indicates whether the page can be accessed from user mode or only from supervisor mode from when you’re in the kernel
- Other flags dictate some things regarding the behavior of the caching subsystem that the hardware has
- **Unused** bits for future use

Page Fault

- Page tables:
    - Addresses translation
    - Access validation

- If the hardware determines that a physical memory access cannot be performed, it causes a page fault

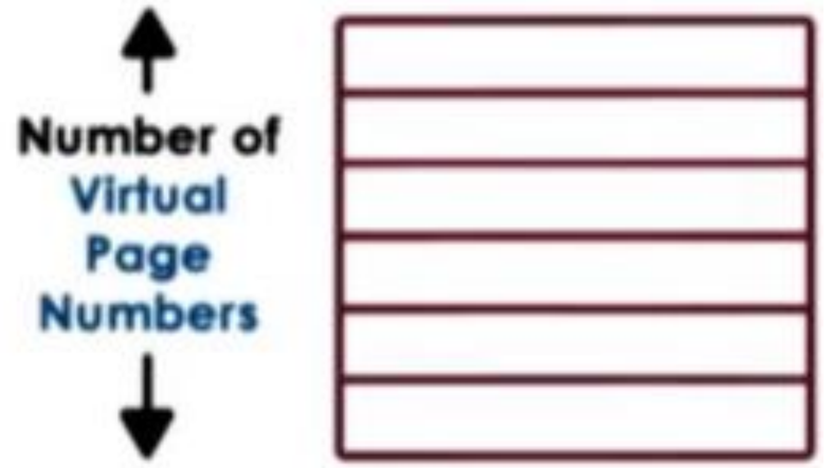


Page Faults in Practice

- Then the CPU will place an error code on the stack of the kernel, and then it will generate a trap into the OS kernel. That will in turn generate a page fault handler
- The page fault handler will determine what is the action that needs to be taken depending on the error code as well as the address that caused the fault
- Faults:
  - Missing pages, that it needs to be brought from disk
  - Some sort of permission protection that was violated, and that's why the page access is forbidden

Page Table Size

- A page table has number of entries that is equal to the number of virtual page numbers that exist in a virtual address space
- For every one of these entries, the page table needs to hold the physical frame number, as well as some other information like permission bits
- On a 32-bit architecture, each of the page table entries is 4 bytes and that includes the page frame number as well as the flags



“The total number of page table entries, that will depend on the total number of VPNs. And how many VPNs we can have, that’s going to depend on the size of the addresses, of the virtual addresses, and of the page size itself”

- Example: we have a 32-bit, both physical memory as well as 32-bit virtual addresses. NOTE, different platforms have different page sizes, here picked as 4KB as a common page size

$$2\hat{}32/_{PAGE\ SIZE} = (2\hat{}32/_{2\hat{}12}) * 4B$$

= 4 MB/process

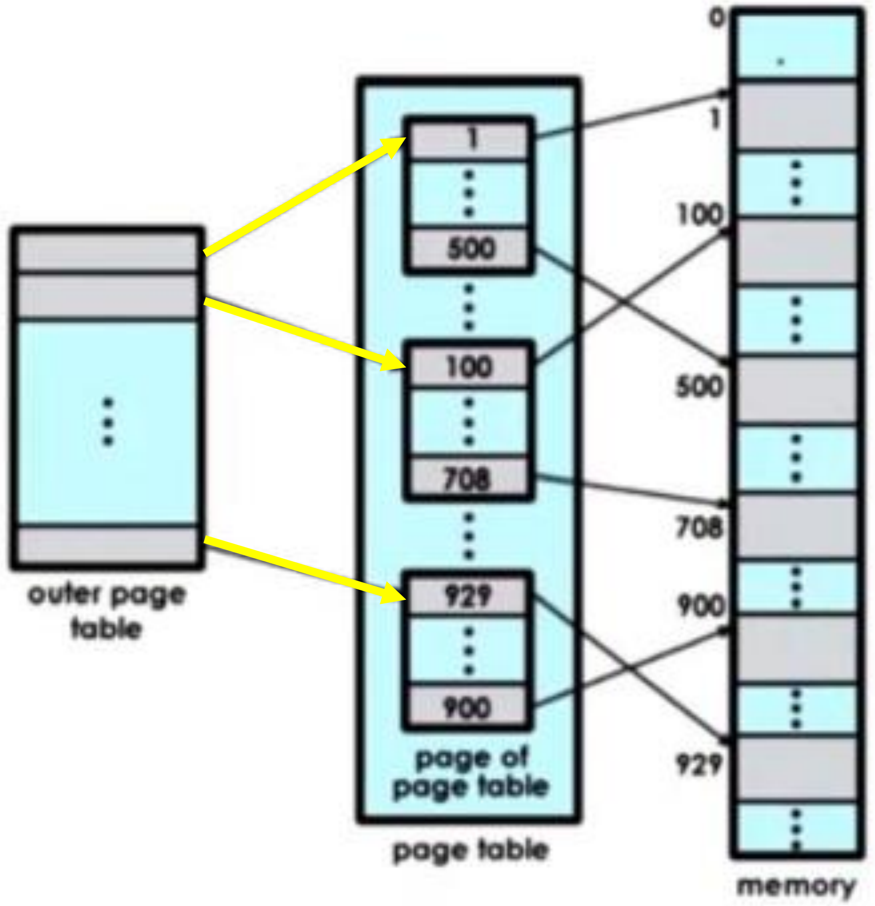
- OR 64-bit architecture →$(2\hat{}46/_{2\hat{}12}) * 8B$ = $32PB$!!

Where does one store all of this?

- A process likely will not use all the theoretical available virtual memory. Even on 32-bit architecture it’s not all the 4 GB of virtual address space is used by every single type of process
- The problem is that the page table as described so far, it assumes that there is an entry for every single VPN. And that is regardless of whether the corresponding virtual memory region is needed by the process or not

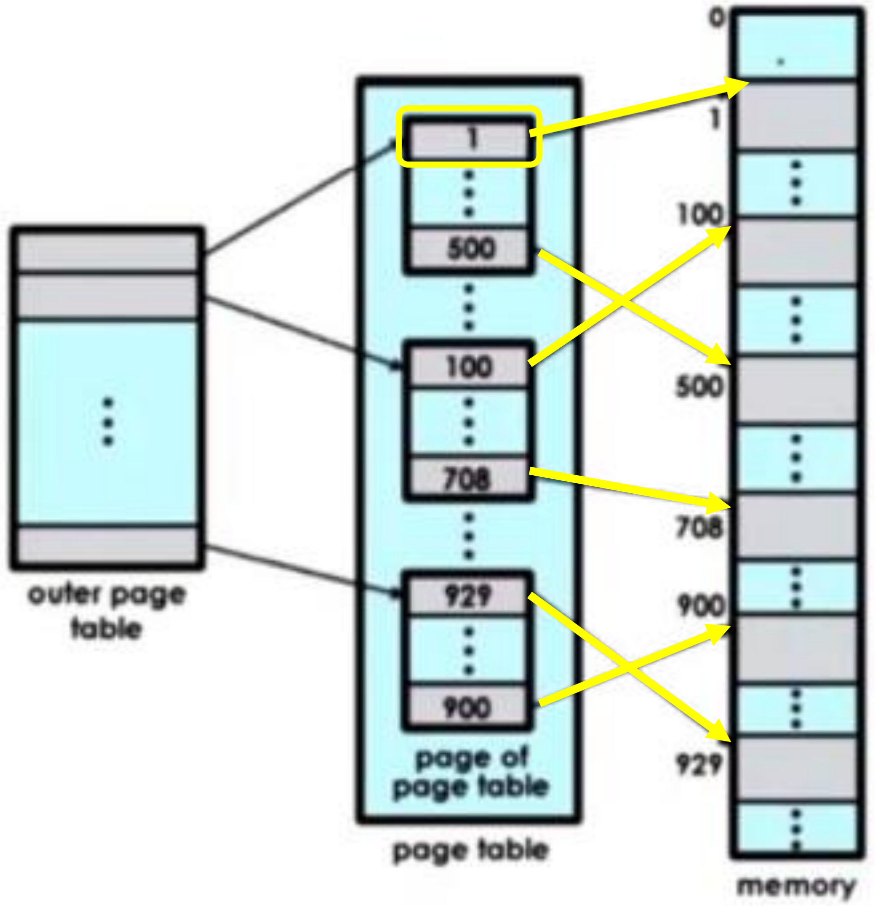
Multi Level Page Tables

- Page tables have evolved from a flat page map to a more hierarchical multilevel structure
- This figure here shows a two-level page table
- Outer page table→ Page table directory represents entries for pointers to page tables



Multi Level Page Tables

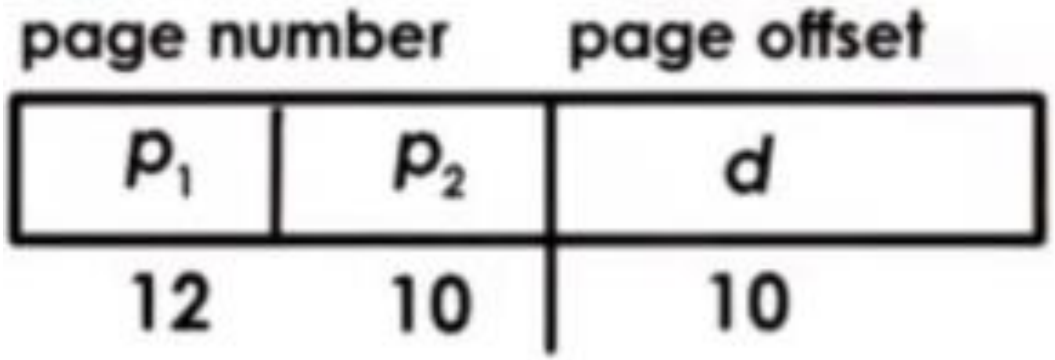
- The internal page has proper page tables as its components that actually point to page tables
- The entry consists of page frame number and all the protection that's for the physical addresses that are referenced by the corresponding virtual address
- Exist only for those virtual memory regions that are actually **valid**



- Any kinds of holes in the virtual memory space will result in lack of internal page tables, so for those holes there won’t be internal page tables allocated for the virtual addresses
- If a process requests via malloc additional virtual memory to be allocated to it, the OS will check and if necessary, it will allocate an additional internal page table element and set the appropriate page table directory to correspond to that entry
- Then new internal page table entry will correspond to some portion of the newly allocated virtual memory region that the process has requested

Address Split

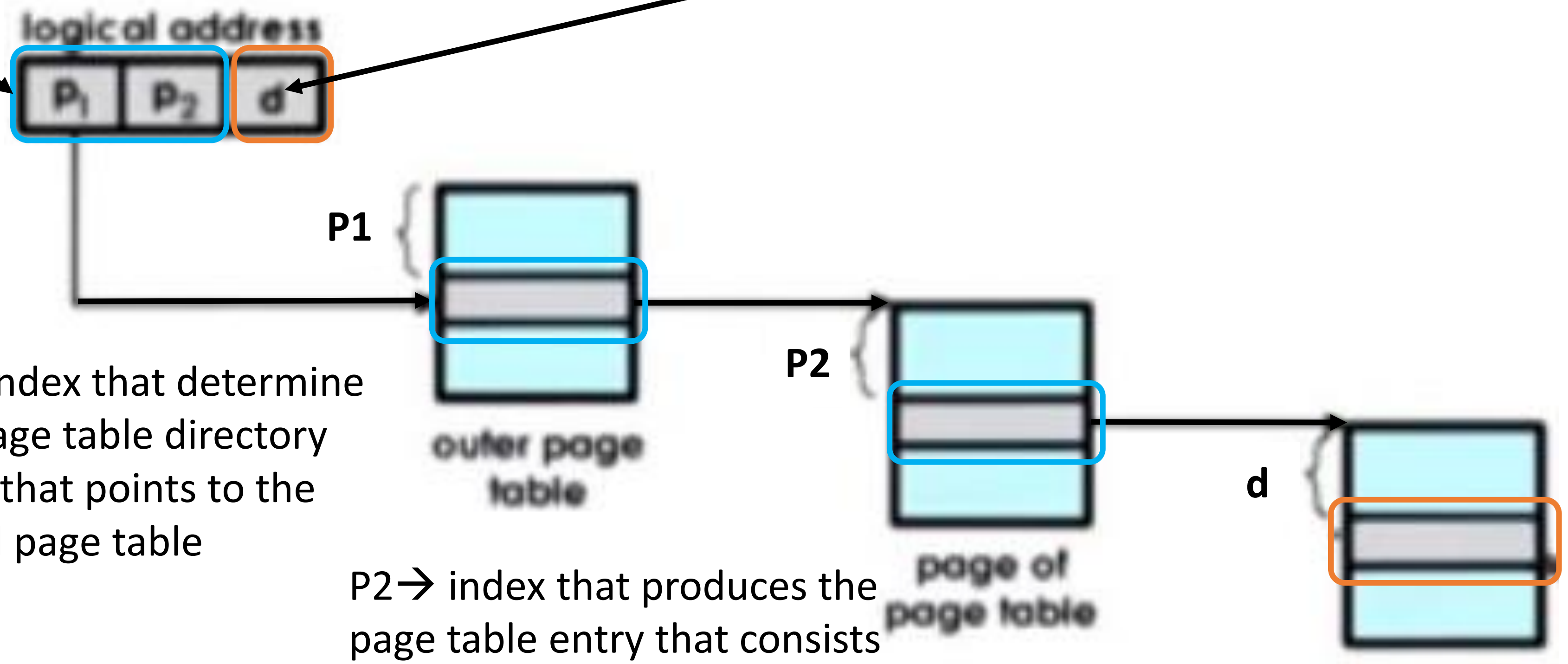
- To find the right element in this page table structure, the virtual address is split into yet another component
- Using this address format, is what's needed to determine the correct physical address



Hierarchical Page Tables

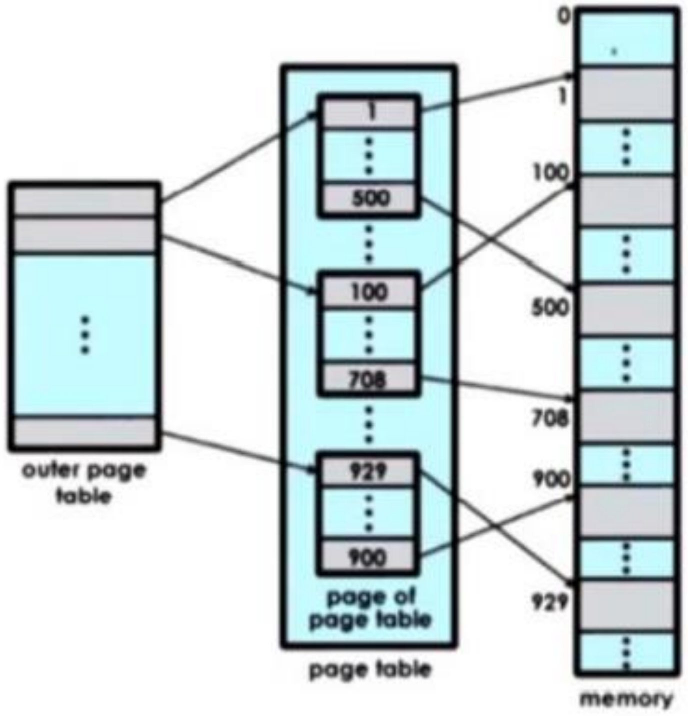
Indices into the different levels of the page table hierarchy. And they're ultimately going to produce the physical frame number that's the starting address of the physical region

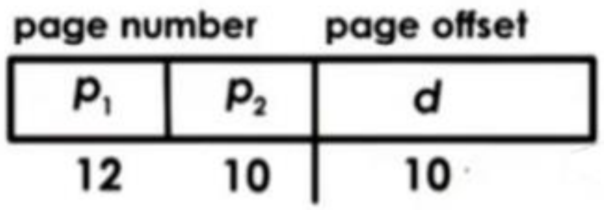
The last portion of the address is still the offset so that's going to be used to compute the offset within the actual physical page



P1→Index that determine the page table directory entry that points to the actual page table

P2→ index that produces the page table entry that consists of the physical frame number





Inner table addresses

➔2^10 * page size

= 2^10 * 2^10

= 1MB

So, 2^10th pages can be addressed in the internal page table

Notes

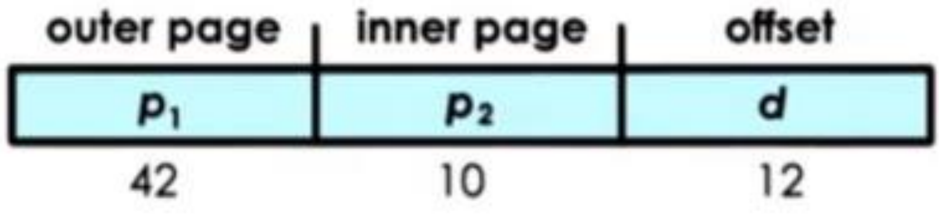
- For single level page table design where the page table has to be able to translate every single virtual address and it has entries for every single virtual page number
- Every single internal page table can address 2^{10}th the number of entries times the page size, that’s another 2^{10}th so 1MB of memory
- This means that whenever there is gap in the virtual memory that’s 1MB size, we don’t need to allocate that internal page table so that will reduce the overall size of the page table that’s required for a particular process

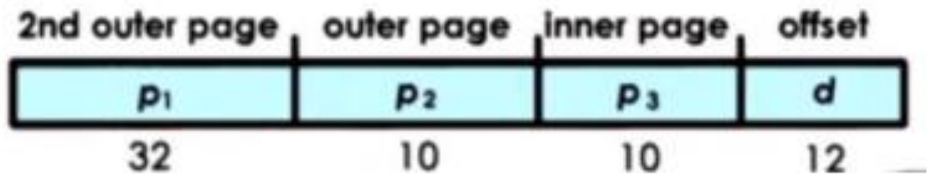
Expanding the solution

- The solution can be further extended to use additional layers using the same principle, third and fourth levels
- Yet 4th level, which consists of a map of pointers to page table directories
  - Important on 64-bit architectures
  - Page table requirements are longer and more sparse
  - More sparse, means that it will have larger gaps in the virtual address space region
  - And the larger the gaps, the larger the number of internal page table components that won’t be necessary as a result of that gap. Saving entire page table directories as a result of certain gaps in the virtual address space

Hierarchical Page Tables Example

- 64-bit virtual address interpretation
- Two-page table layers vs three-page table layers
- The offset field is the index into the physical page table





- Smaller internal page tables/directories granularity of coverage → potential reduce page table size

- Downside, is more memory access required for translation→ increased translation latency

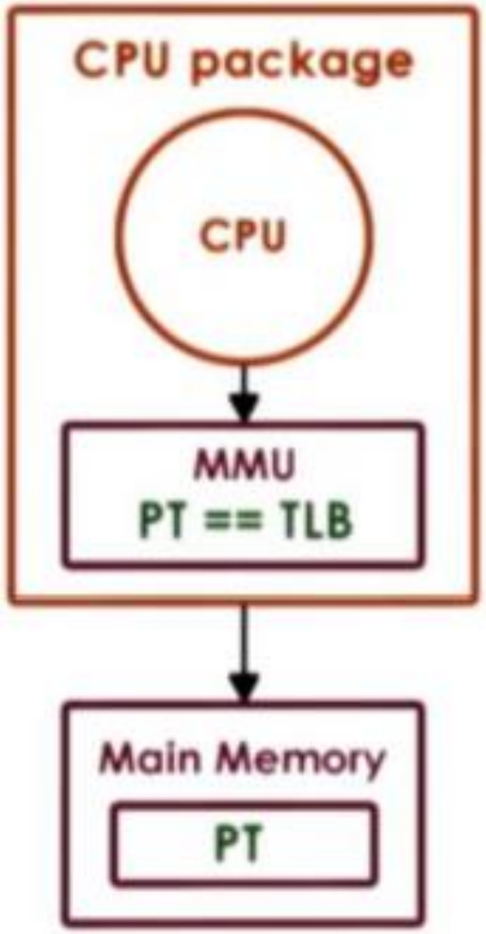
Speeding Up Translation TLB

- For a single level page-table design, two memory references required:
  - x1, To access the page table entries to determine the physical frame number
  - x1, Perform the proper memory access at the correct physical address
- For a 4-level page-table:
  - x4, Four memory accesses to read the page table entries at each level of the memory hierarchy
  - x1, then produce the physical frame number

very costly and can lead to a slowdown

The Fix

- Use page-table cache
- Dedicated for caching address translations, and this cache is called the Translation Look Aside Buffer or TLB
- On each address translation first the TLB cache is quickly referenced and if the resulting address can be generated from the TLB contents then we have a TLB hit, and we can bypass all of the other required memory accesses to perform the translation
- Of course, if we have a TLB miss, so the address isn't present in the TLB cache, perform all the address translation steps. If necessary, generate a fault



More on TLB

- Protection / validity bit
- Small number of cache address leads to high TLB hit
- Temporal and spatial locality will lead to increased TLB rate even with small number of entries
- Example x86 core i7
  - Per core: 64-entry data TLB
    128-entry instruction TLB
    512-entry shared second level TLB

The End