Great Ideas in Computer Architecture

Virtual Memory

Instructor: Jenny Song

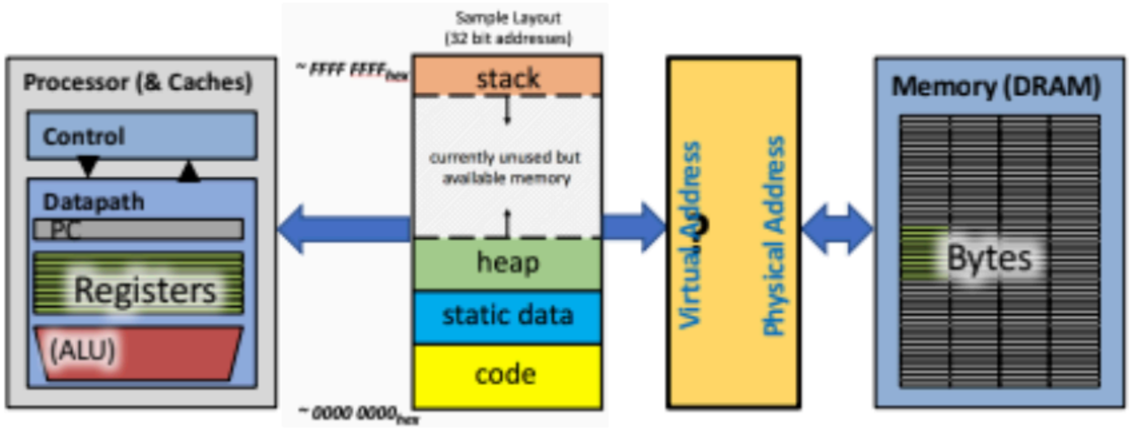
Review: Virtual Memory

- Next level in the memory hierarchy:
  - Provides program with illusion of a very large main memory:
  - Working set of "pages" reside in main memory - others reside on disk.
- Also allows OS to share memory, protect programs from each other
- Today, more important for protections. just another level of memory hierarchy
- Each process thinks it has all the memory to itself
- (Historically, it predates caches)

Review: Address Space

- Address space = set of addresses for all available memory locations
- Now, two kinds of memory addresses:
  - **Virtual Address Space**
    - Set of addresses that the user program knows about
  - **Physical Address Space**
    - Set of addresses that map to actual physical locations in memory
    - Hidden from user applications
- Memory manager maps between these two address spaces

Review: Virtual vs Physical Addresses

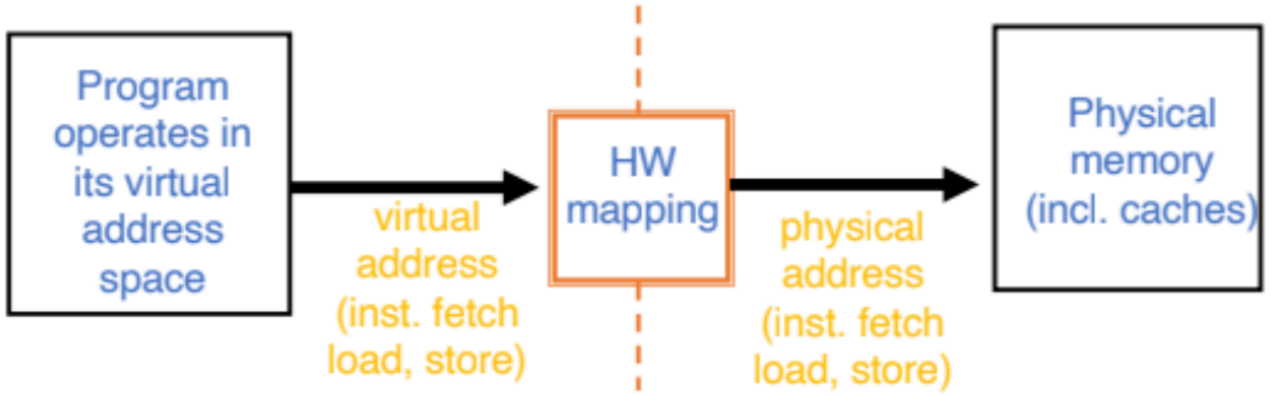


Many of these (software & hardware cores)

One main memory

- Processes use virtual addresses, e.g., 0 … 0xffff,ffff
  - Many processes, all using same (conflicting) addresses
- Memory uses physical addresses (also, e.g., 0 … 0xffff,ffff)
  - ***Memory manager maps virtual to physical addresses***

Review: Virtual to Physical Address Translation


Program operates in its virtual address space
virtual address (inst. fetch load, store)
HW mapping
physical address (inst. fetch load, store)
Physical memory (incl. caches)

- Each program operates in its own virtual address space; ~only program running
- Each is protected from the other
- OS can decide where each goes in memory
- Hardware gives virtual -> physical mapping

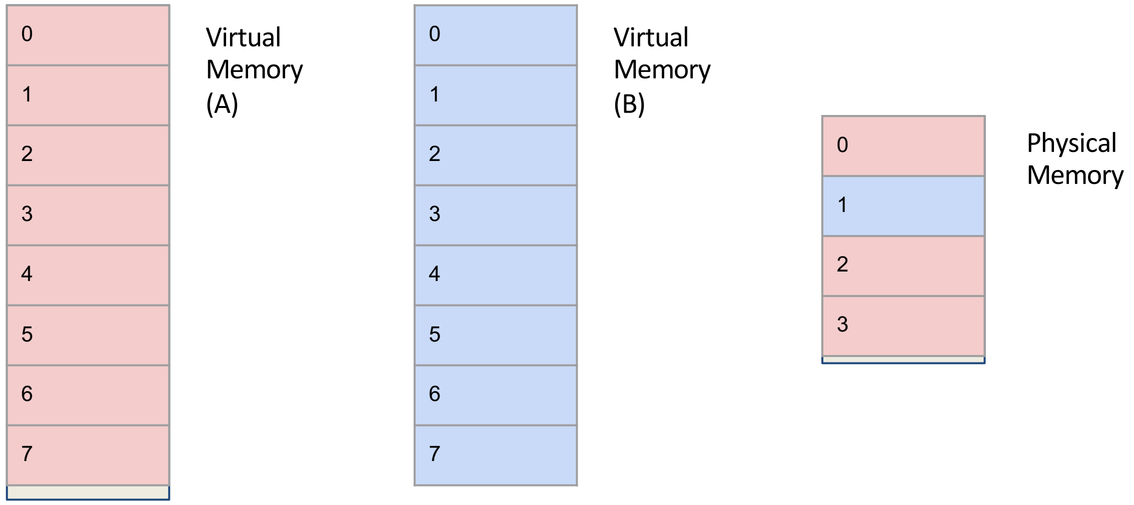
Virtual Memory and Physical Memory

- Programs use *virtual addresses (VAs)*
  - Space of all virtual addresses called *virtual memory (VM)*
  - Divided into pages indexed by *virtual page number (VPN)*

- Main memory indexed by *physical addresses (PAs)*
  - Space of all physical addresses called *physical memory (PM)*
  - Divided into pages indexed by *physical page number (PPN)*

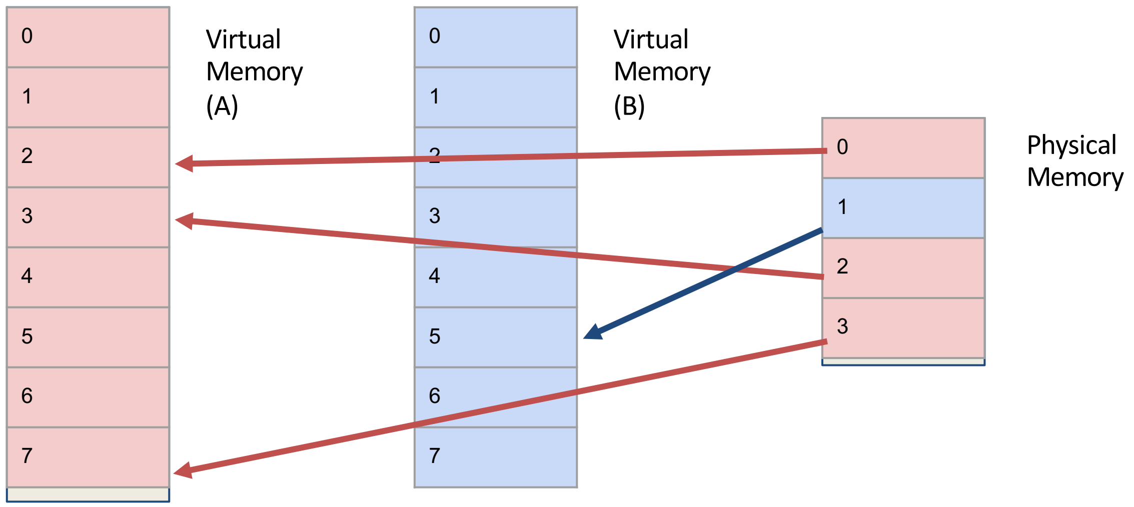
Review: What is Paged VM?

- Each process has its own virtual memory, but all processes must *share* physical memory



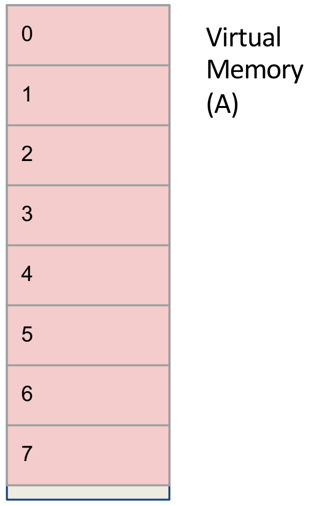
Review: What is Paged VM?

- Pages in physical memory correspond to pages in virtual memory

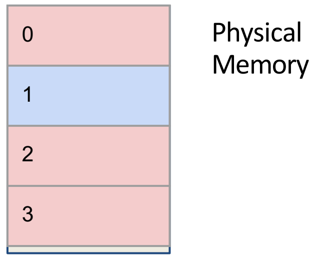


Review: What is Paged VM?

- Each process has a table mapping its physical/virtual pages

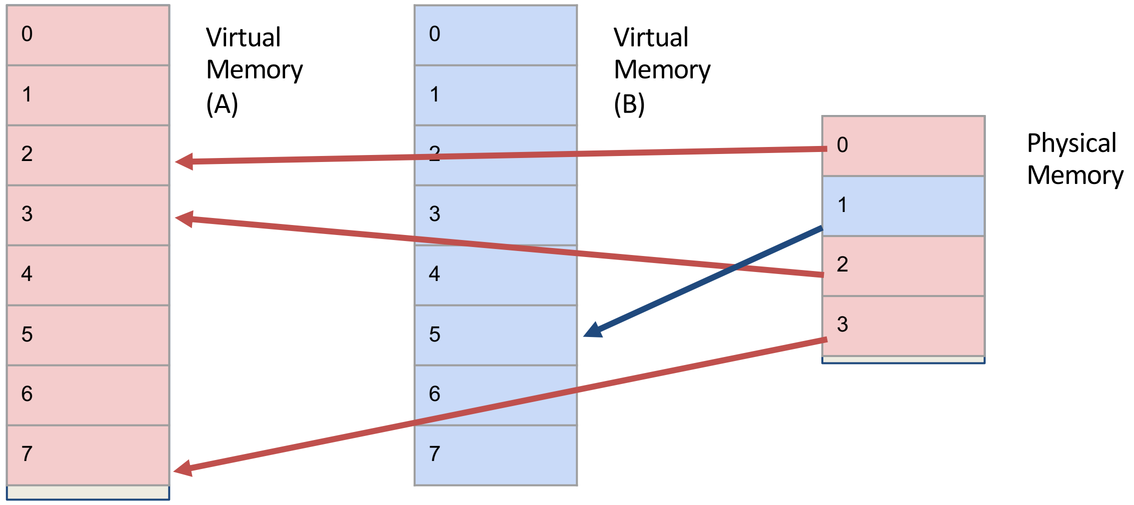


VPN	PPN	Valid
0	X	0
1	X	0
2	0	1
3	2	1
4	X	0
5	X	0
6	X	0
7	3	1



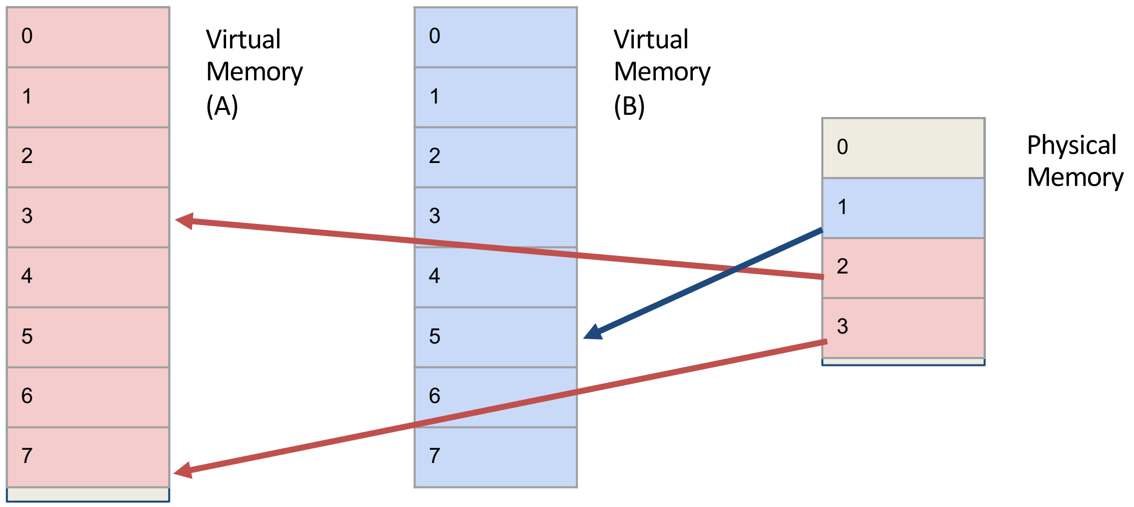
Review: What is Paged VM?

When a process needs more data, the oldest (LRU) page is removed from PM and replaced with a new page from disk



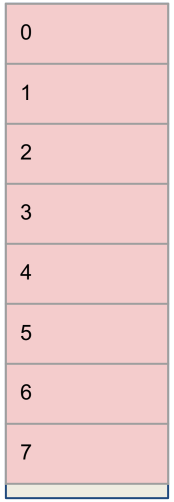
Review: What is Paged VM?

When a process needs more data, its oldest (LRU) page is removed from PM and replaced with a new page from disk



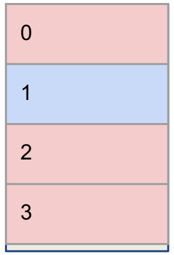
Review: What is Paged VM?

- Each process has a table mapping its physical/virtual pages



Virtual Memory (A)

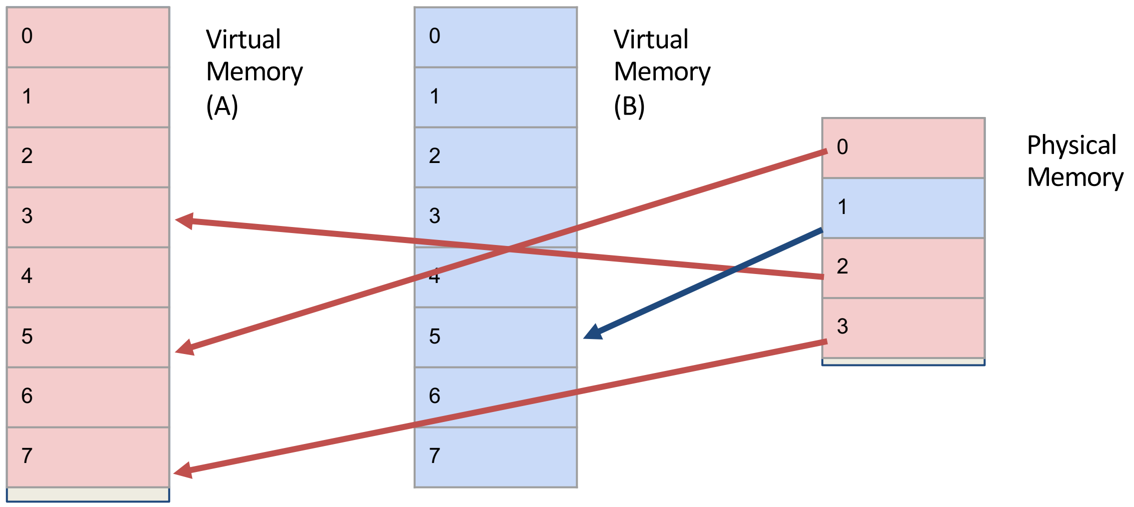
VPN	PPN	Valid
0	X	0
1	X	0
2	0	0
3	2	1
4	X	0
5	X	0
6	X	0
7	3	1



Physical Memory

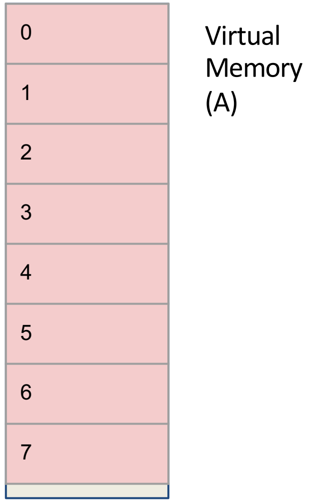
Review: What is Paged VM?

When a process needs more data, its oldest (LRU) page is removed from PM and replaced with a new page from disk

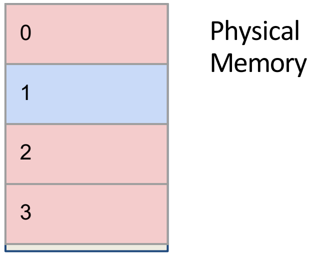


Review: What is Paged VM?

- Each process has a table mapping its physical/virtual pages



VPN	PPN	Valid
0	X	0
1	X	0
2	0	0
3	2	1
4	X	0
5	0	1
6	X	0
7	3	1



Agenda

- Virtual Memory and Page Tables
- Translation Lookaside Buffer (TLB)
- VM Performance
- VM Wrap-up

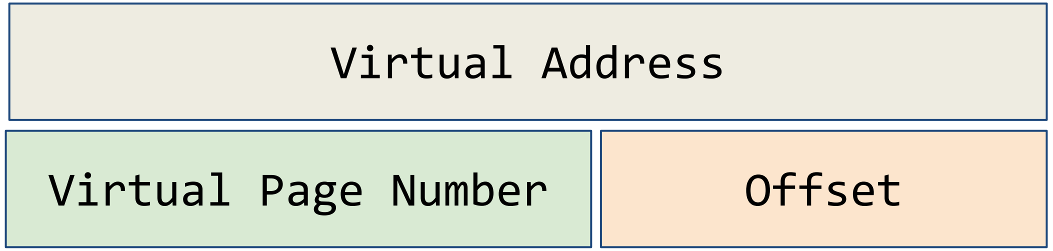
Address Translation

Given a request for an address (virtual) we have to locate the data in physical memory.

<u>The process of converting a virtual address to a physical address is called address translation!</u>

Address Translation

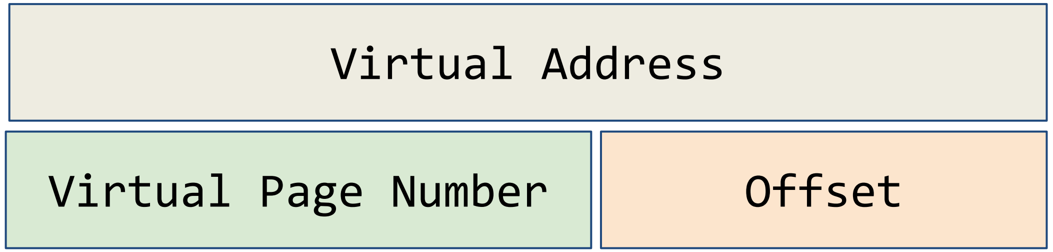
Virtual addresses can be broken into two parts (much like we broke them down for T/I/O): we'll call the two parts the page number and the offset:



Address Translation

We can calculate the size of these fields like we did with caches:

- sizeof(Address) = log_2(Virtual Memory Size)
- sizeof(offset) = log_2(page size)
- sizeof(VPN) = log_2(# of virtual pages)
    - = address size - offset size



Locating a Virtual Byte

256B Virtual Memory

32B Page

Offset: 5 bits

VPN: log_2(256/32) = 3 bits

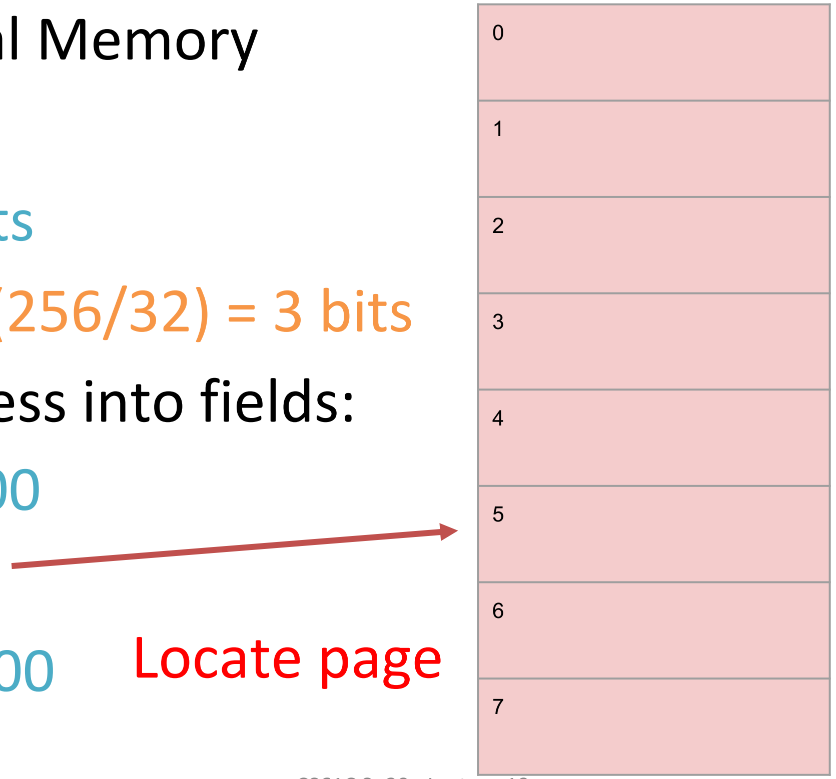
Break address into fields:

0b101 00100

VPN: 101

Offset: 00100

Locate page



Virtual
Memory
(A)

VM = 256 B
Page = 32 B

Locating a Virtual Byte

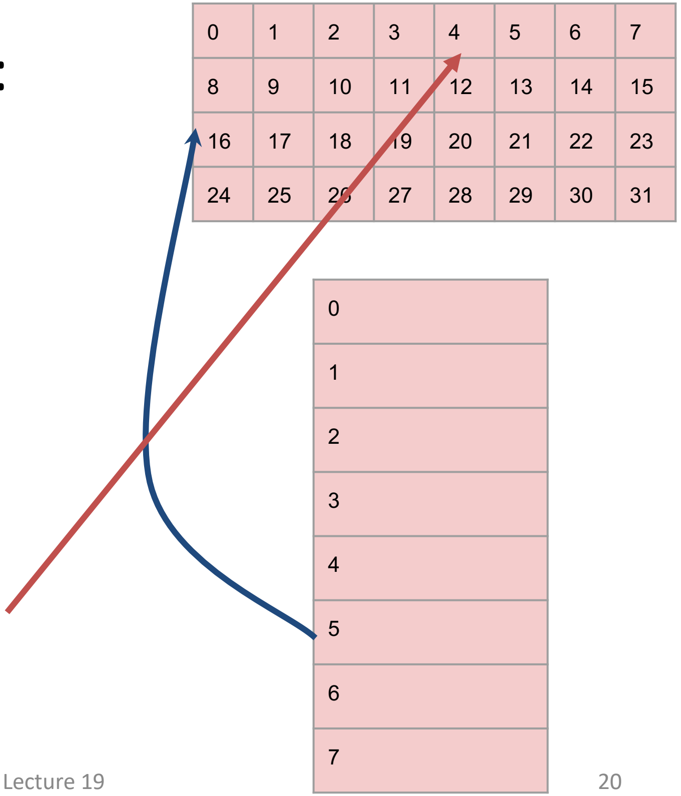
Break address into fields:

0b101 00100

VPN: 101

Offset: 00100

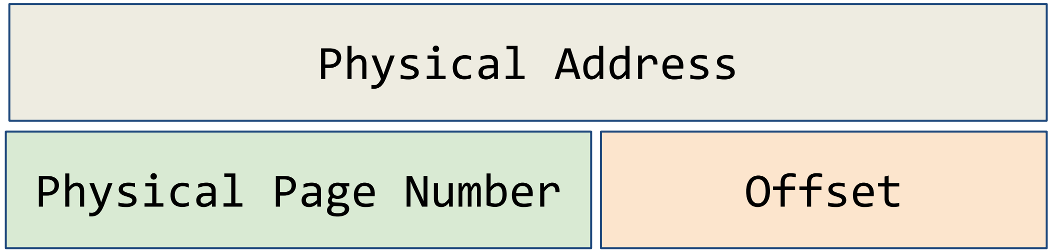
Locate byte within page:



Address Translation

We can calculate the size of these fields like we did with caches:

- sizeof(Address) = log_2(Physical Memory Size)
- sizeof(offset) = log_2(page size)
- sizeof(PPN) = log_2(# of Physical pages)
  - = address size - offset size



Locating a Physical Byte

Because we've also divided physical memory into pages, the same process applies here!

Given a physical address:

- Break the address into page number (PPN) and offset
- Locate the page
- Locate the byte within the page

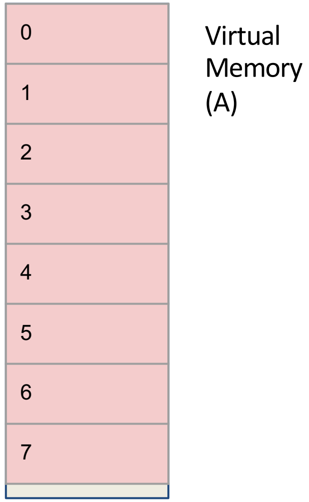
But wait...

We still haven't solved our original problem: given a virtual address I can find a virtual byte, given a physical address I can find a physical byte, but:

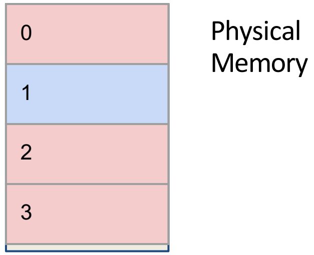
- How do I go from virtual address to physical address?

Review: What is Paged VM?

- Each process has a table mapping its physical/virtual pages

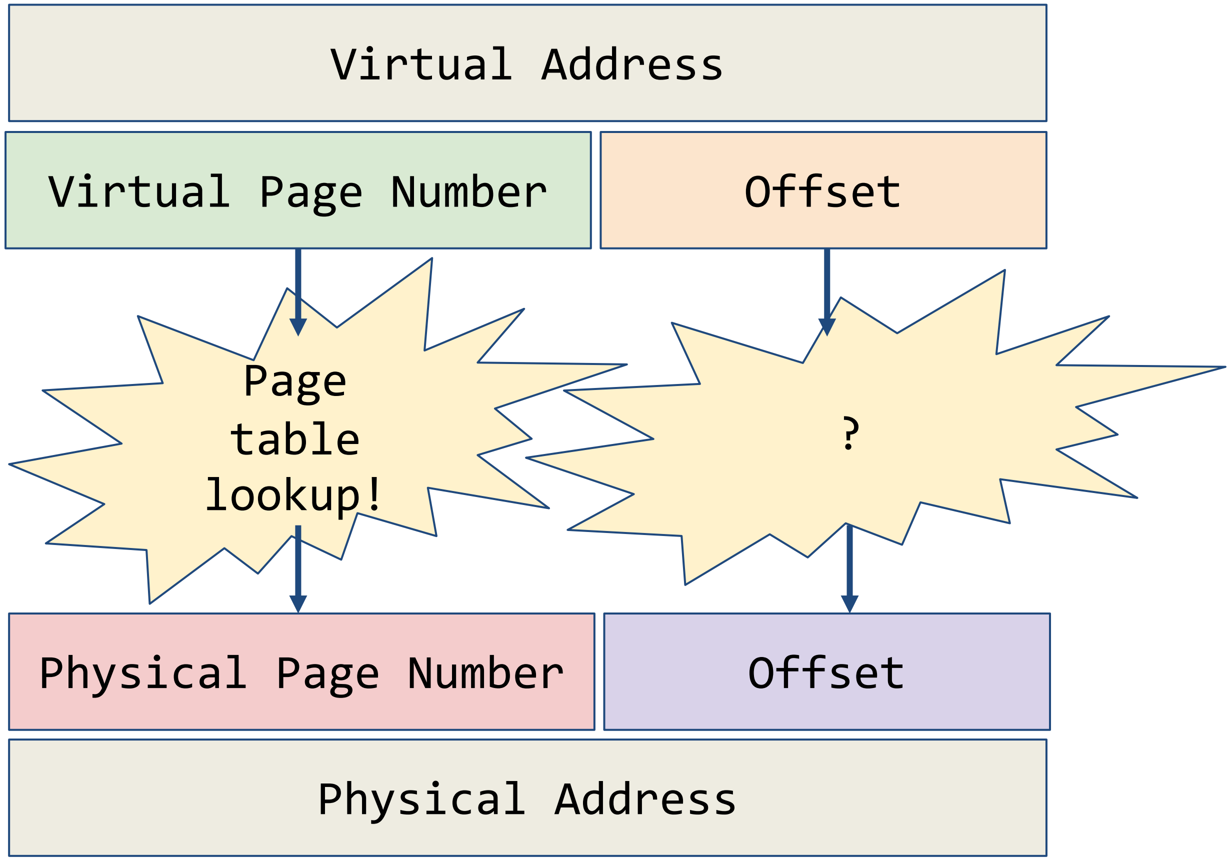


VPN	PPN	Valid
0	X	0
1	X	0
2	0	1
3	2	1
4	X	0
5	X	0
6	X	0
7	3	1

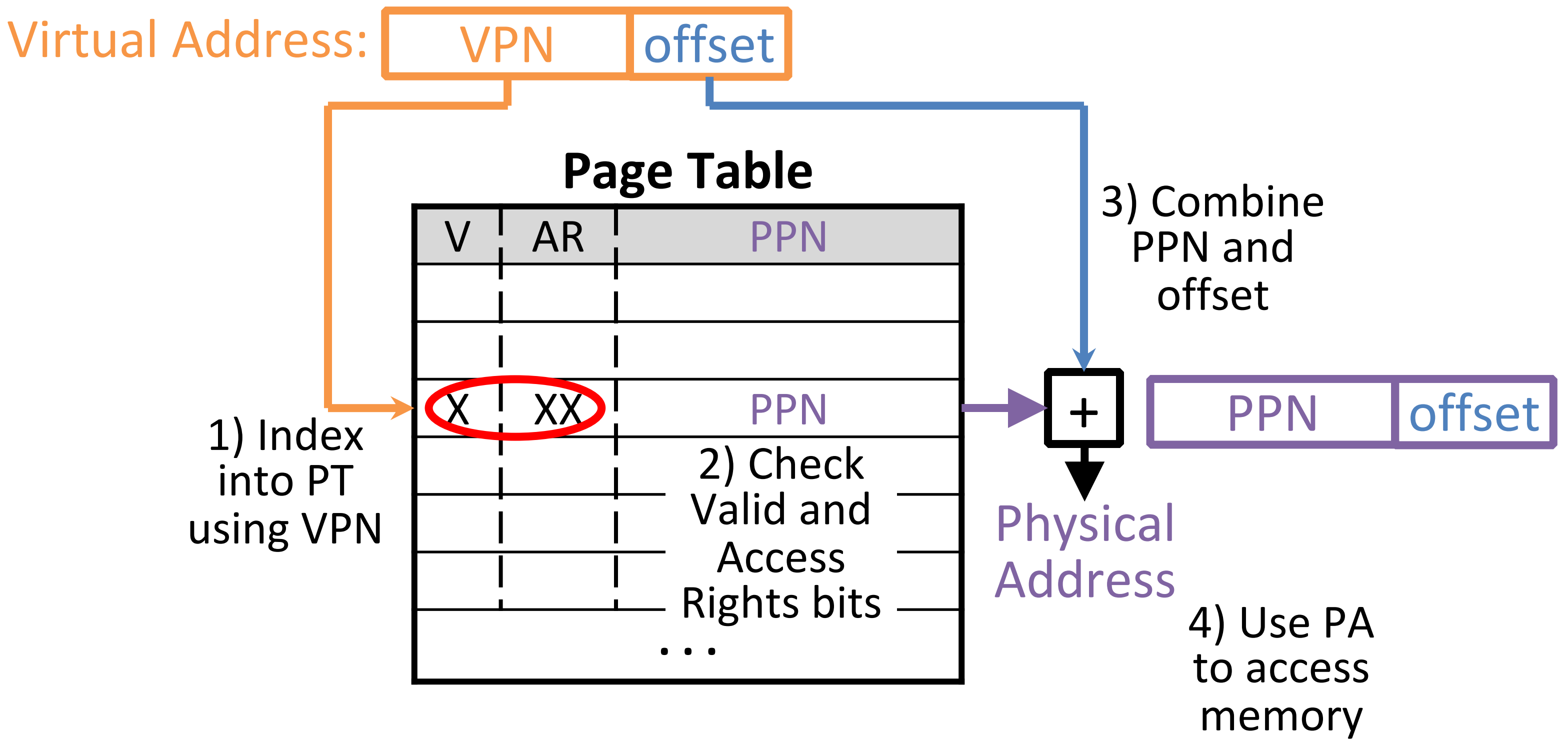


Page Table Entry Format

- Contains either PPN or indication not in main memory
- Dirty = Page has been modified recently
- Valid = Valid page table entry
  - 1 → virtual page is in physical memory
  - 0 → OS needs to fetch page from disk Page Fault!
- Access Rights checked on every access to see if allowed (provides protection)
  - *Read, Write, Executable:* Can fetch instructions from page
  - Protection Fault!


Virtual Address
Virtual Page Number
Offset
Page table lookup!
?
Physical Page Number
Offset
Physical Address

Page Table Layout



Virtual Address:
VPN
offset
Page Table
V
AR
PPN
X
XX
PPN
1) Index into PT using VPN
2) Check Valid and Access Rights bits
. . .
3) Combine PPN and offset
+
PPN
offset
Physical Address
4) Use PA to access memory

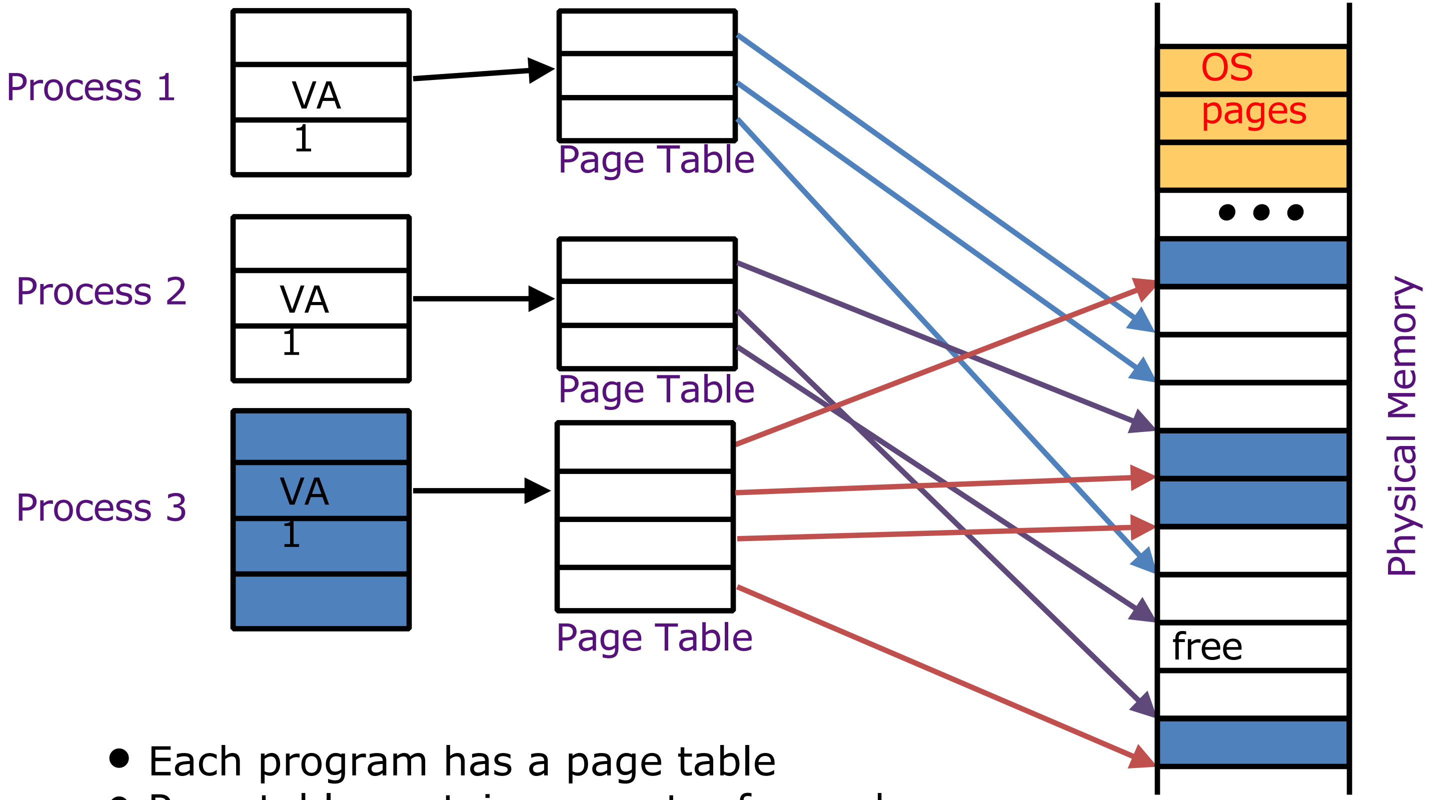
Protection Between Processes

- With a bare system, addresses issued with loads/stores are real physical addresses
- This means any program can issue any address, therefore can access any part of memory, even areas which it doesn't own
  - Example: the OS data structures
- How does having a page table isolate and protect processes?

Protection Between Processes

- In order to create a physical address from a virtual one, we have to convert our page number (VPN -> PPN)
  - To do this, we look up a mapping for the VPN in our page table!
- We can only “find” mappings for pages we own!
  - All other mappings are invalid or blank!
- Therefore, we cannot construct physical addresses we do not have access to :)

Private Address Space per User



- Each program has a page table
- Page table contains an entry for each program page

Agenda

- Virtual Memory and Page Tables
  - Hierarchical Page Table
- Translation Lookaside Buffer (TLB)
- VM Performance
- VM Wrap-up

How Big is the Page Table?

- 64 MiB RAM
- 32-bit virtual address space
- 1 KiB pages

Offset Bits = $\log_2(1024) = 10$

# Virtual Page Bits = 32 - 10 = 22
(2^{22} entries in the page table!)

# Physical Page Bits = $\log_2(2^{26}) - 10$

# Physical Page Bits = 26 - 10 = 16 (2 B)

Total Bytes $\approx 2^{22} * 2 \approx 2^{23}$ B

Number of pages = $2^{23} / 2^{10} = 2^{13}$ PAGES

(A) Less than 1 Page

(B) Less than 100 Pages

(C) Less than 1000 Pages

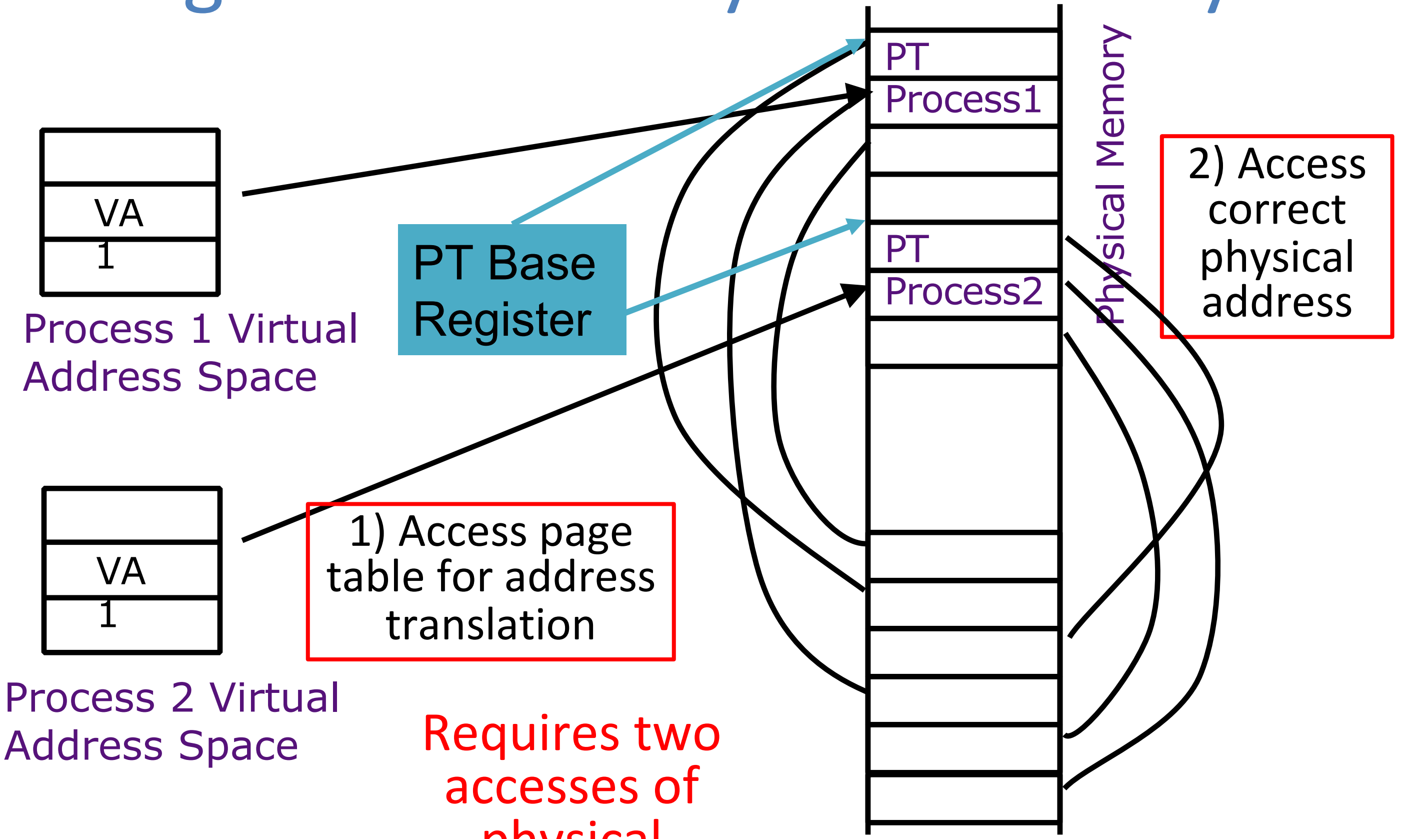
[✓ **(D)** More than 1000 Pages]

Where Should Page Tables Reside?

- Space required by the page tables (PT) is proportional to the address space, number of users, ...

  ⇒ *Too large to keep in registers, or caches….*
- Idea: Keep PTs in the main memory
  - How can we find the page table in memory if the page table is how we learn Physical addresses?????
  - PT Base Register: stores Physical Address of current Page Table

Page Tables in Physical Memory



VA
1
Process 1 Virtual Address Space
VA
1
Process 2 Virtual Address Space
PT Base Register
PT Process1
PT Process2
Physical Memory

1) Access page table for address translation

2) Access correct physical address

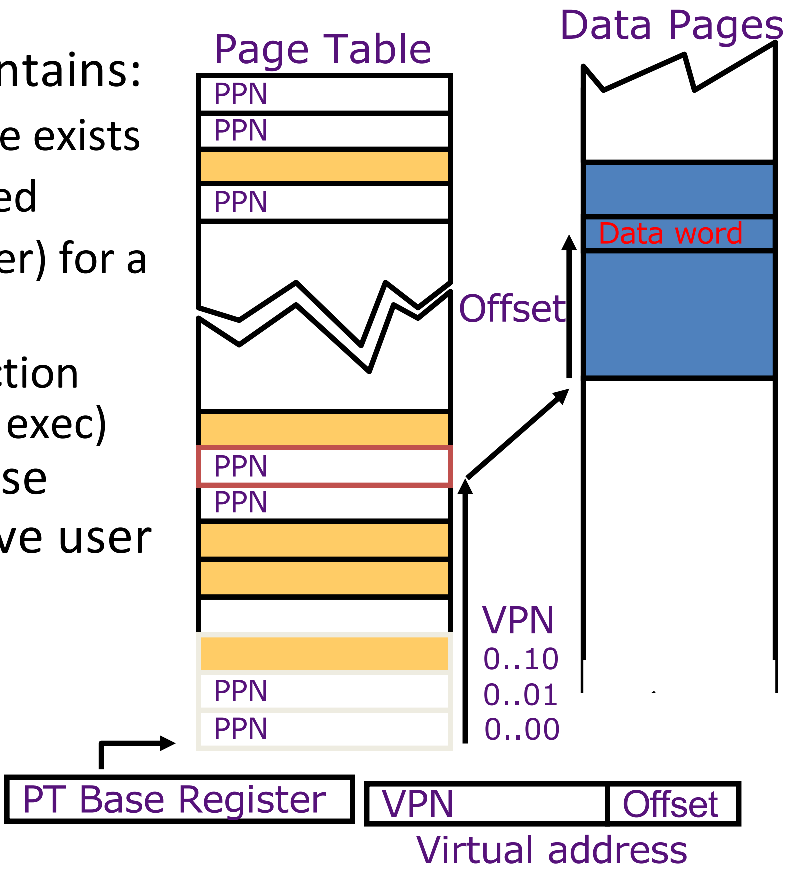
Requires two accesses of physical memory!

Linear Page Table

Page Table Entry (PTE) contains:

- Valid bit to indicate if page exists
- Dirty bit if page is modified
- PPN (physical page number) for a memory-resident page
- Permission bits for protection and usage (read, write, exec)

OS sets the Page Table Base Register whenever active user process changes



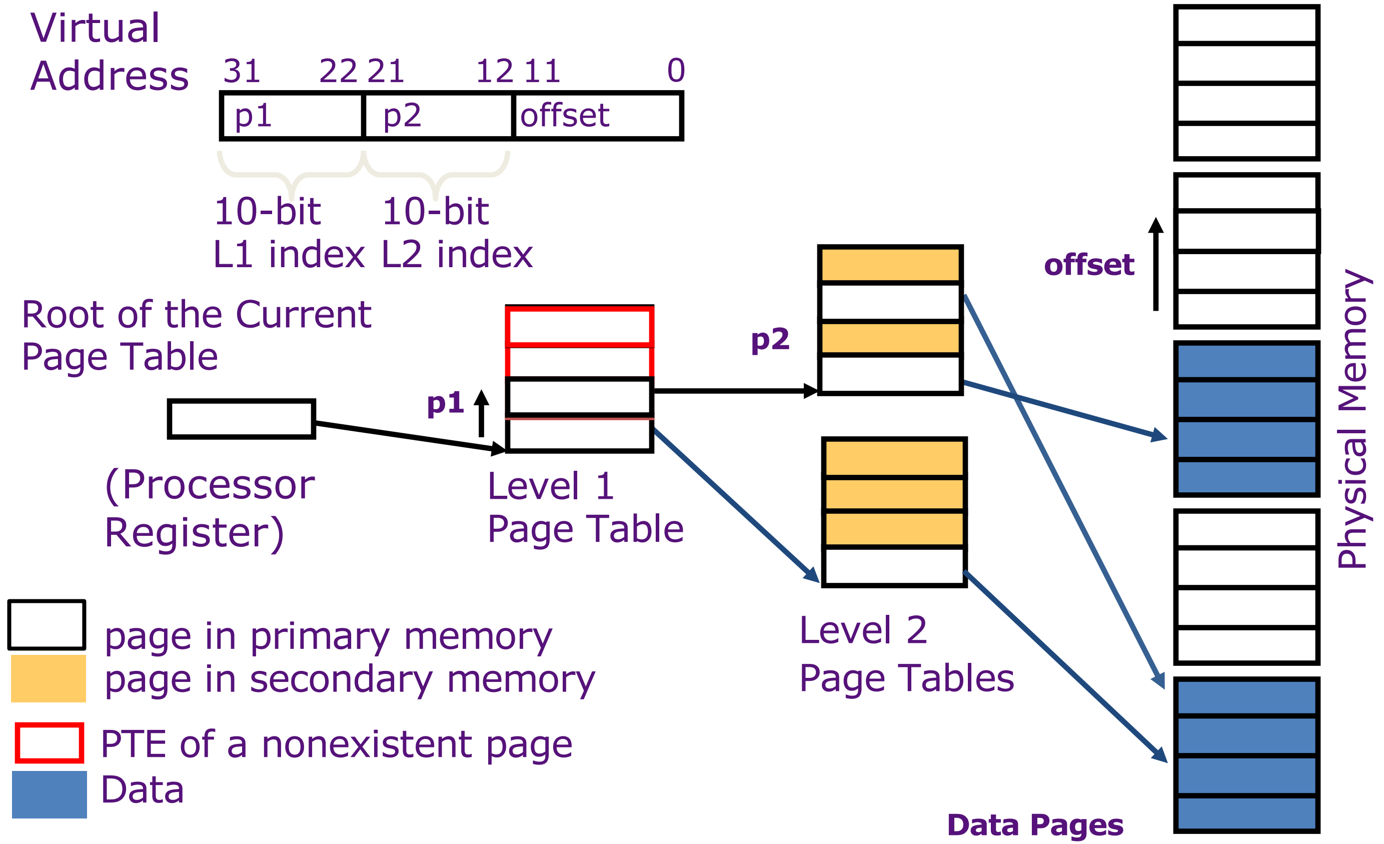
Linear Page Table: Problems

Linear page tables can be really large (span multiple pages), and sparsely populated!

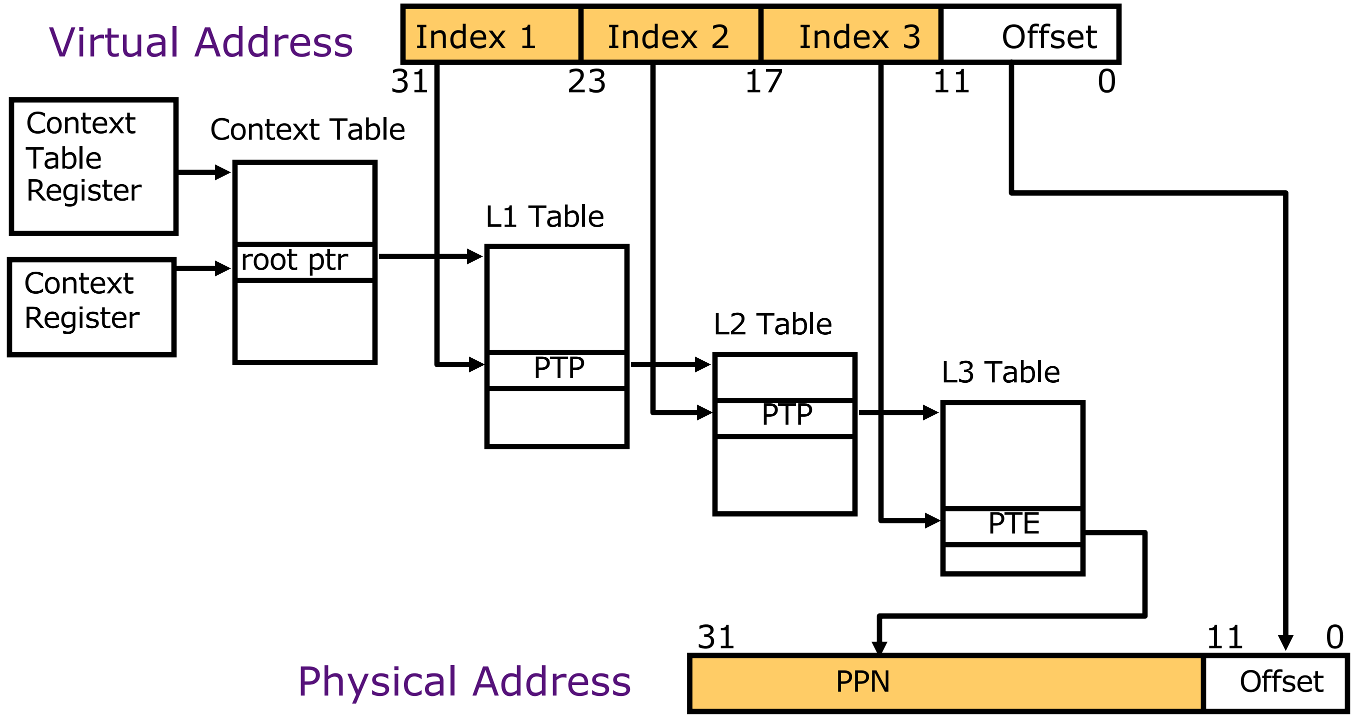
What if I only need the first page (0) and the last page (N) in my virtual memory space? I have to load the *entire page table!*

What if there was a way to only load/create the sections I need *as I need them*?

Hierarchical Page Table

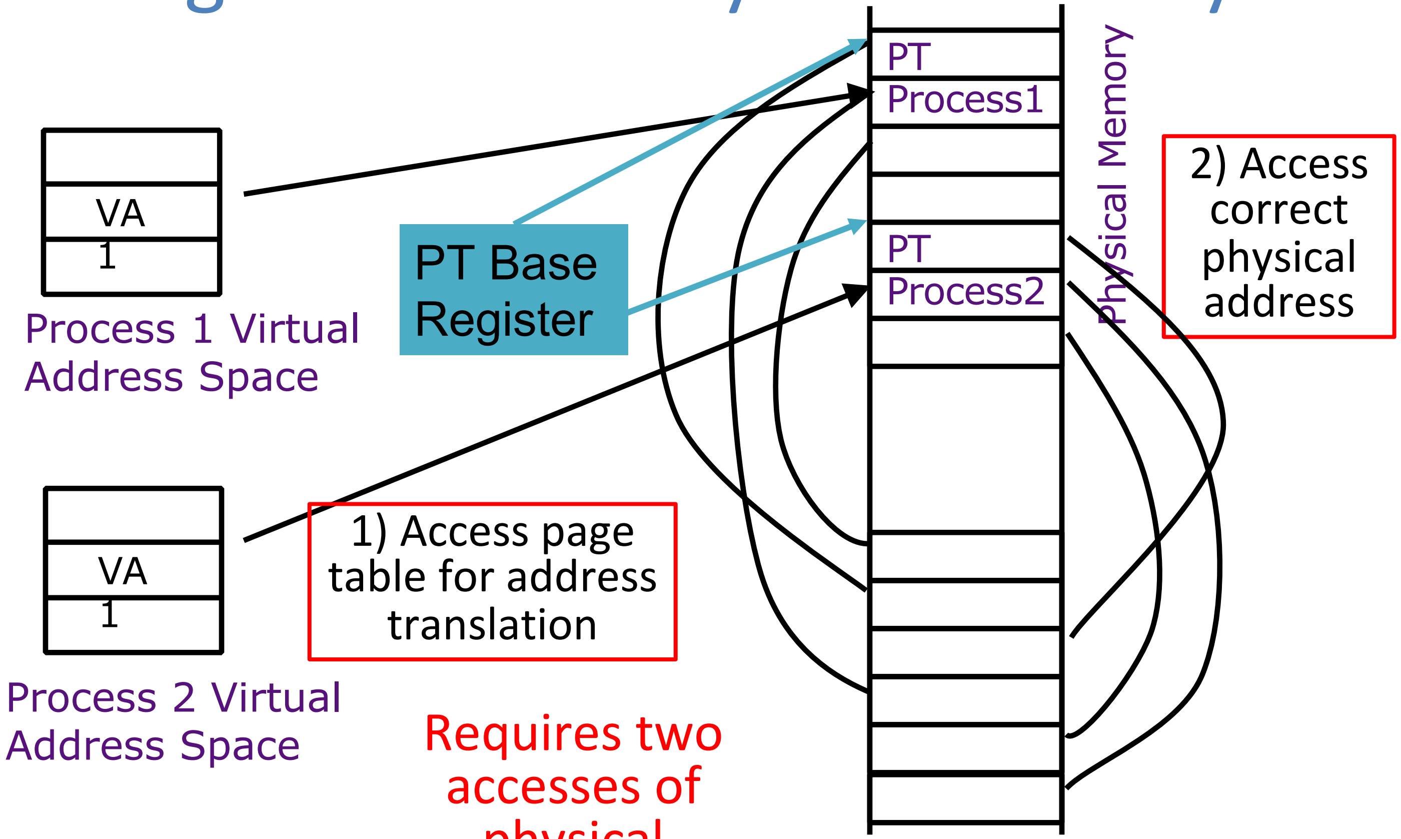


Hierarchical Page Table Walk: SPARC v8



MMU does this table walk in hardware on a TLB miss

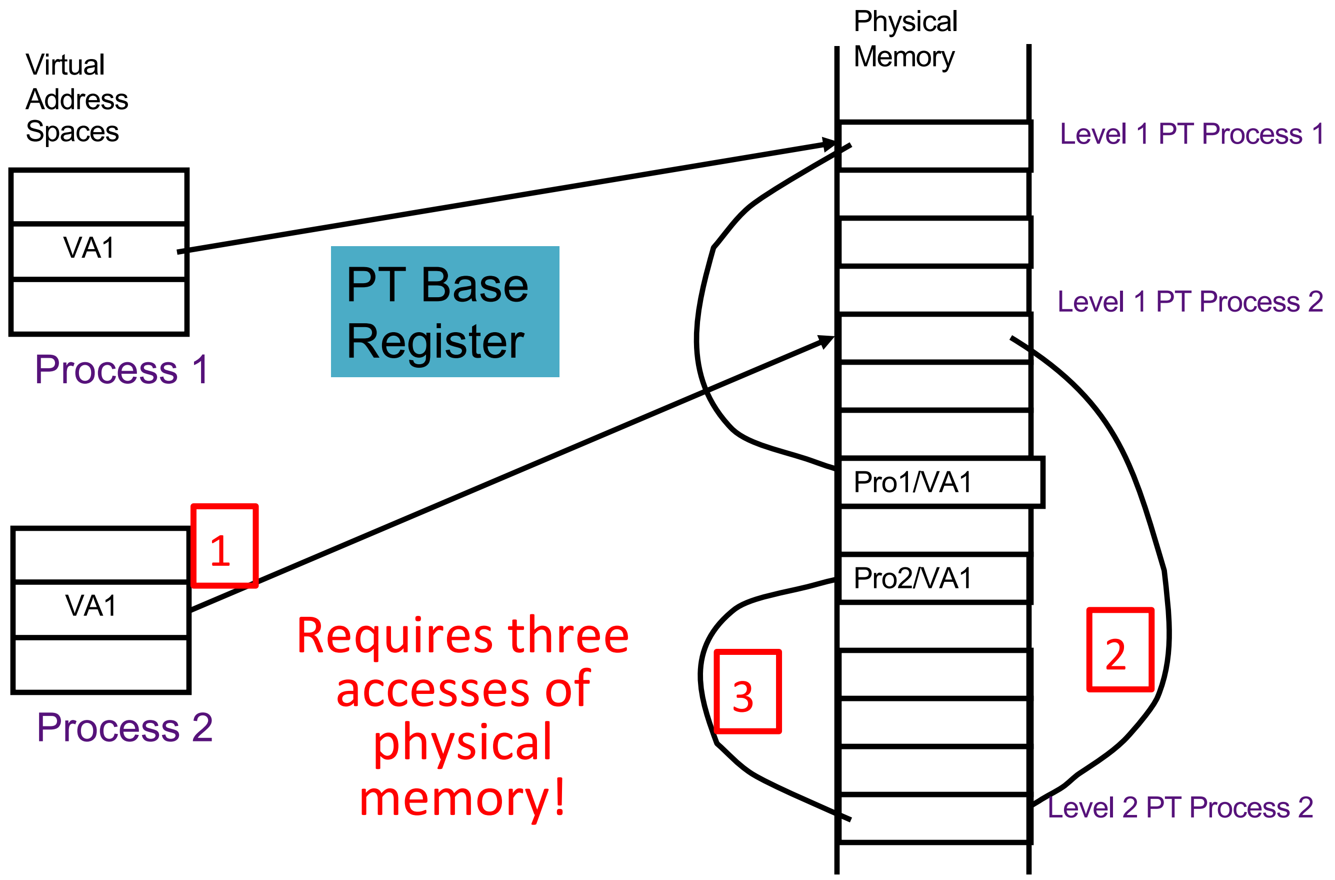
Page Tables in Physical Memory



PT
Process1
PT
Process2
Physical Memory
VA
1
Process 1 Virtual Address Space
VA
1
Process 2 Virtual Address Space
PT Base Register
2) Access correct physical address
1) Access page table for address translation

Requires two accesses of physical memory!

Two-Level Page Tables in Physical Memory



Requires three accesses of physical memory!

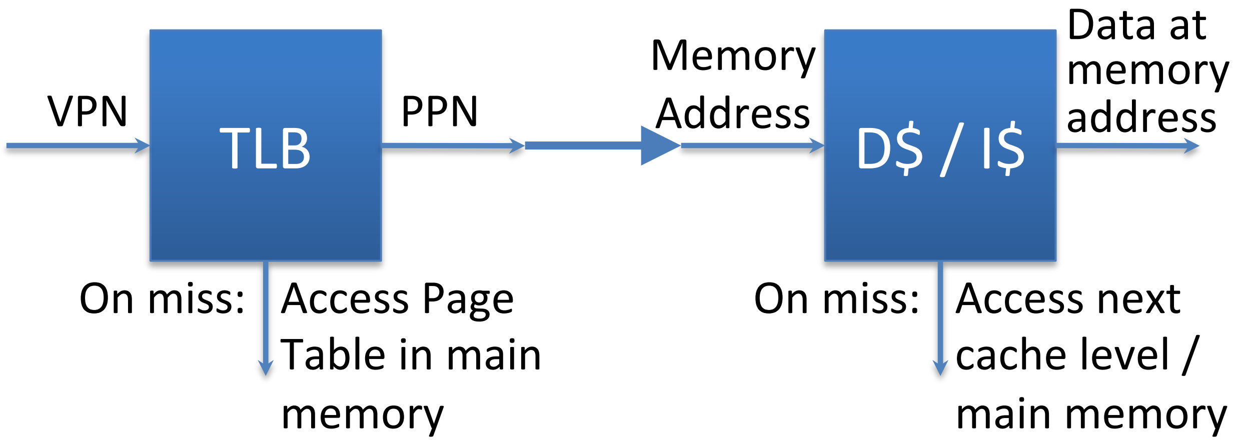
Agenda

- Virtual Memory
- Page Tables
- Translation Lookaside Buffer (TLB)
- VM Performance
- VM Wrap-up

Virtual Memory Problem

- 2+ physical memory accesses per data access = SLOW!
- Since locality in pages of data, there must be locality in the translations of those pages
- **Build a separate cache for the Page Table**
  - For historical reasons, cache is called a *Translation Lookaside Buffer (TLB)*
  - Notice that what is stored in the TLB is NOT memory data, but the VPN → PPN mappings

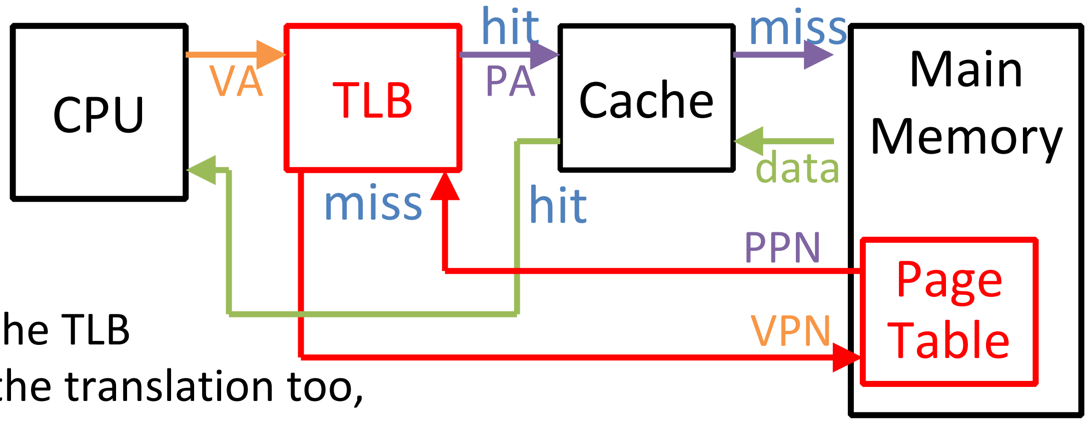
TLBs vs. Caches


VPN
TLB
PPN
Memory Address
D$ / I$
Data at memory address
On miss:
Access Page Table in main memory
On miss:
Access next cache level / main memory

- TLBs usually small, typically 32–128 entries
- TLB access time comparable to cache (much faster than accessing main memory)
- TLBs usually are fully/highly associativity

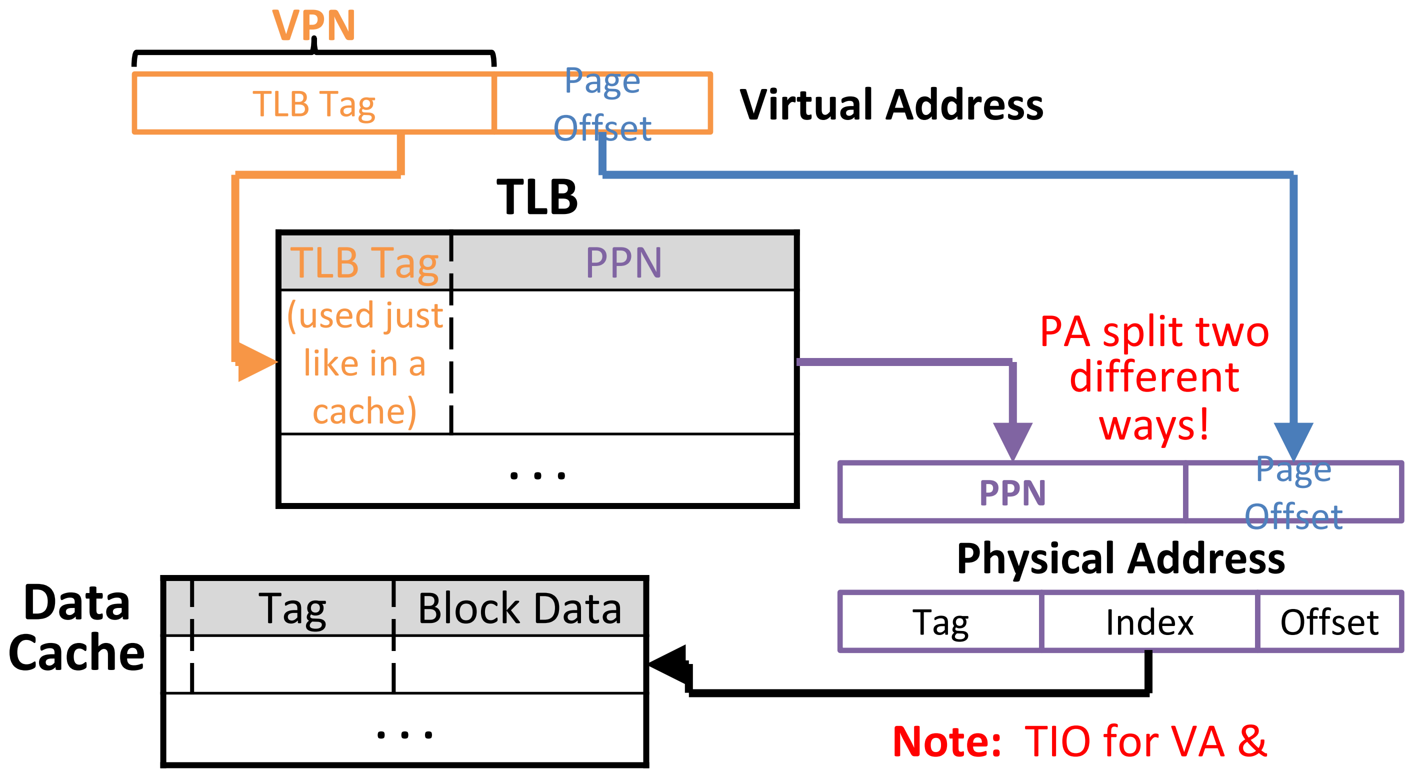
Where Are TLBs Located?

- Which should we check first: Cache or TLB?
  - Can cache hold requested data if corresponding page is not in physical memory? No – check PT first
  - With TLB first, does cache receive VA or PA?



Now the TLB
does the translation too,
not just the Page Table!

Address Translation Using TLB



Note: TIO for VA & PA unrelated

Typical TLB Entry Format

Valid	Ref	Access Rights	Dirty	VPN (Tag)	PPN
X	X	XXX	X		

- *Valid* whether that **TLB ENTRY** is valid (unrelated to PT)
- *Access Rights:* *Data* from the PT
- Dirty: Consistent with PT
- *Ref:* Used to implement LRU
  - Set when page is accessed, cleared periodically by OS
- *PPN: Data* from PT
- VPN: *Data* from PT

Fetching Data on a Memory Read

1) Check TLB (input: VPN, output: PPN)
   - *TLB Hit:* Fetch translation, return PPN
   - *TLB Miss:* Check page table (in memory)
     - *Page Table Hit:* Load page table entry into TLB
     - *Page Table Miss (Page Fault):* Fetch page from disk to memory, update corresponding page table entry, then load entry into TLB
2) Check cache (input: PPN+Page Offset, output: data)
   - *Cache Hit:* Return data value to processor
   - *Cache Miss:* Fetch data value from memory, store it in cache, return it to processor

Page Faults

- Load the page off the disk into a free page of memory
  - Switch to some other process while we wait
- Interrupt thrown when page loaded and the process’ page table is updated
  - When we switch back to the task, the desired data will be in memory
- If memory full, replace page (LRU), writing back if necessary, and update *both* page table entries
  - Continuous swapping between disk and memory called “thrashing”

Context Switching

- How does a single processor run many programs at once?
- *Context switch:* Changing of internal state of processor (switching between processes)
  - Save register values (and PC) and change value in Page Table Base register
- What happens to the TLB?
  - Current entries are for a different process (similar VAs, though!)
  - Set all entries to invalid on context switch

Agenda

- Virtual Memory
- Page Tables
- Translation Lookaside Buffer (TLB)
- VM Performance
- VM Wrap-up

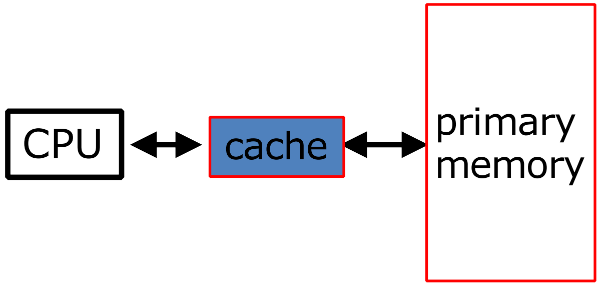
Performance Metrics

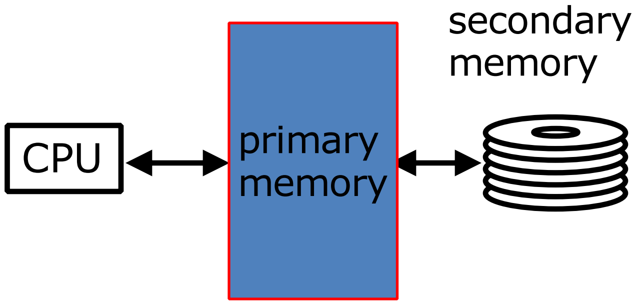
- VM performance also uses Hit/Miss Rates and Miss Penalties
  - *TLB Miss Rate:* Fraction of TLB accesses that result in a TLB Miss
  - *Page Table Miss Rate:* Fraction of PT accesses that result in a page fault
- Caching performance definitions remain the same
  - Somewhat independent, as TLB will always pass PA to cache regardless of TLB hit or miss

VM Performance

- Virtual Memory is the level of the memory hierarchy that sits *below* main memory
  - TLB comes *before* cache, but affects transfer of data from disk to main memory
  - Previously we assumed main memory was lowest level, now we just have to account for disk accesses
- Same CPI, AMAT equations apply, but now treat main memory like a mid-level cache

Typical Performance Stats





Caching	*VM paging*
cache entry	page frame
cache block (≈32 bytes)	page (≈4 Ki bytes)
cache miss rate (1% to 20%)	page miss rate (<0.001%)
cache hit (≈1 cycle)	page hit (≈100 cycles)
cache miss (≈100 cycles)	page fault (≈5M cycles)

Impact of Paging on AMAT (1/2)

- Memory Parameters:
  - L1 cache hit = 1 clock cycles, hit 95% of accesses
  - L2 cache hit = 10 clock cycles, hit 60% of L1 misses
  - DRAM = 200 clock cycles (≈100 nanoseconds)
  - Disk = 20,000,000 clock cycles (≈10 milliseconds)
- Average Memory Access Time (no paging):
  - $1 + 5\% \times 10 + 5\% \times 40\% \times 200 = 5.5$ clock cycles
- Average Memory Access Time (with paging):
  - 5.5 (AMAT with no paging) + ?

Impact of Paging on AMAT (2/2)

- Average Memory Access Time (with paging) =
  - $5.5 + 5\% \times 40\% \times (1\text{-}HR_{Mem}) \times 20{,}000{,}000$
- AMAT if $HR_{Mem} = 99\%$?
  - $5.5 + 0.02 \times 0.01 \times 20{,}000{,}000 = 4005.5$ (≈728x slower)
  - 1 in 20,000 memory accesses goes to disk: 10 sec program takes 2 hours!
- AMAT if $HR_{Mem} = 99.9\%$?
  - $5.5 + 0.02 \times 0.001 \times 20{,}000{,}000 = 405.5$
- AMAT if $HR_{Mem} = 99.9999\%$
  - $5.5 + 0.02 \times 0.000001 \times 20{,}000{,}000 = 5.9$

Impact of TLBs on Performance

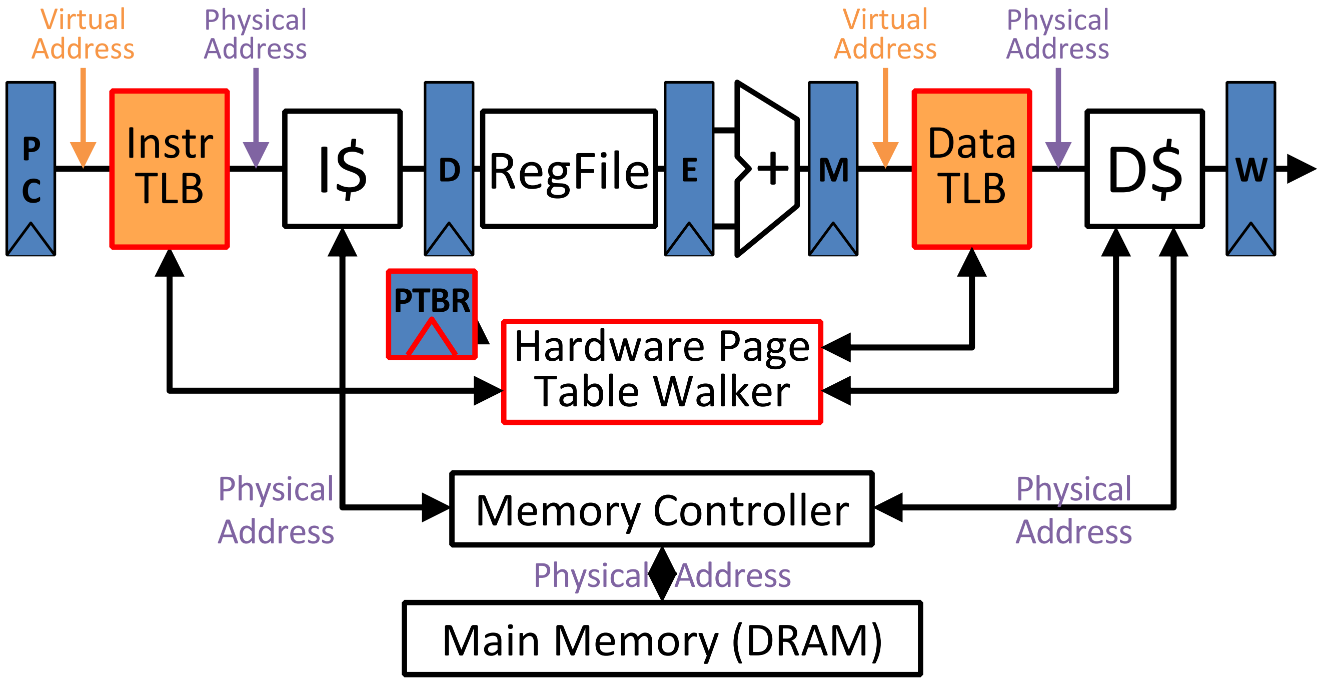
- Each TLB miss to Page Table ~ L1 Cache miss
- *TLB Reach:* Amount of virtual address space that can be simultaneously mapped by TLB:
  - TLB typically has 128 entries of page size 4-8 KiB
  - 128 × 4 KiB = 512 KiB = just 0.5 MiB
- What can you do to have better performance?
  - Multi-level TLBs $\longleftarrow$ Conceptually same as multi-level caches
  - Variable page size (segments)
  - Special situationally-used "superpages"

$\left.\begin{array}{l}\text{Variable page size (segments)}\\ \text{Special situationally-used “superpages”}\end{array}\right\}$ Not covered in CS61C

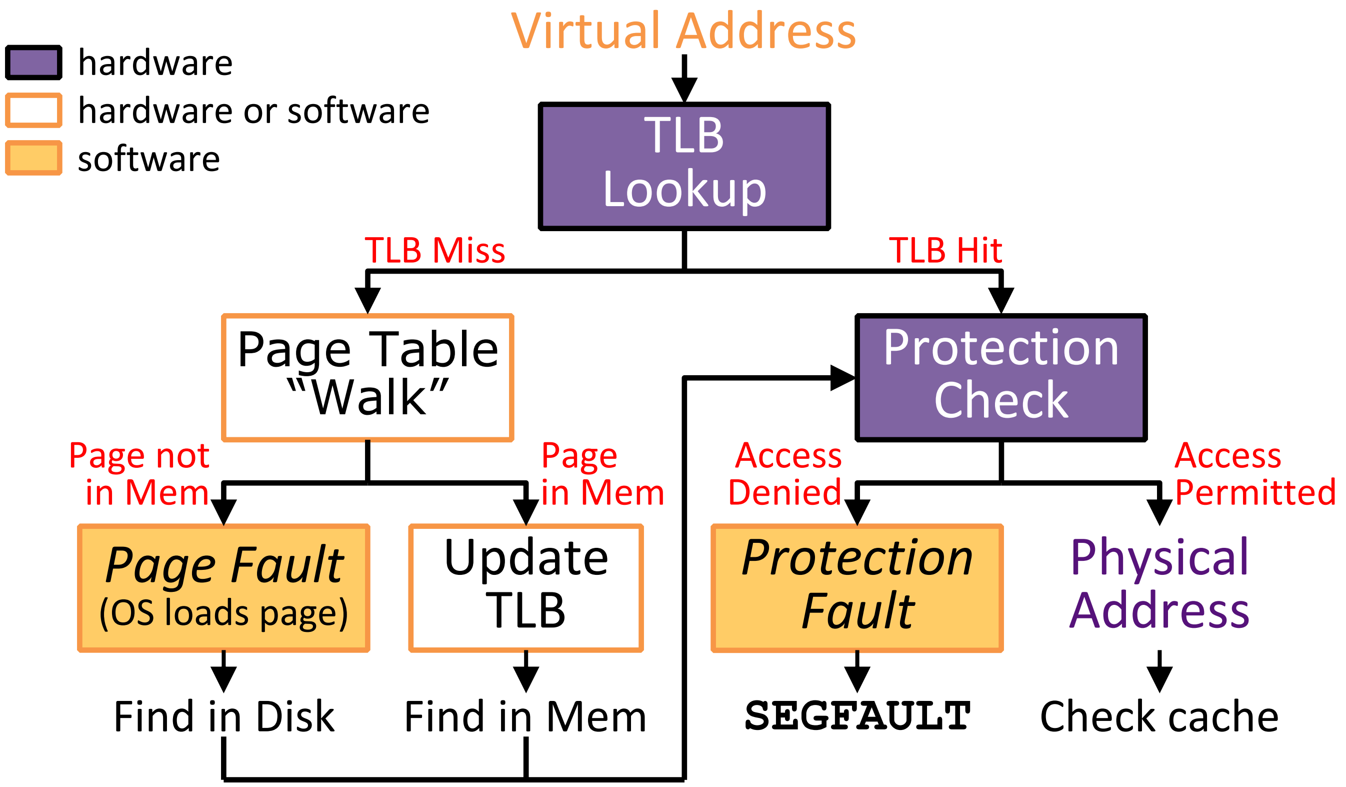
Agenda

- Virtual Memory
- Page Tables
- Administrivia
- Translation Lookaside Buffer (TLB)
- VM Performance
- VM Wrap-up

Page-Based Virtual-Memory Machine



Address Translation



Summary

- User program view:
  - Contiguous memory
  - Start from some set VA
  - “Infinitely” large
  - Is the only running program
- Reality:
  - Non-contiguous memory
  - Start wherever available memory is
  - Finite size
  - Many programs running simultaneously
- Virtual memory provides:
  - Illusion of contiguous memory
  - All programs starting at same set address
  - Illusion of ~ infinite memory (2^{32} or 2^{64} bytes)
  - Protection, Sharing
- Implementation:
  - Divide memory into chunks (pages)
  - OS controls page table that maps virtual into physical addresses
  - memory as a cache for disk
  - TLB is a cache for the page table