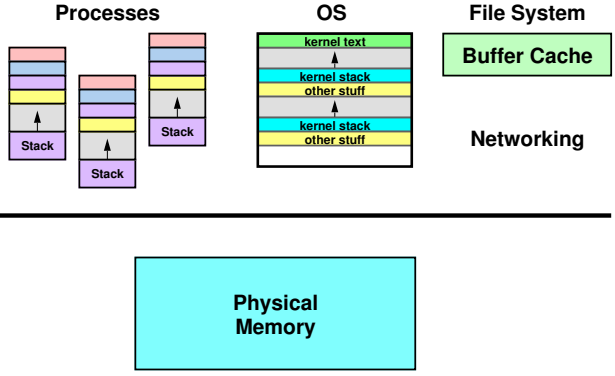
Ch 7: Memory Management

Bill Cheng

http://merlot.usc.edu/cs402-s16

Memory Management

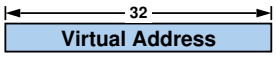


- Challenges
  - what to do when you run out of space?
  - protection

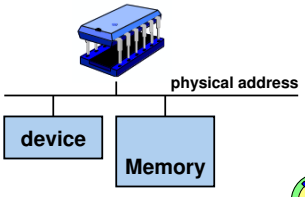
The Address-Space Concept

- Protect processes from one another
- Protect the OS from user processes
- Provide efficient management of available storage
  - illusion of large memory
  - sharing (code, data, communication)
  - new abstraction (such as pipes, memory-mapped files)

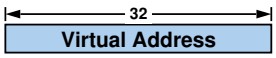
Virtual Address



- Who uses *virtual address*?
  - user processes
  - kernel processes
  - pretty much every piece of software
- You would use a virtual address to address any memory location in the 32-bit address space
- Anything uses *physical address*?
  - nothing in OS
  - well, the hardware uses physical address (and the processor is hardware)
  - the OS *manages* the *physical address space*

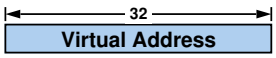


Virtual Address



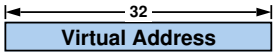
- To *access a memory location*, you need to specify a *memory address*
  - in a user process (or even a kernel process), you would use a *virtual address* to address any memory location in the 32-bit address space
- Why would you want to access a memory location?
  - e.g., to fetch a machine instruction
    - you need to specify a memory location to fetch from
    - how do you know which memory location to fetch from?
      - EIP (on an x86 machine), which contains a virtual address

Virtual Address



- To *access a memory location*, you need to specify a *memory address*
  - in a user process (or even a kernel process), you would use a *virtual address* to address any memory location in the 32-bit address space
- Why would you want to access a memory location?
  - e.g., to fetch a machine instruction
  - e.g., to push EBP onto the stack
    - you need to specify a memory location to store the content of EBP
    - how do you know which memory location to write to?
      - ESP, which contains a virtual address

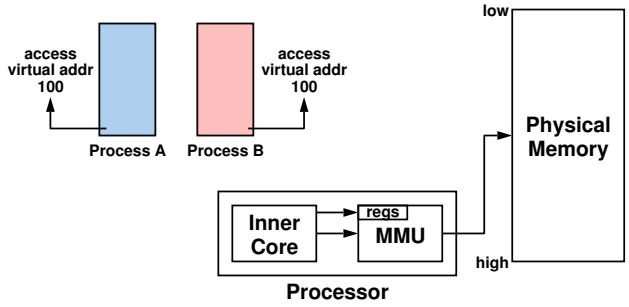
Virtual Address



- To *access a memory location*, you need to specify a *memory address*
  - in a user process (or even a kernel process), you would use a *virtual address* to address any memory location in the 32-bit address space
- Why would you want to access a memory location?
  - e.g., to fetch a machine instruction
  - e.g., to push EBP onto the stack
  - e.g., `x = 123`, where `x` is a local variable
    - you need to specify a memory location to write `123` to
    - how do you know which memory location to wrote to?
      - EBP, which contains a virtual address

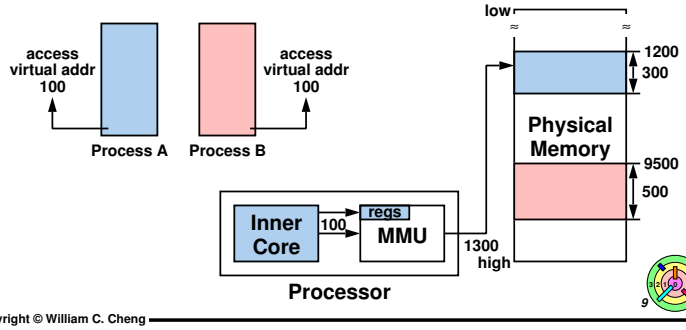
Basic Idea: Address Translation

- One level of *indirection* with a *Memory Management Unit (MMU)*
  - don't address physical memory directly
    - address out of CPU inner core is *virtual*
  - use a *Memory Management Unit (MMU)*
    - virtual address is *translated* into physical address via MMU



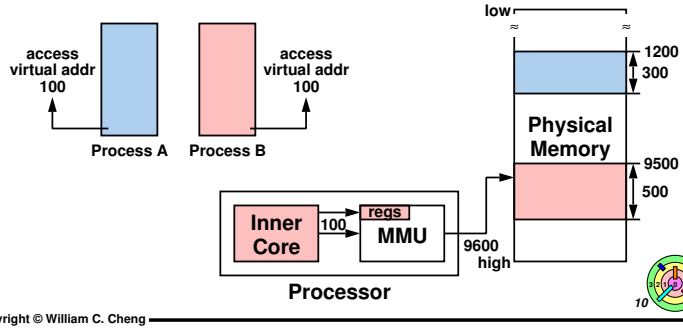
Basic Idea: Address Translation

- One level of *indirection* with a *Memory Management Unit (MMU)*
  - don't address physical memory directly
    - address out of CPU inner core is *virtual*
  - use a *Memory Management Unit (MMU)*
    - virtual address is *translated* into physical address via MMU



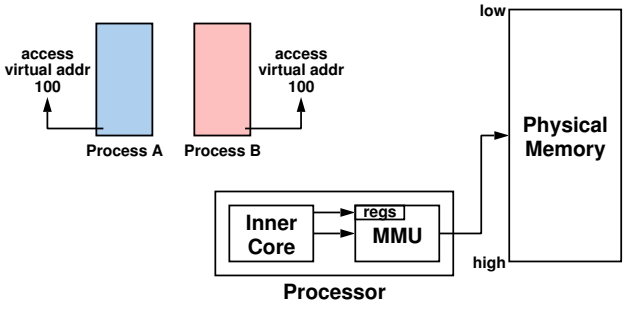
Basic Idea: Address Translation

- One level of *indirection* with a *Memory Management Unit (MMU)*
  - don't address physical memory directly
    - address out of CPU inner core is *virtual*
  - use a *Memory Management Unit (MMU)*
    - virtual address is *translated* into physical address via MMU

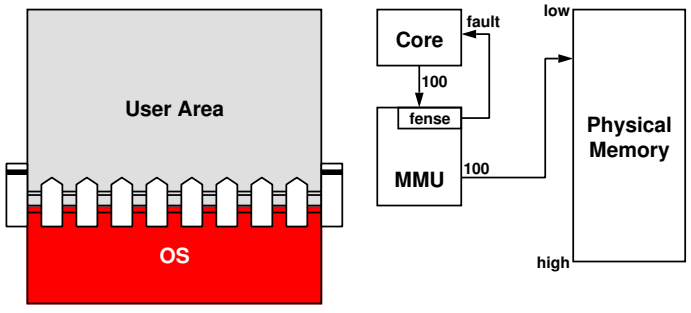


Address Translation

- Protection/isolation
- Illusion of large memory
- Sharing
- New abstraction (such as memory-mapped files)

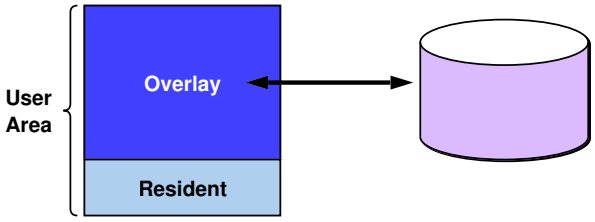


Memory Fence



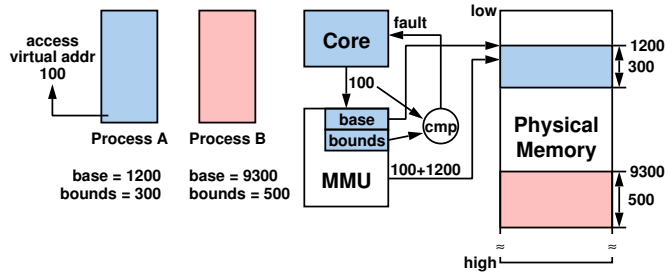
- In the old days
  - if a user program tries to access OS area, *hardware* (very simple MMU) will generate a trap
  - does not protect user pocesses from each other
    - there's only one user process anyway

Memory Fence and Overlays



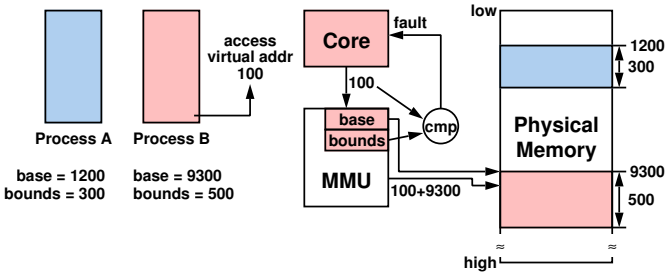
- What if the user program won't fit in memory?
  - use *overlays*
  - programmers (not the OS) have to keep track of which overlay is in physical memory and deal with the complexities of managing *overlays*

Base and Bounds Registers



- Multiple user processes
  - OS maintains a pair of registers for each user process
    - *bounds register:* address space size of the user process
    - *base register:* start of physical memory for the user process
  - addresses *relative* to the *base register*
  - memory reference >= 0 and < bounds, *independent* of *base* (this is known as "position independence")

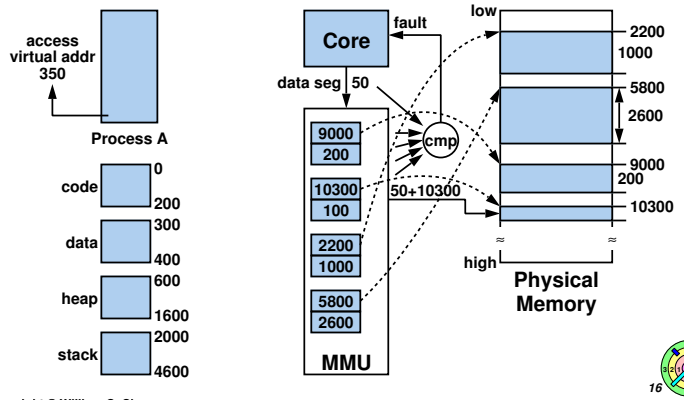
Base and Bounds Registers



- Multiple user processes
  - OS maintains a pair of registers for each user process
    - *bounds register:* address space size of the user process
    - *base register:* start of physical memory for the user process
  - addresses *relative* to the *base register*
  - memory reference >= 0 and < bounds, *independent* of *base* (this is known as "position independence")

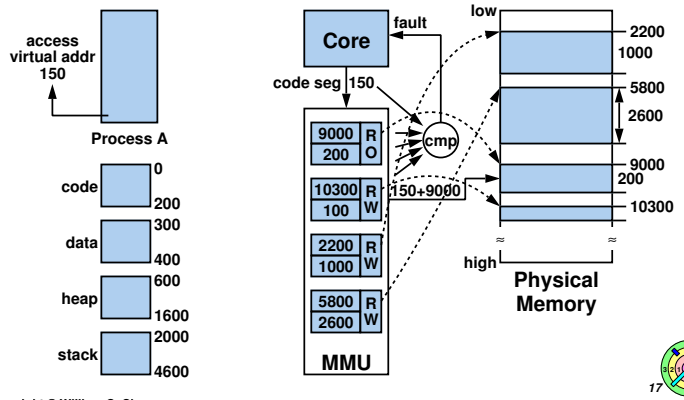
Generalization of Base and Bounds: Segmentation

- One pair of *base* and *bounds* registers *per segment*
  - code, data, heap, stack, and may be more
  - compiler compiles programs into segments



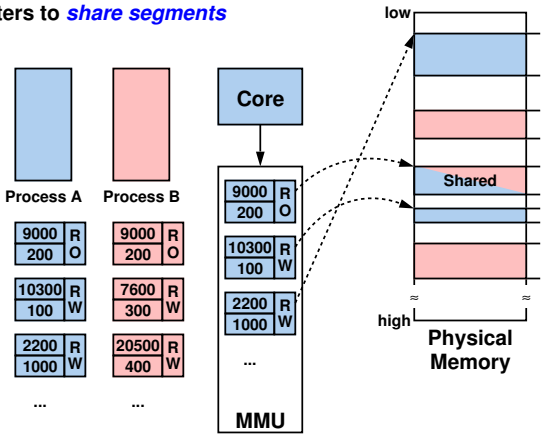
Access Control With Segmentation

- Access control / protection
  - *read-only*, *read/write*



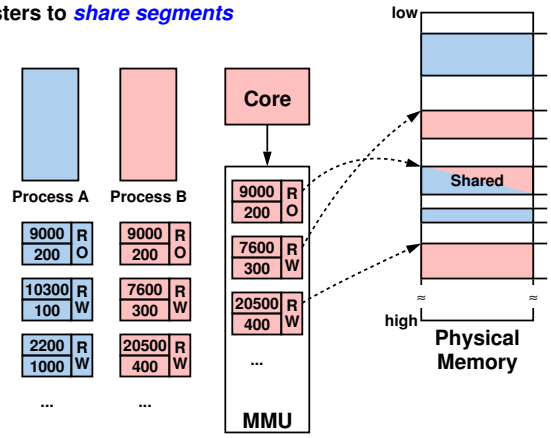
Sharing Segments

- Can simply setup base and bounds registers to *share segments*



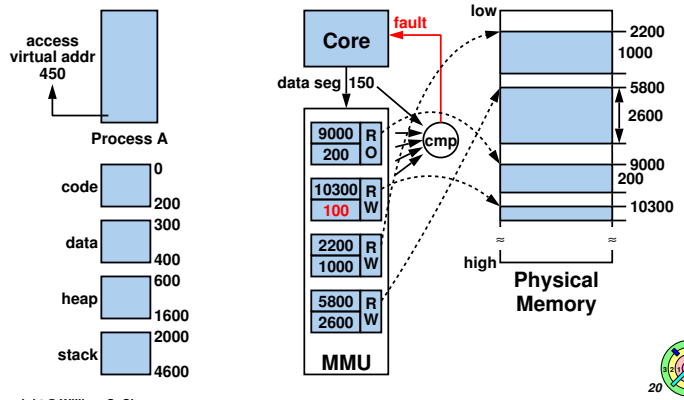
Sharing Segments

- Can simply setup base and bounds registers to *share segments*



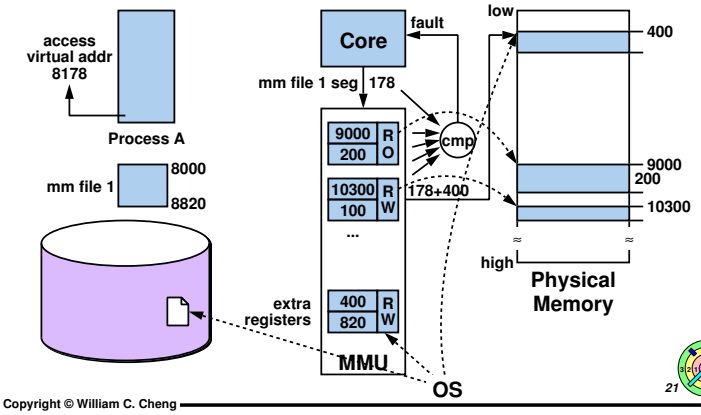
Segmentation Fault

- *Segmentation fault*
  - virtual address not within range of any base-bounds registers



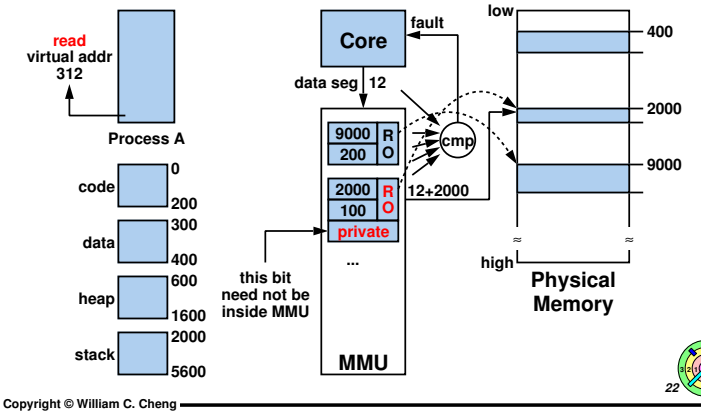
Memory Mapped File

- *Memory Mapped File*
  - the `mmap()` system call
  - can map an entire file (or part of it) into a segment



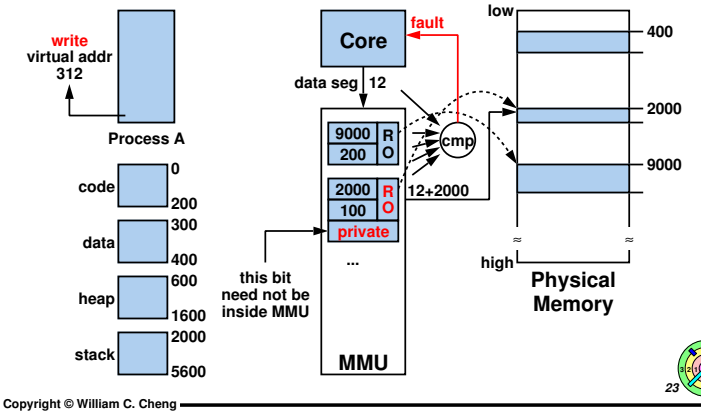
Copy-On-Write

- *Copy-on-write (COW):*
  - a process gets a *private* copy of the page after a thread in the process performs a *write* for the *first time*



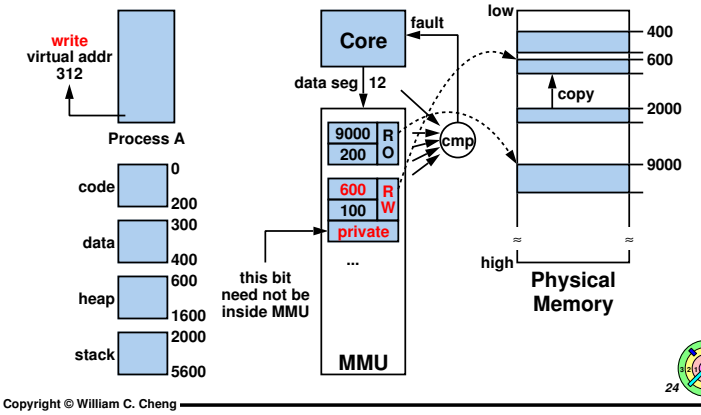
Copy-On-Write

- *Copy-on-write (COW):*
  - a process gets a *private* copy of the page after a thread in the process performs a *write* for the *first time*



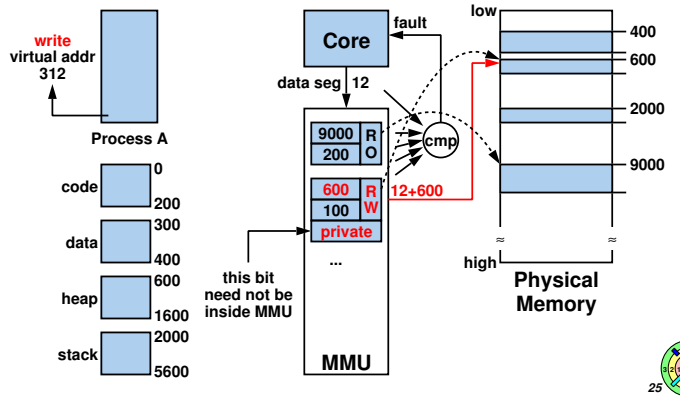
Copy-On-Write

- *Copy-on-write (COW):*
  - a process gets a *private* copy of the page after a thread in the process performs a *write* for the *first time*



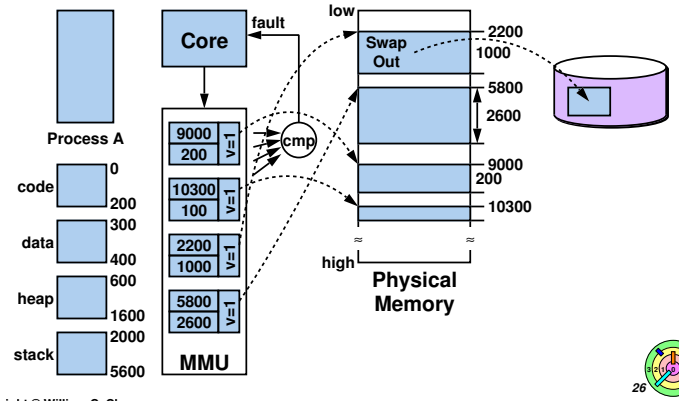
Copy-On-Write

- *Copy-on-write (COW):*
  - a process gets a *private* copy of the page after a thread in the process performs a *write* for the *first time*



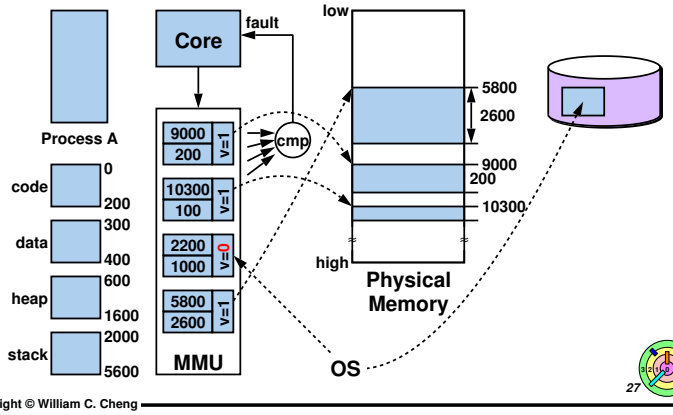
Swapping / Backing Store

- No space for new segment, make room by swapping out a segment
  - use a *validity* bit for each segment (in addition to access control bits)



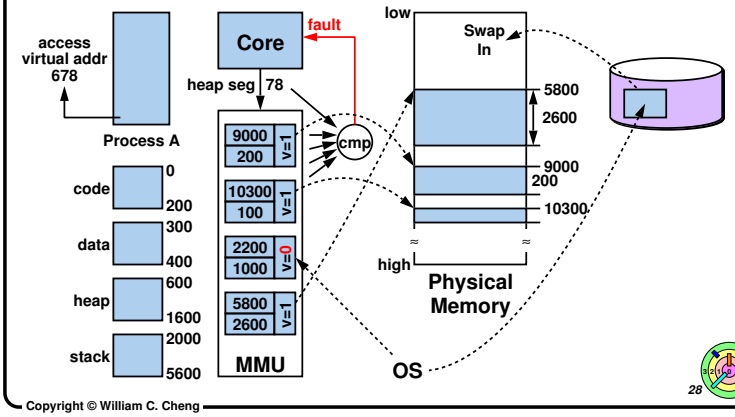
Swapping / Backing Store

- No space for new segment, make room by swapping out a segment
  - use a *validity* bit for each segment (in addition to access control bits)



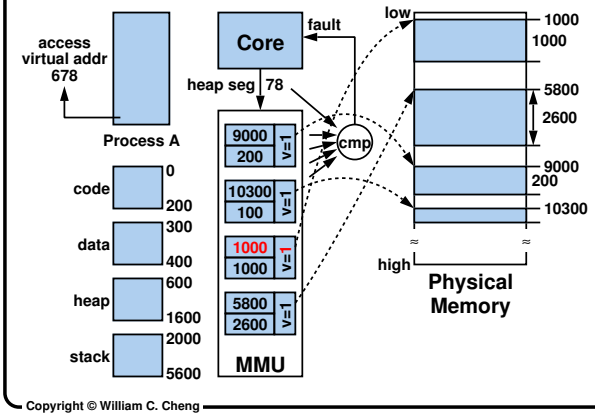
Swapping / Backing Store

- No space for new segment, make room by swapping out a segment
  - use a *validity* bit for each segment (in addition to access control bits)



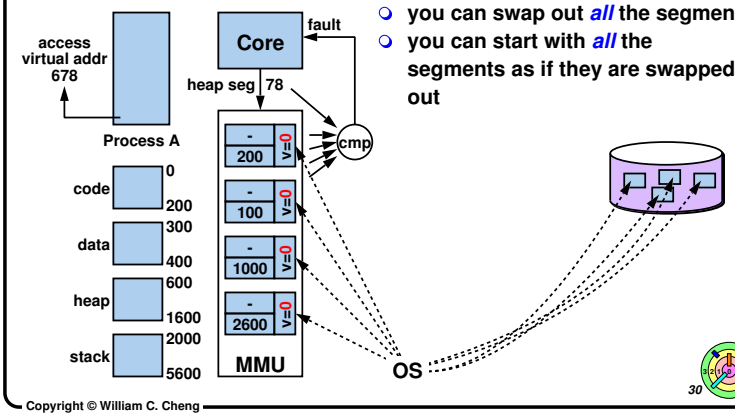
Swapping / Backing Store

- No space for new segment, make room by swapping out a segment
  - use a *validity* bit for each segment (in addition to access control bits)



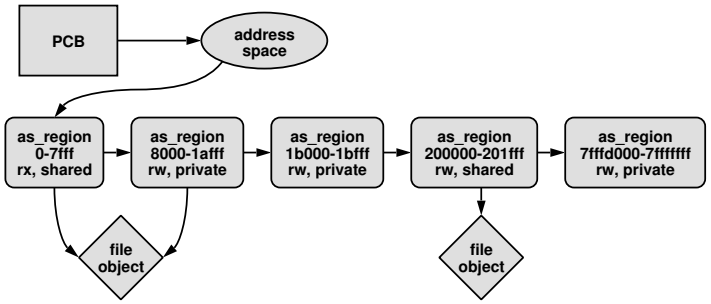
Swapping / Backing Store

- No space for new segment, make room by swapping out a segment
  - use a *validity* bit for each segment (in addition to access control bits)



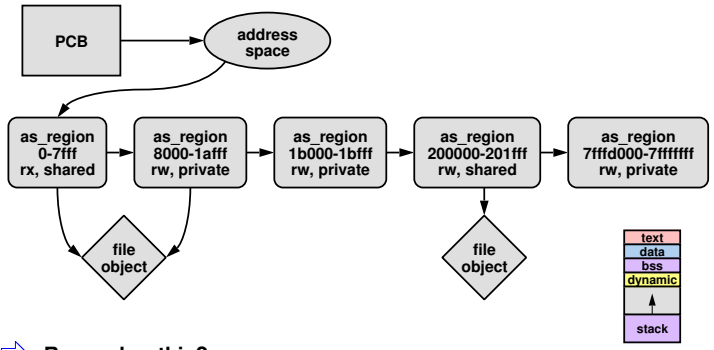
- you can swap out *all* the segments
- you can start with *all* the segments as if they are swapped out

Swapping / Backing Store



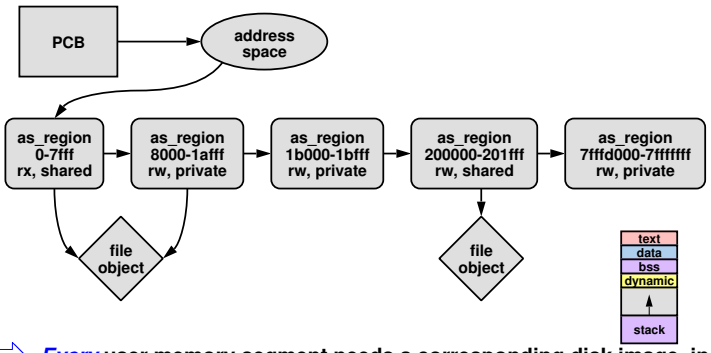
- Remember this?

Swapping / Backing Store



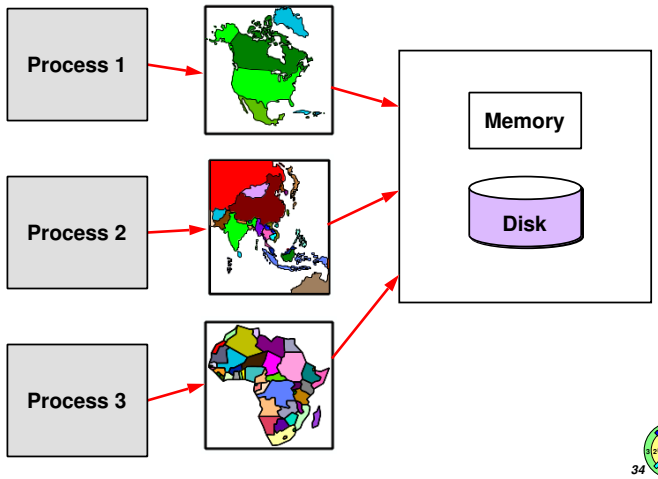
- Remember this?
  - this is the representation of the *address space* of a user process
    - each segment corresponds to an `as_region`

Swapping / Backing Store



- *Every* user memory segment needs a corresponding disk image, in case the user process needs to be *swapped out*
  - some kernel memory can be *locked down* to prevent it from being swapped out accidentally

Virtual Memory



7.2 Hardware Support for Virtual Memory

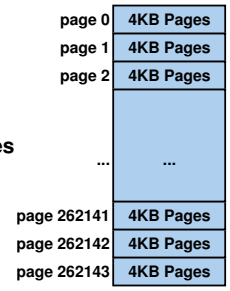
- *Forward-Mapped Page Tables*
- Linear Page Tables
- Hashes Page Tables
- Translation Lookaside Buffers
- 64-Bit Issues
- Virtualization

Structuring Virtual Memory

- *Segmentation* (just discussed)
  - divide the address space into variable-size segments (typically each corresponding to some logical unit of the program, such as a module or subroutine)
  - external fragmentation possible
  - "first-fit" is slow
  - not very common these days
- *Paging*
  - divide the address space into fixed-size pages
  - internal fragmentation possible

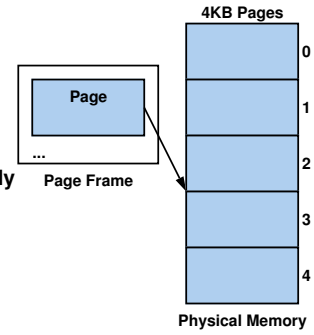
Paging

- Map *fixed-size pages* into physical memory *(into physical pages)*
  - address space is divided into pages
    - indexed by virtual page number
  - physical memory is divided into pages (of the same size)
    - indexed by physical page number
  - need a loopup table to map *virtual page numbers* to *physical page numbers*
- Ex: 1GB of physical memory with 4KB pages
  - 2^{18} physical pages
  - an address (either physical or virtual) is *page-aligned* if its least significant 12 bits are all zero
- Many *hardware* mapping techniques
  - *MMU* and *page table* (mostly in software)
  - *translation lookaside buffers (TLB)*



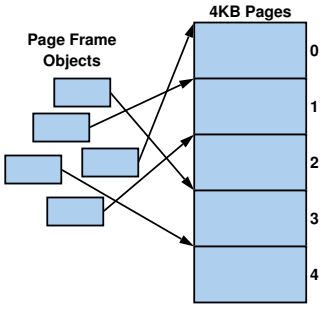
Page Frames

- A *page frame* data structure / object is used to maintain information about physical pages and their association with important kernel data structures
  - contains a *physical page number*
  - there is a *one-to-one mapping* between page frames and physical pages
    - we use "page frame" and "physical page" interchangeably

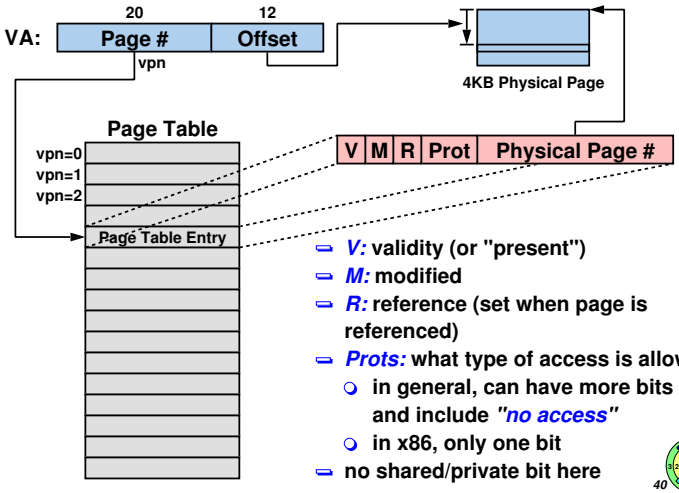


Page Frames

- It is important to be able to perform both *forward lookup* and *reverse lookup*
  - given a virtual address of a process, find page frame
  - given a page frame, find processes and virtual addresses that uses this page frame
  - `weenix` page frame data structure is a bit involved
    - see kernel 3 FAQ
    - the kernel must use a *virtual address* to write into a physical page or read the content of a physical page

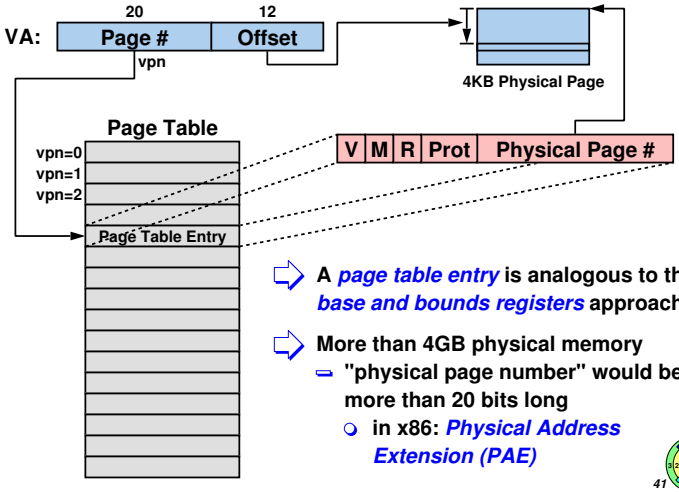


Basic (Two-level) Page Tables



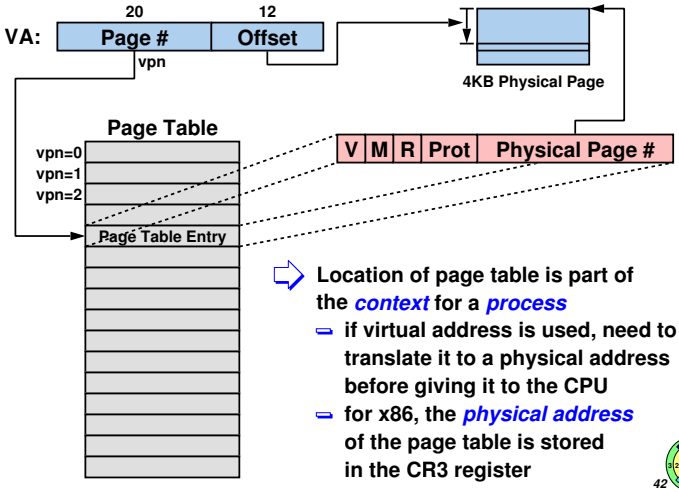
- *V:* validity (or "present")
- *M:* modified
- *R:* reference (set when page is referenced)
- *Prots:* what type of access is allowed
  - in general, can have more bits and include *"no access"*
  - in x86, only one bit
- no shared/private bit here

Basic (Two-level) Page Tables



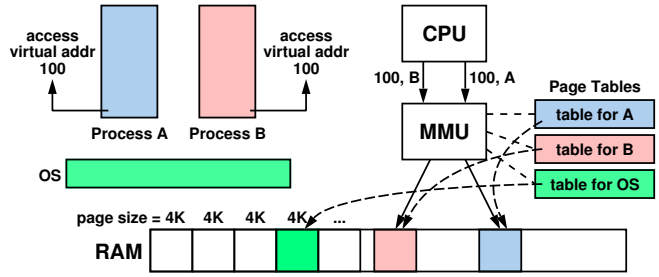
- A *page table entry* is analogous to the *base and bounds registers* approach
- More than 4GB physical memory
  - "physical page number" would be more than 20 bits long
    - in x86: *Physical Address Extension (PAE)*

Basic (Two-level) Page Tables



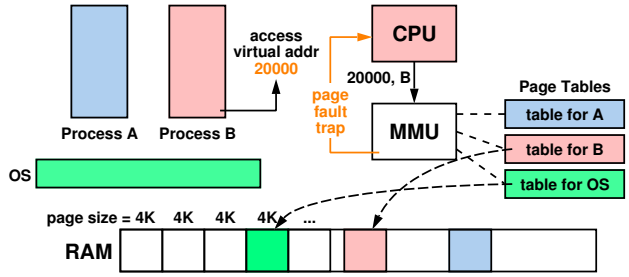
- Location of page table is part of the *context* for a *process*
  - if virtual address is used, need to translate it to a physical address before giving it to the CPU
  - for x86, the *physical address* of the page table is stored in the CR3 register

Page Table



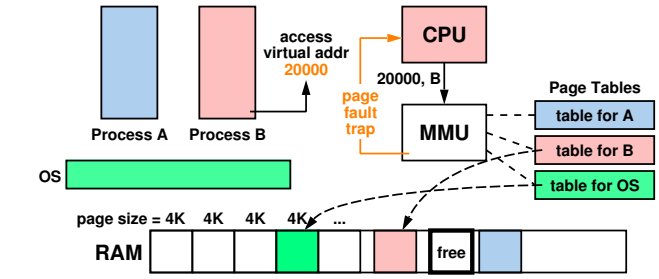
- A page table *(usually sits in physical memory)* is associated with each *process*
  - OS has its page table as well
- Memory Management Unit (MMU) maps virtual address to physical address
  - MMU got turned on some time during boot

Page Table



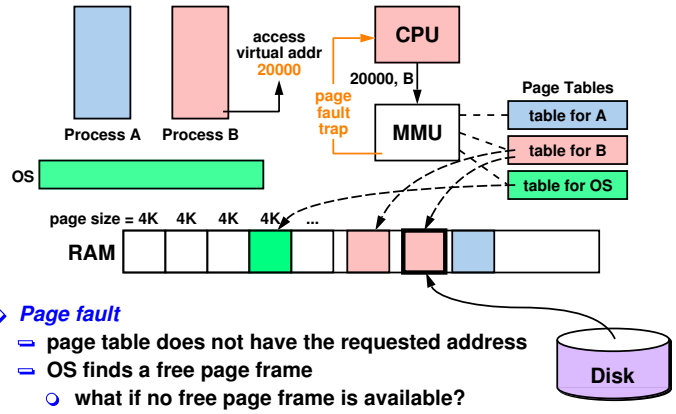
- *Page fault*
  - page table does not have the requested address

Page Table



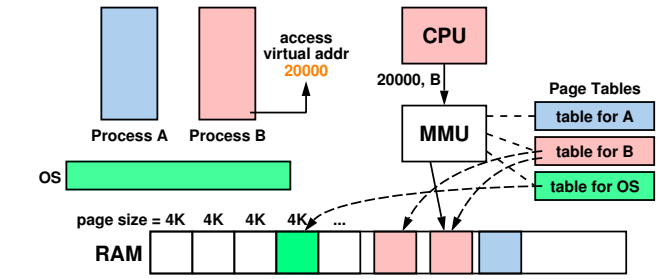
- *Page fault*
  - page table does not have the requested address
  - OS finds a free page frame
    - what if no free page frame is available?

Page Table



- *Page fault*
  - page table does not have the requested address
  - OS finds a free page frame
    - what if no free page frame is available?
  - OS loads the requested page from disk

Page Table

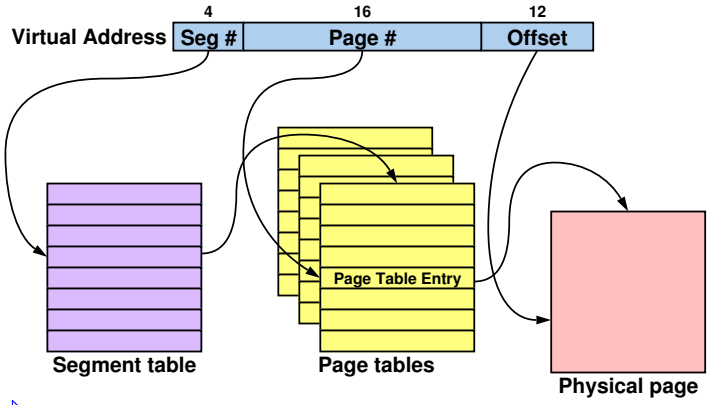


- *Page fault*
  - page table does not have the requested address
  - OS finds a free page frame
    - what if no free page frame is available?
  - OS loads the requested page from disk
  - OS adjusts MMU and *restarts* user memory reference

Page-Table Size

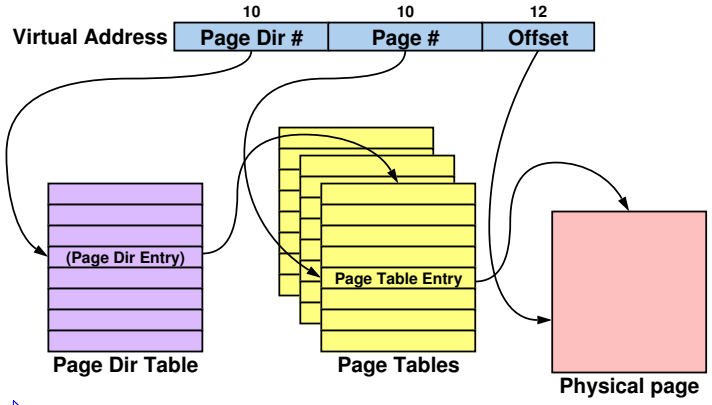
- Consider a full 2^{32}-byte address space
  - assume 4096-byte (2^{12}-byte) pages
  - 4 bytes per page table entry
  - the page table would consist of $2^{32}/2^{12}$ (= 2^{20}) entries
  - its size would be 2^{22} bytes (or 4 megabytes)
- This is a general *scaling* problem
  - solutions:
    - *hash page tables* or hierarchy *(forward-mapped page tables)*
    - *virtual linear page tables* (i.e., page tables in virtual memory)

Paged Segmentation



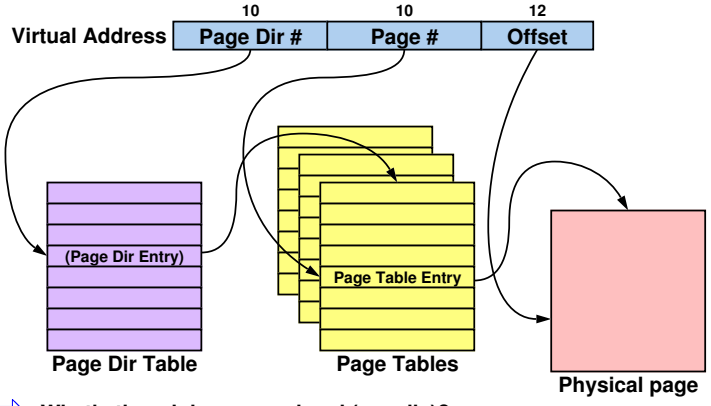
- If a process only uses 4 segments, it would need 4 page tables
  - a total of 256K page table entries (if an entry is 4 bytes long, this would take up 1MB)

Forward-Mapped (Multilevel) Page Table



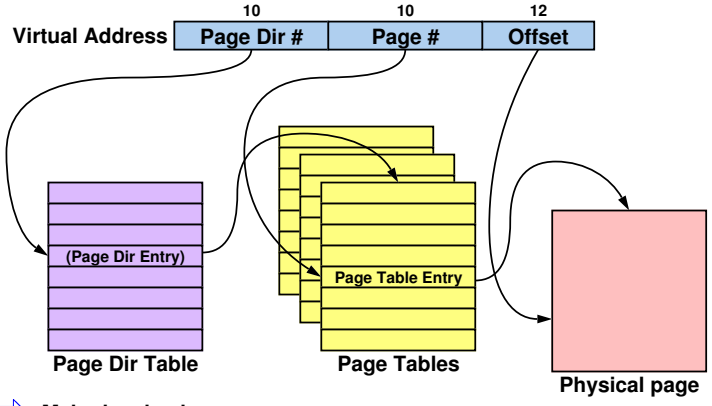
- What's the minimum overhead (usually)?

Forward-Mapped (Multilevel) Page Table



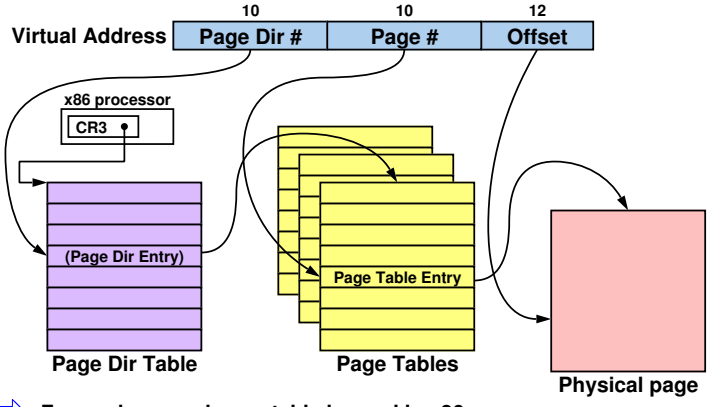
- What's the minimum overhead (usually)?
  - 12KB - one page dir page table and two page tables (one for low addresses and one for high addresses)

Forward-Mapped (Multilevel) Page Table



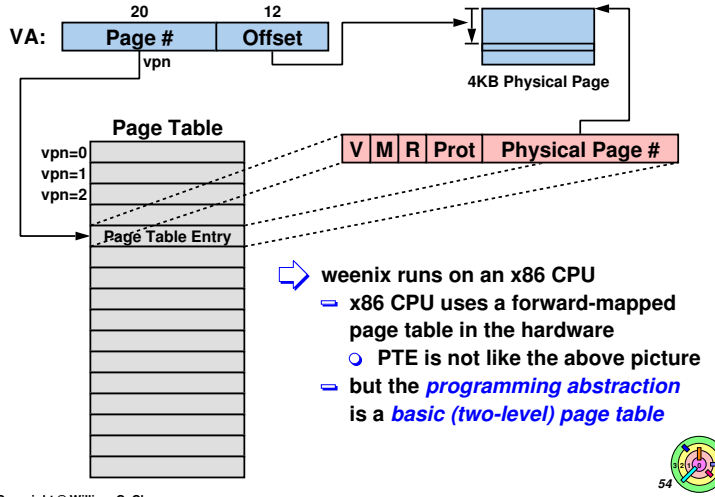
- Main drawback
  - two physical memory accesses just to *map* a virtual address to a physical address

Forward-Mapped (Multilevel) Page Table



- Forward-mapped page table is used in x86 processors
  - the CR3 register contains *physical address* of the currently running process

Forward-Mapped (Multilevel) Page Table

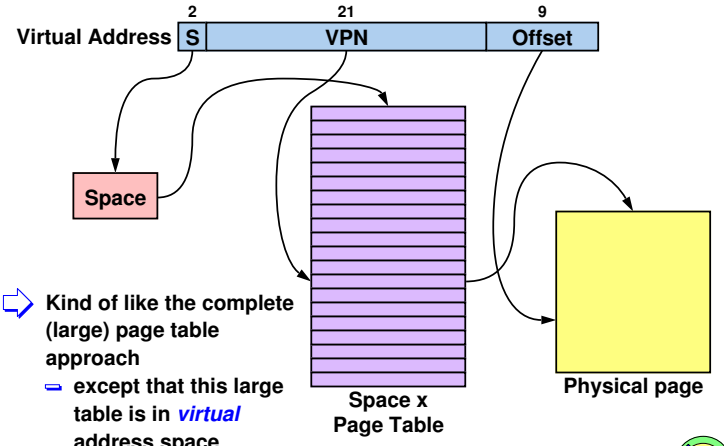


- weenix runs on an x86 CPU
  - x86 CPU uses a forward-mapped page table in the hardware
    - PTE is not like the above picture
  - but the *programming abstraction* is a *basic (two-level) page table*

7.2 Hardware Support for Virtual Memory

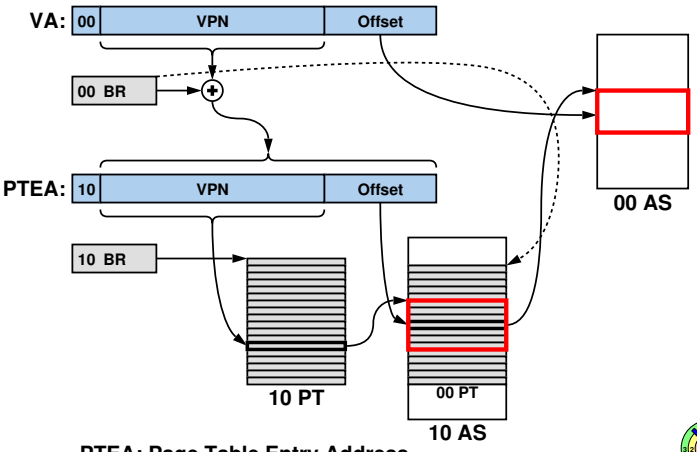
- Forward-Mapped Page Tables
- *Linear Page Tables*
- Hashes Page Tables
- Translation Lookaside Buffers
- 64-Bit Issues
- Virtualization

Linear Page Table



- Kind of like the complete (large) page table approach
  - except that this large table is in *virtual* address space
  - its entries may be cached inside the processor (in TLB)

VAX Linear Page Translation



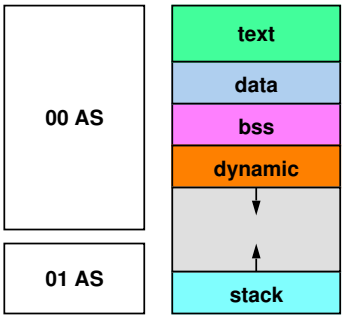
- PTEA: Page Table Entry Address

Linear Page Table Management

- 00 and 01 page tables each require contiguous locations in 10 space
  - with 512-byte pages, 8MB (= 4bytes $\times 2^{21}$) each:
    - memory cost was $40,000 per MB for the VAX
    - maximum of 64 such page tables (to fill up 500MB of physical memory)
    - (need room for other things, e.g. OS)
- Reduce size requirements with partial page tables
  - *length register* constrains size of each space

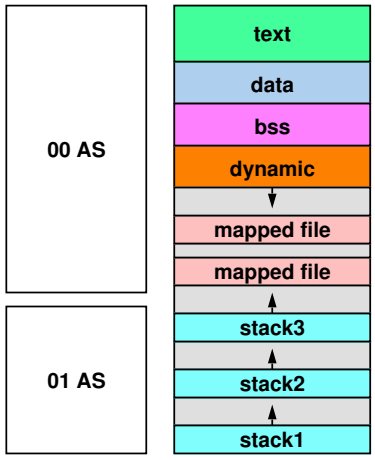
Traditional Unix with Linear PTs

- With traditional Unix, using a length register worked pretty well



Modern Unix

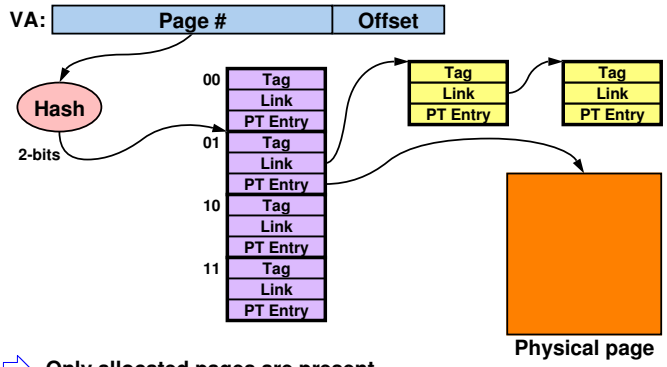
- Not so well with modern Unix



7.2 Hardware Support for Virtual Memory

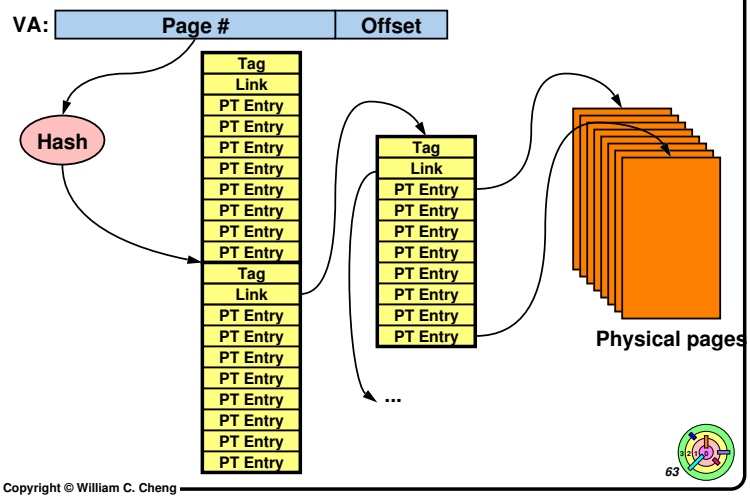
- Forward-Mapped Page Tables
- Linear Page Tables
- *Hashes Page Tables*
- Translation Lookaside Buffers
- 64-Bit Issues
- Virtualization

Hashed Page Tables

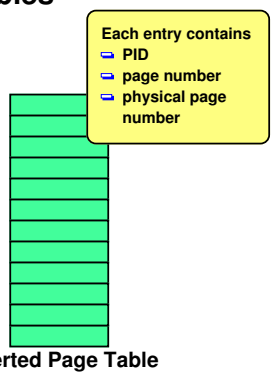


- Only allocated pages are present
  - works well for *sparcely allocated address space*
  - too much overhead (tag and link) when there are large regions of allocated memory

Clustered Page Tables

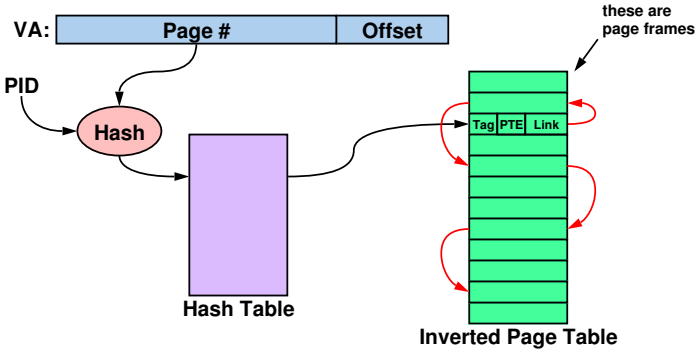


Inverted Page Tables



- Page table is *indexed* by *physical page number*
  - number of entries in IPT tends to be limited and small
  - how do you map page number (i.e., VPN) to physical page number?

Inverted Page Tables



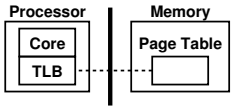
- Page table is *indexed* by *physical page number*
  - number of entries in IPT tends to be limited and small
  - how do you map page number (i.e., VPN) to physical page number?

7.2 Hardware Support for Virtual Memory

- Forward-Mapped Page Tables
- Linear Page Tables
- Hashes Page Tables
- *Translation Lookaside Buffers*
- 64-Bit Issues
- Virtualization

Translation Lookaside Buffers (TLB)

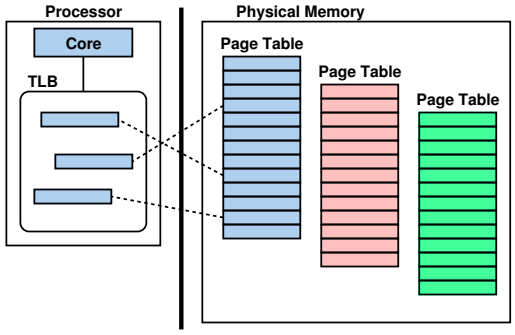
- Table lookup requires one memory access
  - to access memory starting with a virtual address will take at least *two* memory accesses
    - that's one access too many



- If the processor has additional memory
  - it can be used to *cache page table entries*
  - *Translation Lookaside Buffer (or TLB)*
    - the TLB caches the *mapping* from virtual page number to physical page number (along with other information in the page table entry)
    - hopefully, resolving a virtual address into a physical address will take one TLB lookup and one addition, inside the processor
- TLB miss vs. page fault
  - penalty for a TLB miss is O(1) memory accesses
  - penalty for a page fault is trap into the kernel

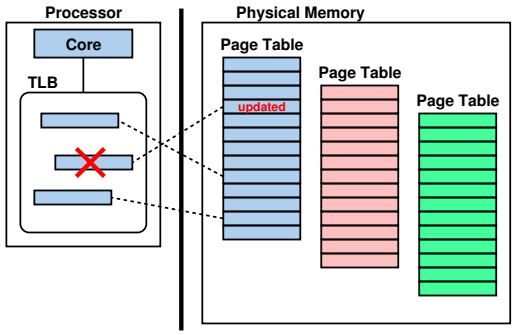
Translation Lookaside Buffers (TLB)

- Conceptually:



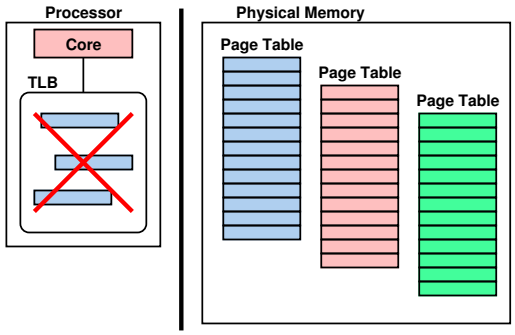
Translation Lookaside Buffers (TLB)

- When a page table entry is modified, the OS must *flush* (invalidate) the corresponding TLB entry

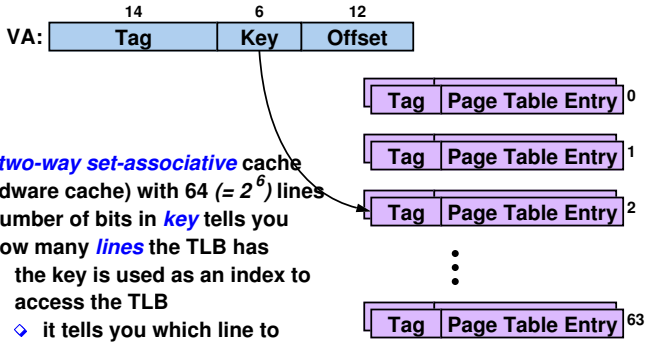


Translation Lookaside Buffers (TLB)

- When a page table entry is modified, the OS must *flush* (invalidate) the corresponding TLB entry
- When switching to a different *process*, must *flush* the *entire TLB*
  - in x86, this can be achieved by setting the CR3 register

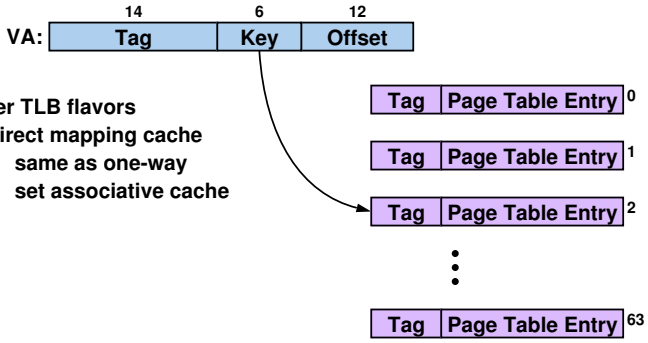


Translation Lookaside Buffers (TLB)



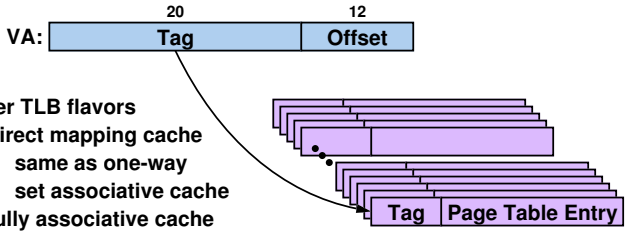
- Ex: *two-way set-associative* cache (hardware cache) with 64 *(= 2^6)* lines
  - number of bits in *key* tells you how many *lines* the TLB has
    - the key is used as an index to access the TLB
      - it tells you which line to look at
  - amount of *set-associativity* is the "array size" in each line
  - the *"tag"* in the virtual address is compared against *all* tags in a line simultaneously (i.e., under the same key)

Translation Lookaside Buffers (TLB)



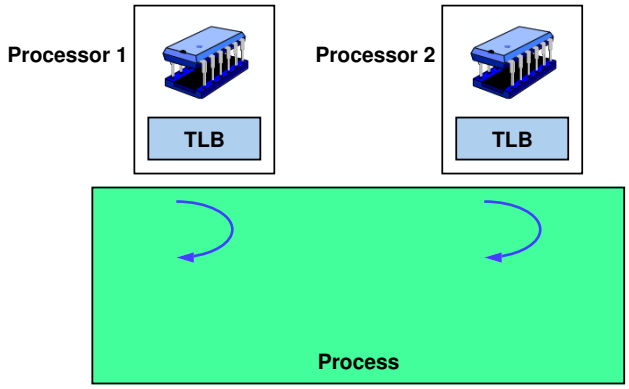
- Other TLB flavors
  - direct mapping cache
    - same as one-way set associative cache

Translation Lookaside Buffers (TLB)



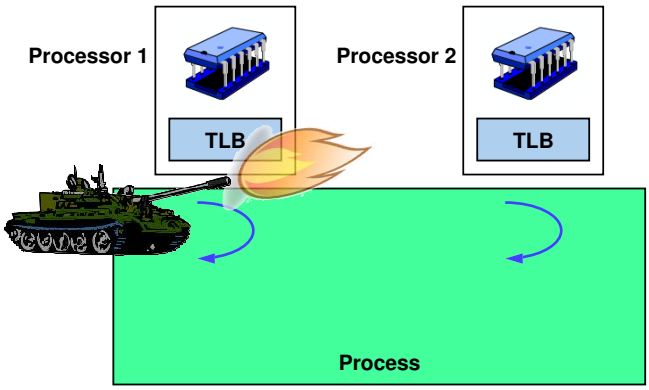
- Other TLB flavors
  - direct mapping cache
    - same as one-way set associative cache
  - fully associative cache
    - same as single-entry set associative cache
    - expensive
      - need to compare with all the tags *in parallel*

TLBs and Multiprocessors



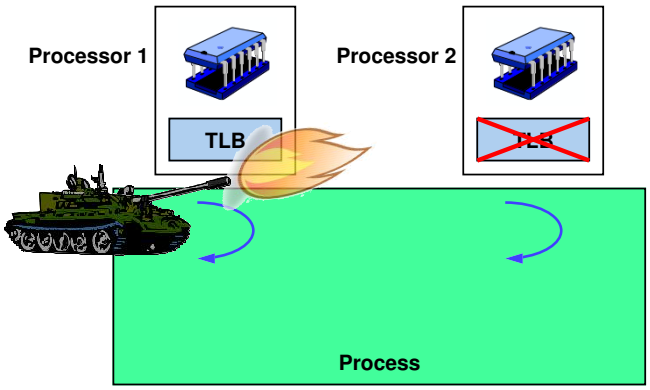
- In a multiprocessors environment, one processor can modify a mapping cached in the TLB of another processor

TLBs and Multiprocessors



- Before such a mapping is modified, Processor 1 must *shoot-down* (invalidate) the TLB of Processor 2

TLBs and Multiprocessors



- Need to get the timing right

TLB Shootdown Algorithm (In Hardware / HAL)

```
// shooter code
for all processors i sharing address space
   interrupt(i);
for all processors i sharing address space
   while (noted[i] == 0)
      ;
modify_page_table();
update_or_flush_tlb();
done[me] = 1;

// shootee i interrupt handler
receive_interrupt_from_processor j
noted[i] = 1
while (done[j] == 0)
   ;
flush_tlb()
```

Storage / Cache Hierarchy in Today's Systems

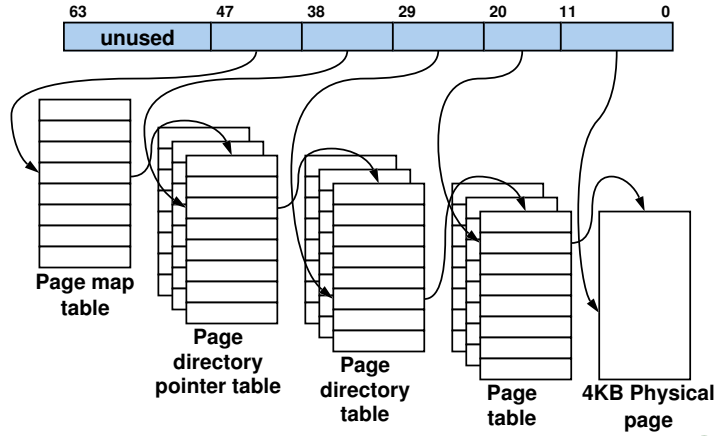
Storage	Access Time	Size
TLB	1 ns	64 KB
L2	4 ns	256 KB
L3	10 ns	2 MB
RAM	100 ns	10 GB
SSD	100 μs	100 GB
Remote RAM	100 μs	100 GB
Disk	1 - 10 ms	1 TB
Remote Disk	100 ms	1 XB
...	...	...

- There can even be something before TLB
  - "Virtually Addressed Cache"
- L2 and L3 are "Physically Addressed Caches"
- TLB, L2, L3 are managed in hardware
  - OS can only invalidate entries in a cache

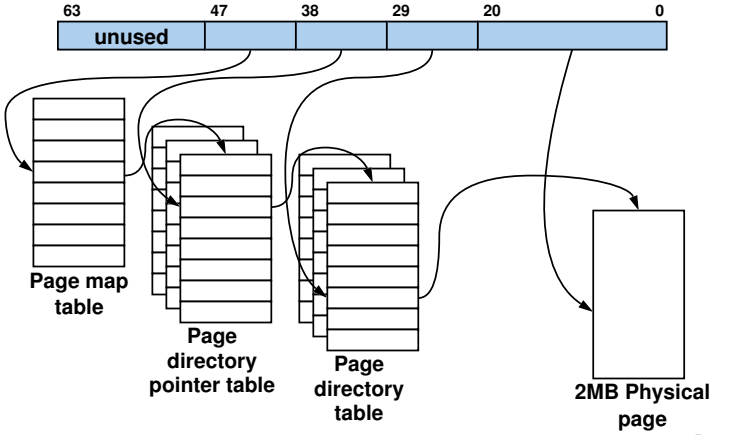
7.2 Hardware Support for Virtual Memory

- Forward-Mapped Page Tables
- Linear Page Tables
- Hashes Page Tables
- Translation Lookaside Buffers
- *64-Bit Issues*
- Virtualization

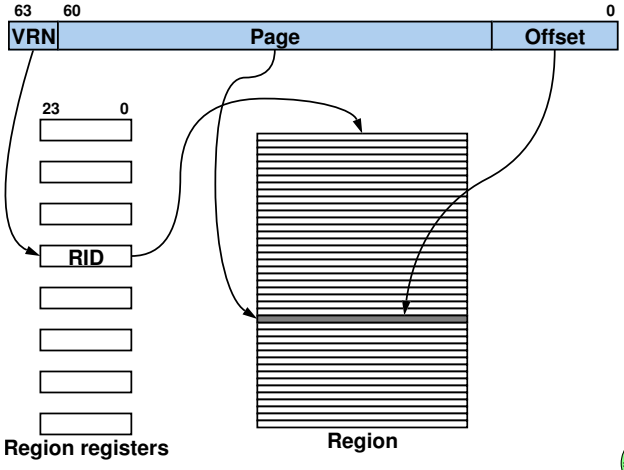
x86-64 (AMD) Virtual Address Format 1



x86-64 (AMD) Virtual Address Format 2



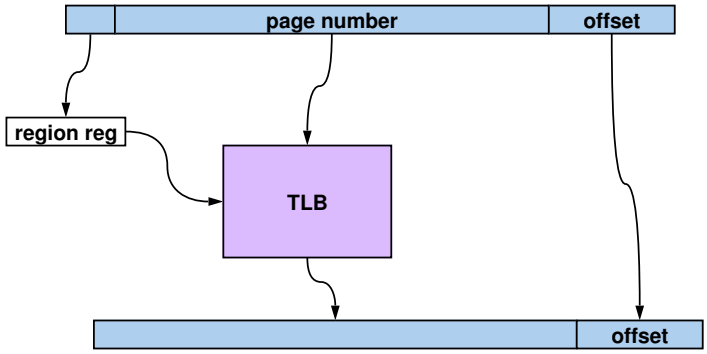
Intel IA-64



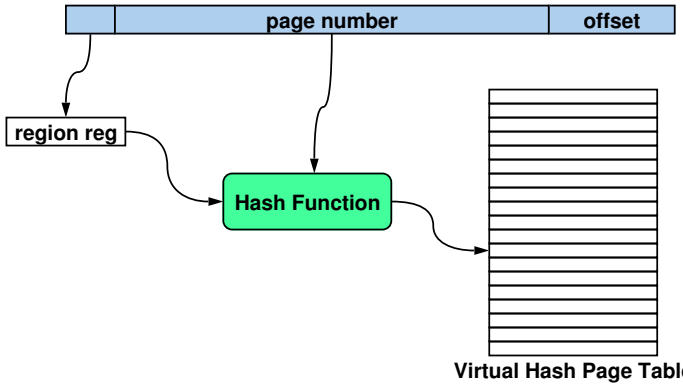
IA-64 Address Translation

- TLB
  - software-managed
- Virtual Hash Page Table (VHPT)
  - per-region linear page table
  - single large hashed page table

Translation: TLB



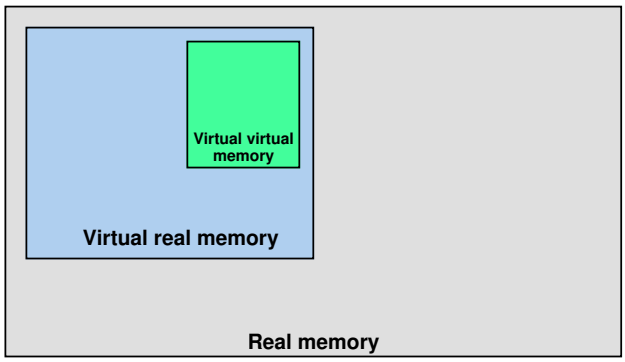
Translation: TLB Miss



7.2 Hardware Support for Virtual Memory

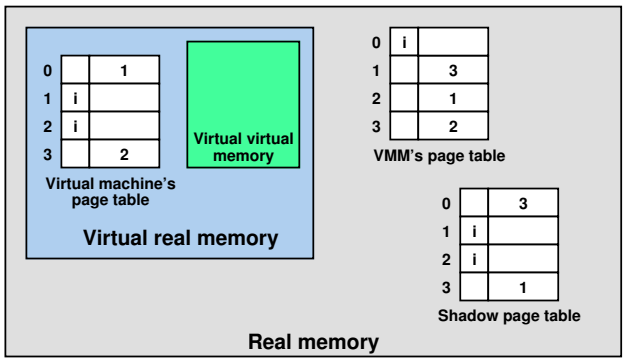
- Forward-Mapped Page Tables
- Linear Page Tables
- Hashes Page Tables
- Translation Lookaside Buffers
- 64-Bit Issues
- *Virtualization*

Virtual Machines Meet Virtual Memory



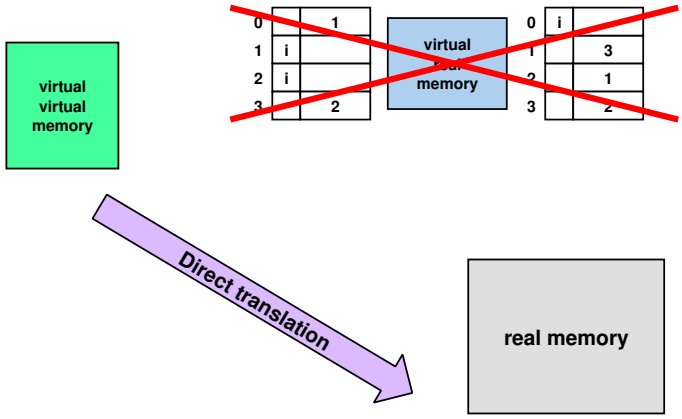
- How can we virtualize virtual memory?

Virtual Machines Meet Virtual Memory

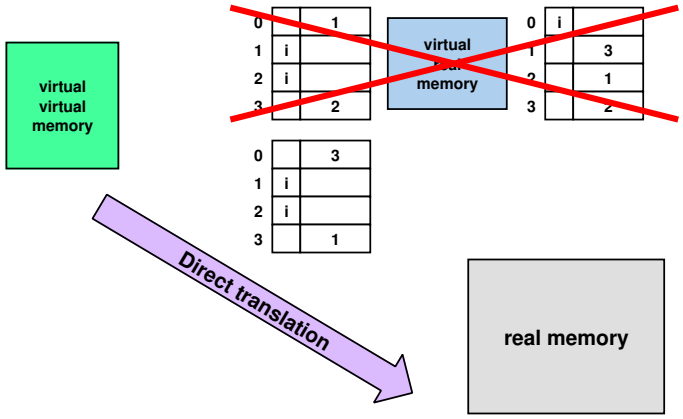


- When a VM changes its page table, VMM must update the corresponding *Shadow Page Table*
  - main problem: poor performance

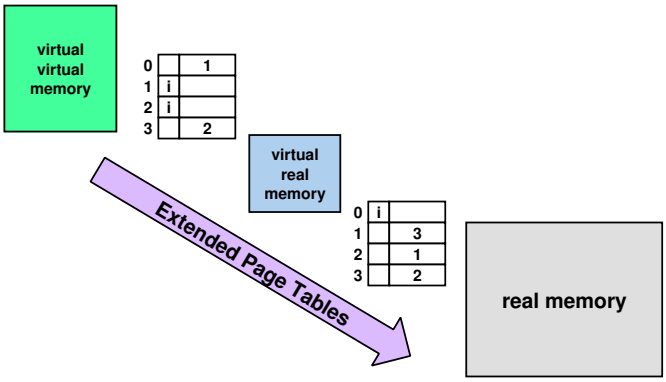
Solution 1: Paravirtualization to the Rescue



Solution 1: Paravirtualization to the Rescue



Solution 2: Hardware to the Rescue



x86 Paging with EPT