CS 61C:
Great Ideas in Computer Architecture

Lecture 23:
Virtual Memory

John Wawrzynek & Nick Weaver

http://inst.eecs.berkeley.edu/~cs61c

From Previous Lecture: Operating Systems

- Input / output (I/O)
  - Memory mapped: appears like "special kind of memory"
  - Access with usual load/store instructions (e.g., **`lw,sw`**)
- Exceptions
  - Notify processor of special events, e.g. divide by 0, *page fault* (this lecture)
  - "Precise" handling: immediately at offending instruction
- Interrupts
  - Notification of external events, e.g., keyboard input, disk or Ethernet traffic
- Machine "boot" procedure

This Lecture: Operating Systems

- Multiprogramming and supervisory mode
  - Enables and isolates multiple programs
- **Virtual Memory**

Supervisor Mode

- If something goes wrong in an application, it could crash the entire machine. And what about malware, etc.?
- The OS enforces resource constraints to applications (e.g., access to memory, devices)
- To help OS provide protection from applications, CPUs have a supervisor mode (e.g., set by a status bit in a special register)
  - When not in supervisor mode (i.e., in *user mode*), a process can only access a subset of instructions and (physical) memory
  - Process can change out of supervisor mode using a special instruction, but not into it directly – only using an exception
  - Supervisory mode is a bit like "superuser"
    - But used much more sparingly (most of OS code does *not* run in supervisory mode)

Syscalls

- What if we want to call an OS routine? E.g.,
  - to read a file,
  - launch a new process,
  - ask for more memory (“sbreak” used by malloc),
  - send data over the network, etc.
- Need to perform a syscall:
  - Set up function arguments in registers,
  - Raise exception (with special assembly “trap” instruction)
- OS will perform the operation and return to user mode
- This way, the OS can mediate access to all resources, and devices

Agenda

- Devices and I/O
- Polling
- Interrupts
- OS Boot Sequence
- Multiprogramming/time-sharing

Multiprogramming

- The OS runs multiple applications and processes at the same time
- But not really (unless you have a core per process)
- Switches between processes very quickly (on human time scale) – this is called a "context switch"
- When jumping into process, set timer interrupt
  1. When it expires, store PC, registers, etc. (process state)
  2. Pick a different process to run and load its state
  3. Set timer, change to user mode, jump to the new PC
- Deciding what process to run is called scheduling
  - All processes in the system reside in a scheduling queue
  - Some are ready to execute (waiting their turn)
  - Others are waiting on I/O

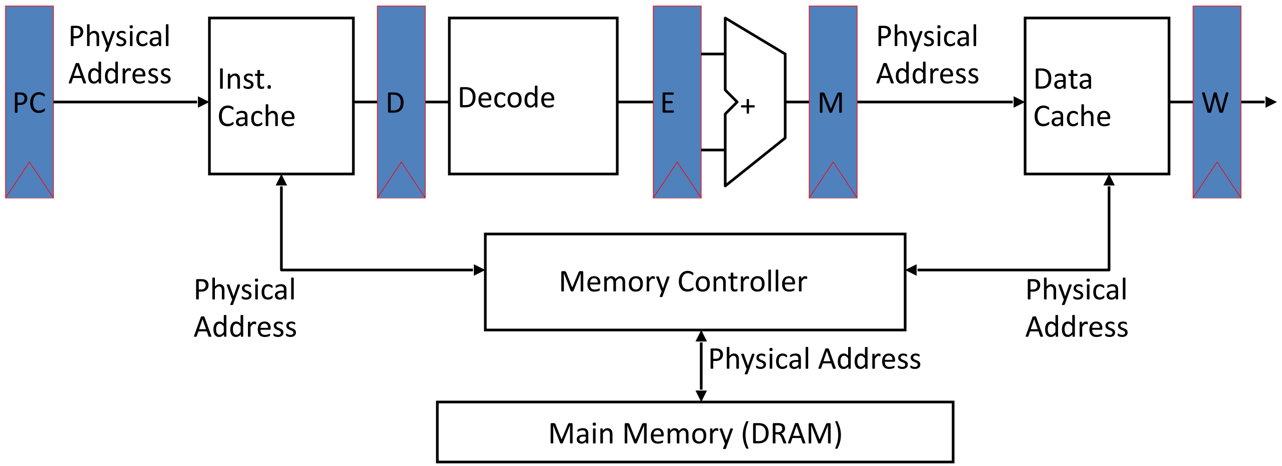
Protection, Translation, Paging

1. Supervisor mode alone is not sufficient to fully isolate applications from each other or from the OS
   - Application could overwrite another application's memory.
2. Typically programs start at some fixed address, e.g. 0x8FFFFFFF
   - How can 100's of programs all share memory at location 0x8FFFFFFF?
3. Also, may want to address more memory than we actually have (e.g., for sparse data structures)

- Solution: Virtual Memory
  - Gives each process the *illusion* of a full memory address space that it has completely for itself

Virtual Memory (VM)

"Bare" 5-Stage Pipeline

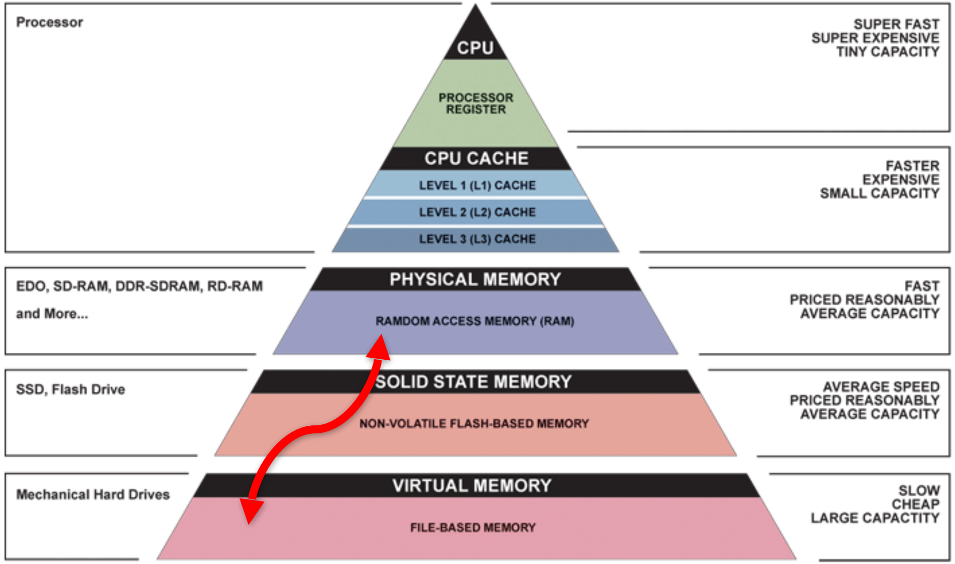


- In a bare machine, the only kind of address is a physical address

What do we need Virtual Memory for?
Reason 1: Adding Disks to Hierarchy

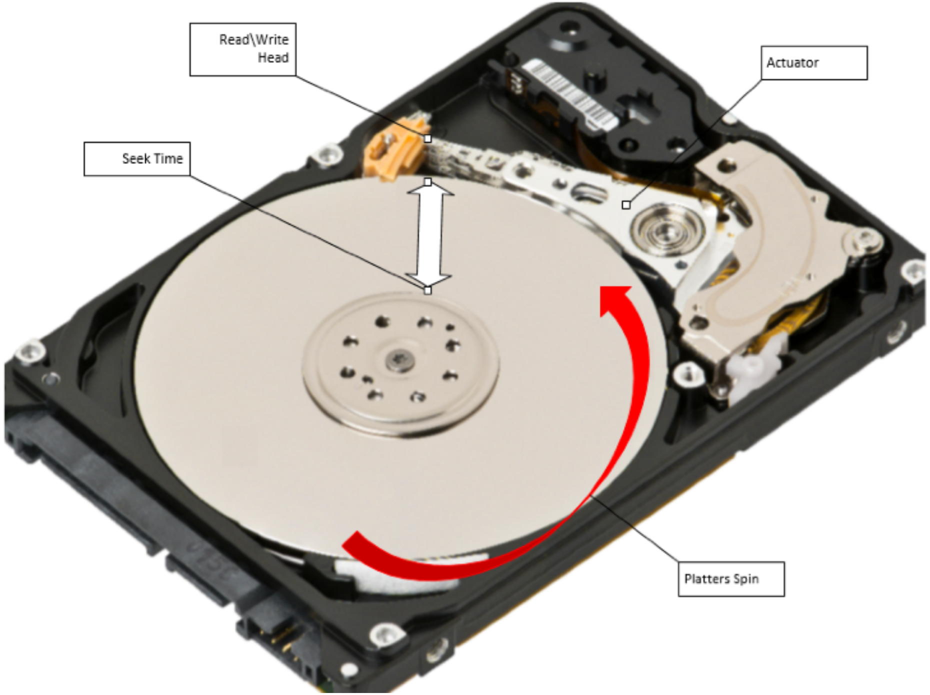
- Need to devise a mechanism to “connect” memory and disk in the memory hierarchy

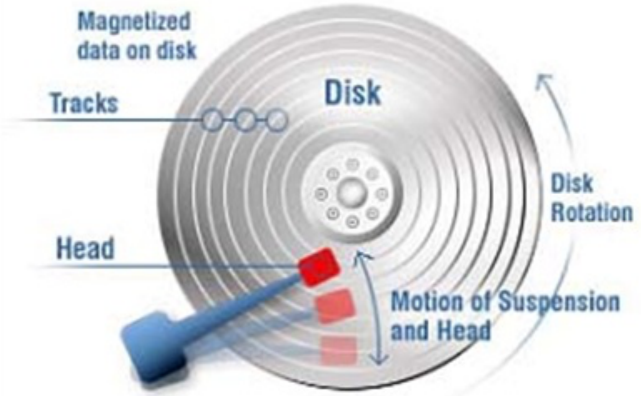
- Disk
  - Slow
  - But huge
  - How could we make use of its capacity (when running low on DRAM)?



▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

Aside … Why are Disks So Slow?

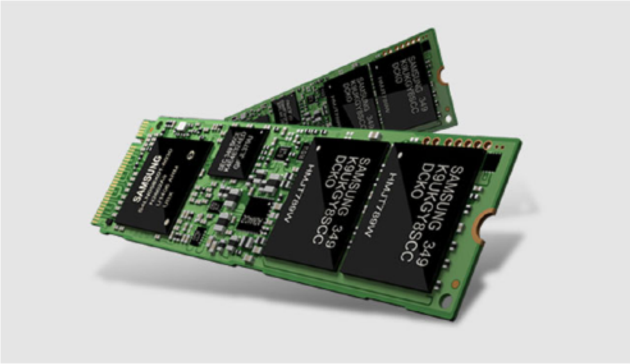




- 10,000 rpm (revolutions per minute)
- 6 ms per revolution
- Average random access time: 3 ms

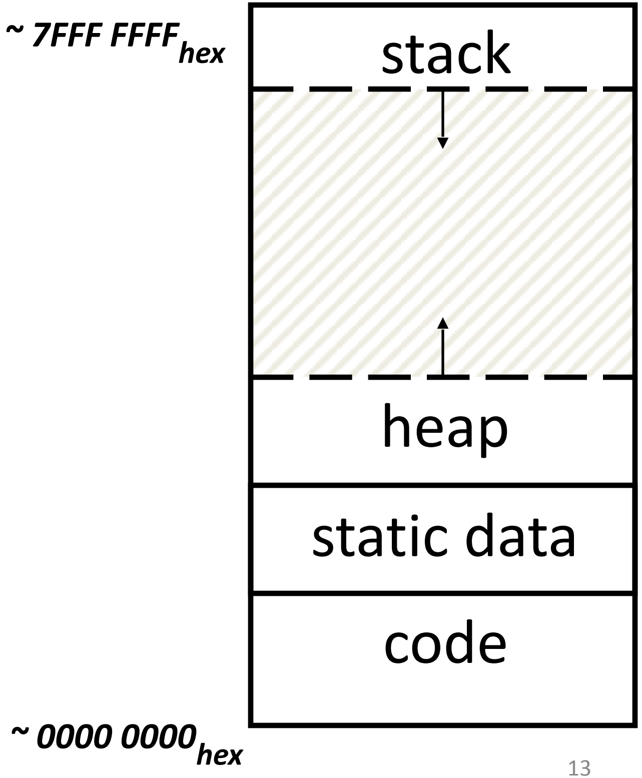
What About SSD?

- Made with transistors - same technology as Flash memory
- Nothing mechanical that turns
- Like “Ginormous” DRAM
  - Except “nonvolatile” - holds contents when power is off
- Fast access to all locations, regardless of address
- Still much slower than register, caches, DRAM
  - Read/write blocks, not bytes



What do we need Virtual Memory for? Reason 2: Simplifying Memory for Apps

- Processes should see the straightforward memory layout we saw earlier ->
- User-space applications should think they own all of memory
- So we give them a **virtual** view of memory



What do we need Virtual Memory for? Reason 3: Protection Between Processes

- With a bare system, addresses issued with loads/stores are real **physical** addresses
- This means any process can issue any address, therefore can access any part of memory, even areas which it doesn't own
  - Ex: The OS data structures
- We should send all addresses through a mechanism that the OS controls, before they make it out to DRAM - **a translation mechanism**
  - Can check that process has permission to access a particular part of memory


KEEP
CALM
IT'S
BREAK
TIME

Address Spaces

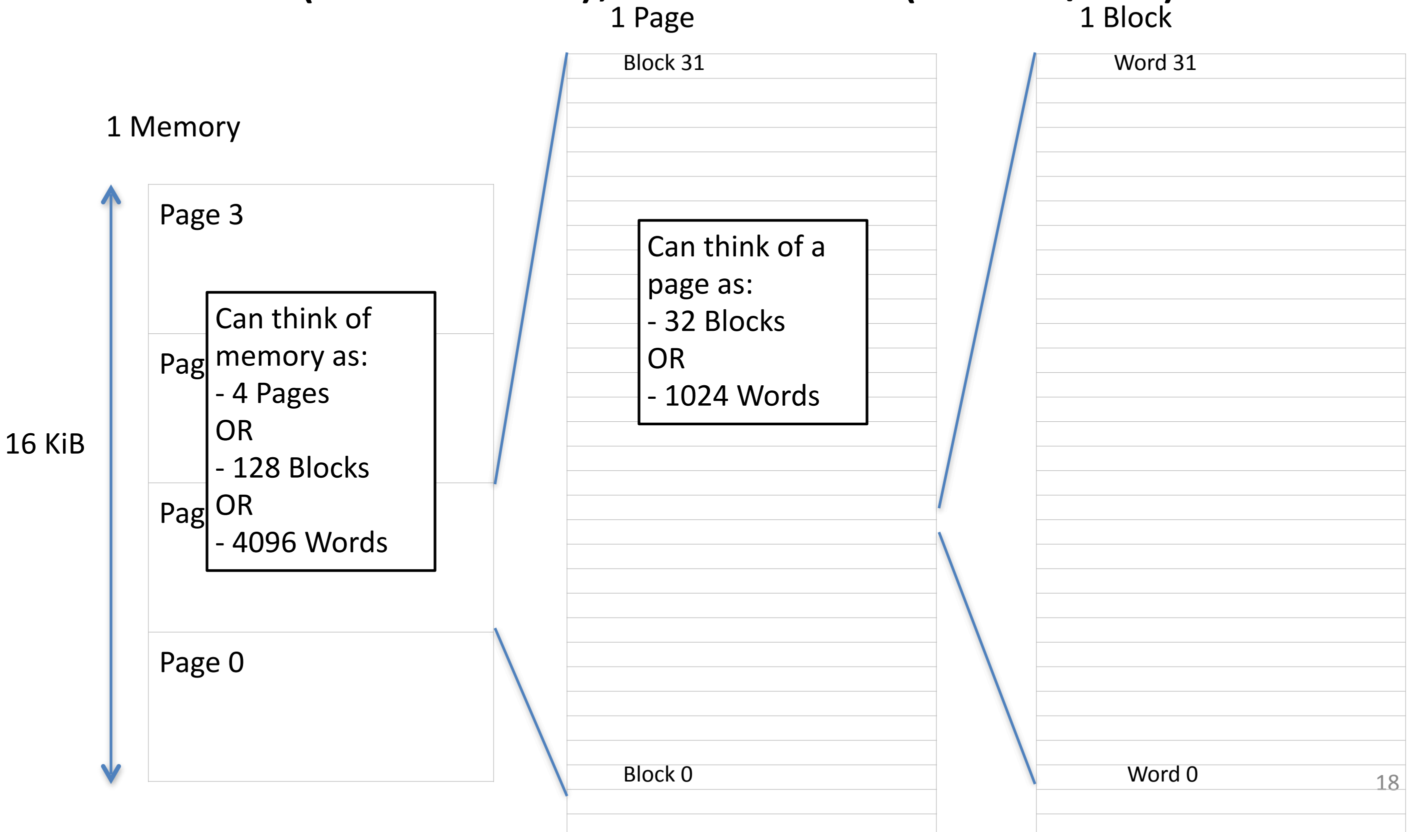
- Address space = set of addresses for all available memory locations
- Now, two kinds of memory addresses:
  - **Virtual Address Space**
    - Set of addresses that the user program knows about
  - **Physical Address Space**
    - Set of addresses that map to actual physical locations in memory
   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

Aside: Blocks vs. Pages

- In caches, we dealt with individual *blocks*
  - Usually ~64B on modern systems
  - We “divide” memory into a set of blocks
- In VM, we deal with individual *pages*
  - Usually ~4 KB on modern systems
  - Now, we’ll “divide” memory into a set of pages
- Common point of confusion: Bytes, Words, Blocks, Pages are all just different ways of looking at memory!

Bytes, Words, Blocks, Pages

Ex: 16 KiB DRAM, 4 KiB Pages (for VM), 128 B blocks (for caches), 4 B words (for lw/sw)

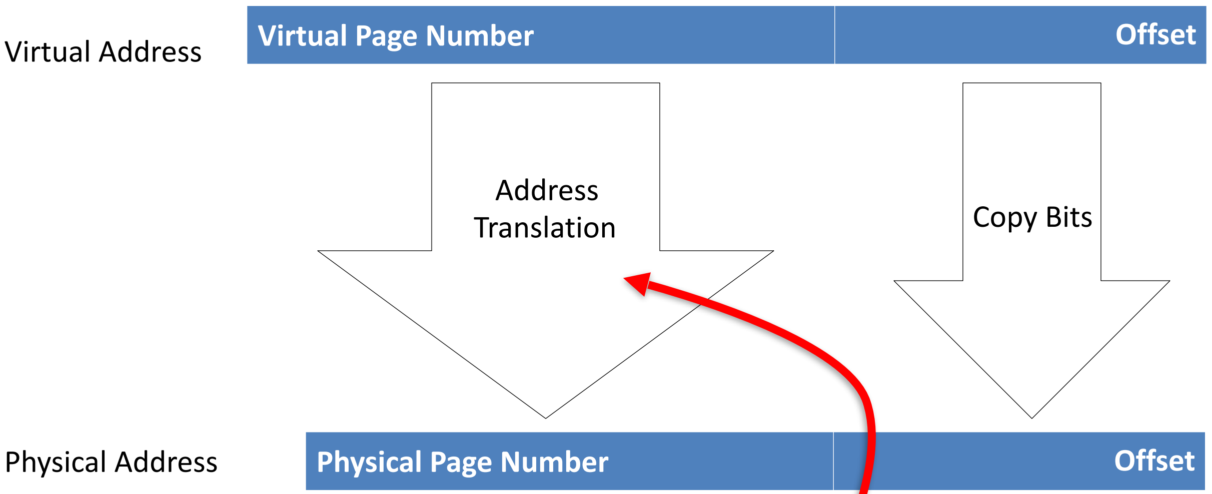


Address Translation

- So, what do we want to achieve at the hardware level?
  - Take a Virtual Address, that points to a spot in the Virtual Address Space of a particular program, and map it to a Physical Address, which points to a physical spot in DRAM of the whole machine

Virtual Address	**Virtual Page Number**	**Offset**
Physical Address	**Physical Page Number**	**Offset**

Address Translation



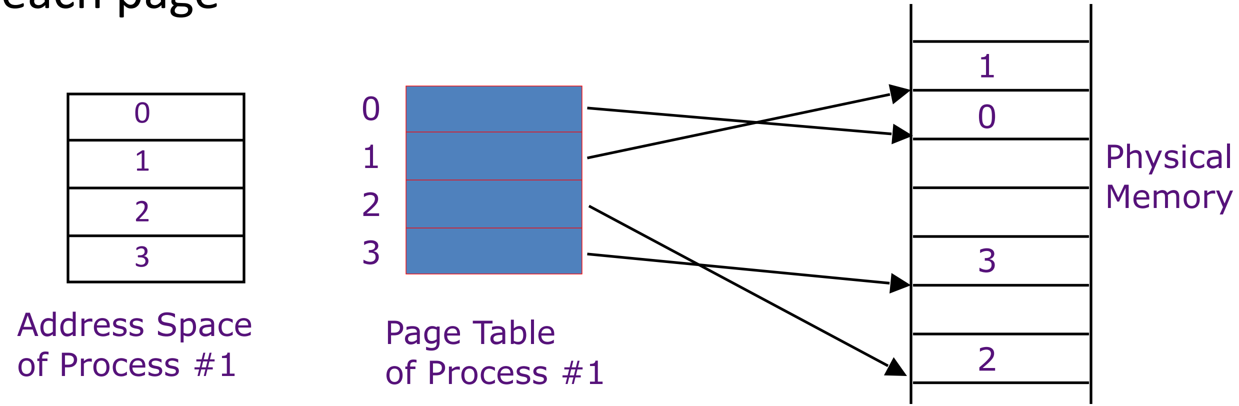
The rest of the lecture is all about implementing

Paged Memory Systems

- Processor-generated address can be split into:

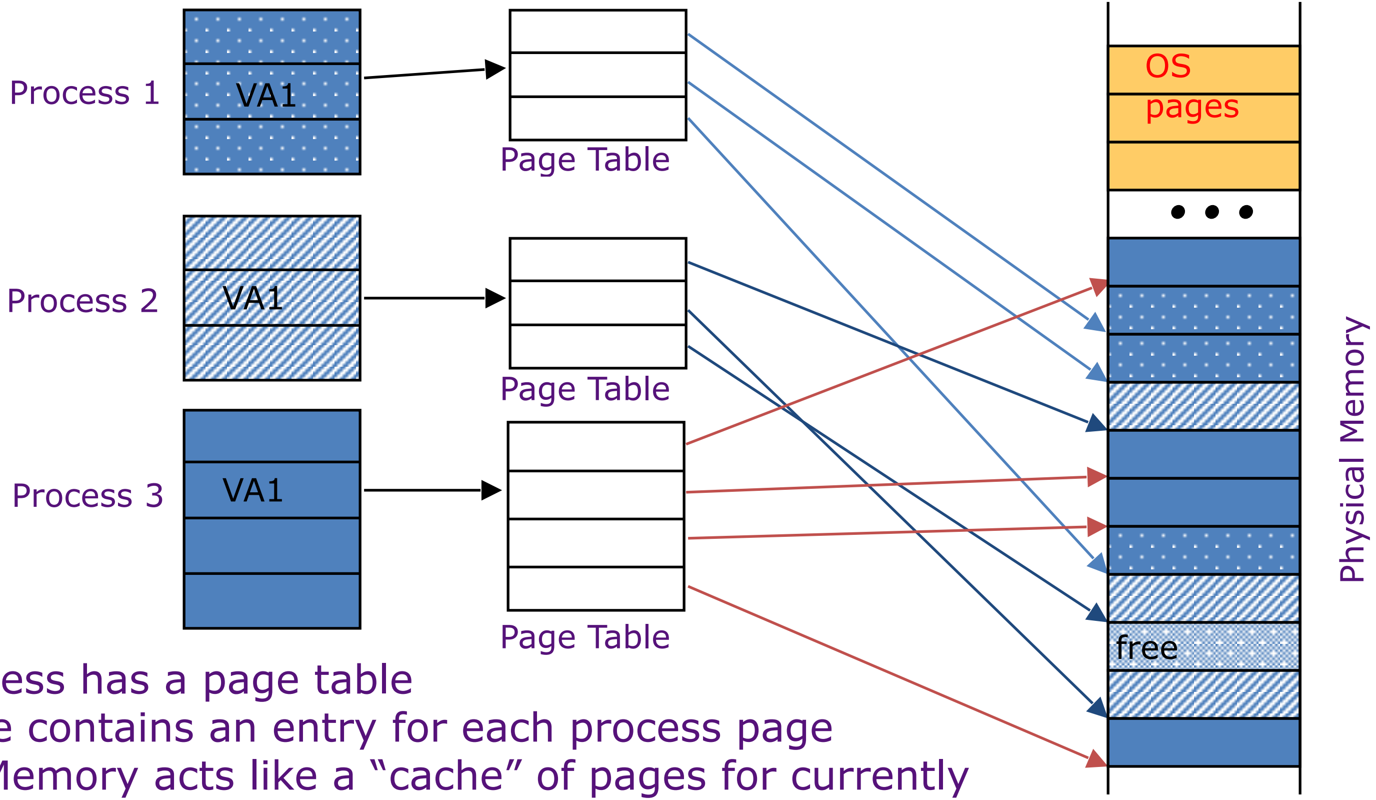
Virtual Page Number	Offset

- A *page table* contains the physical address of the base of each page



Page tables make it possible to store the pages of a process non-contiguously.

Private (Virtual) Address Space per Process



- Each process has a page table
- Page table contains an entry for each process page
- Physical Memory acts like a "cache" of pages for currently running programs. **Not recently used pages are stored in secondary memory, e.g. disk (in "swap partition")**

Where Should Page Tables Reside?

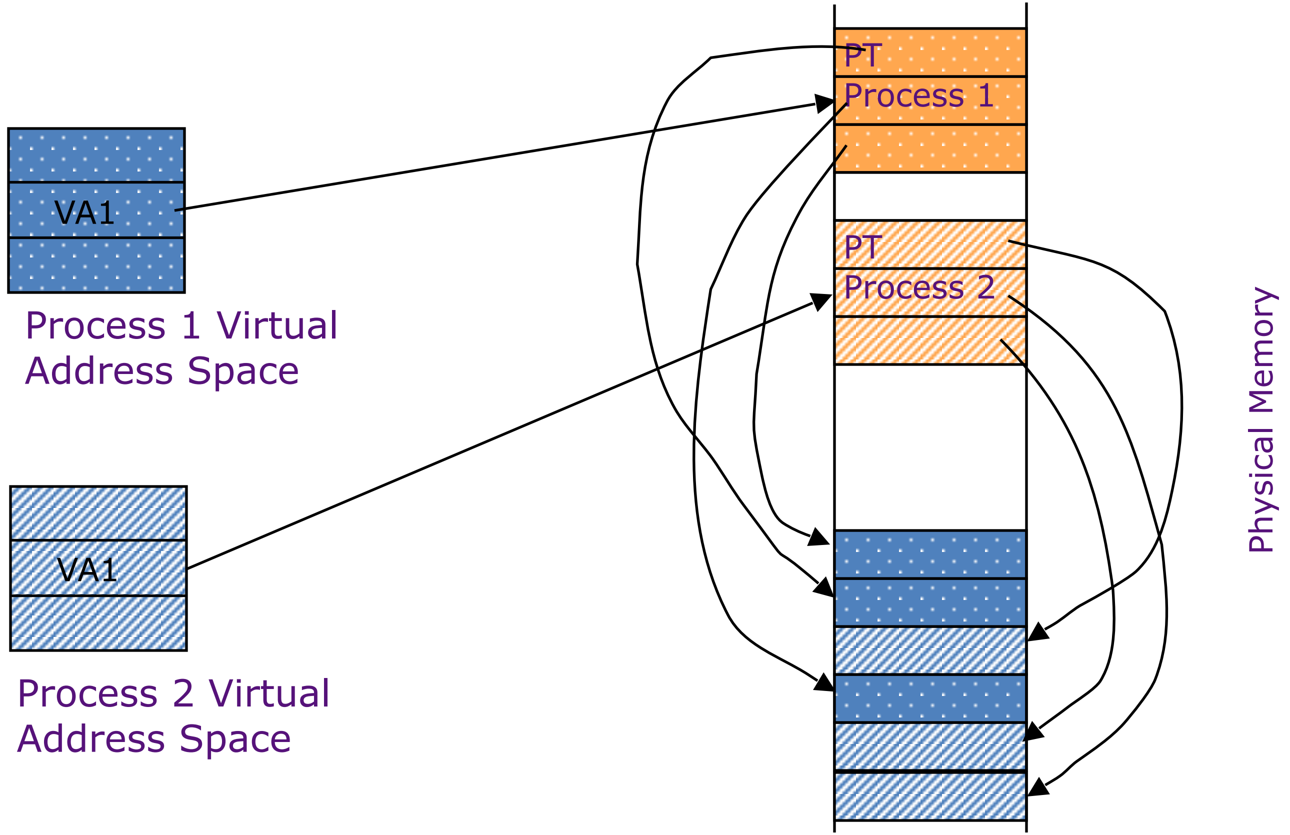
- Space required by the page tables (PT) is proportional to the address space, number of processes, ...

  $\Rightarrow$ *Too large to keep in registers inside CPU*

- Idea: Keep page tables in the main memory
  - Needs one reference to retrieve the page physical address and another to access the data word

    $\Rightarrow$ *doubles the number of memory references! (but we can fix this using something we already know about…)*

Page Tables in Physical Memory

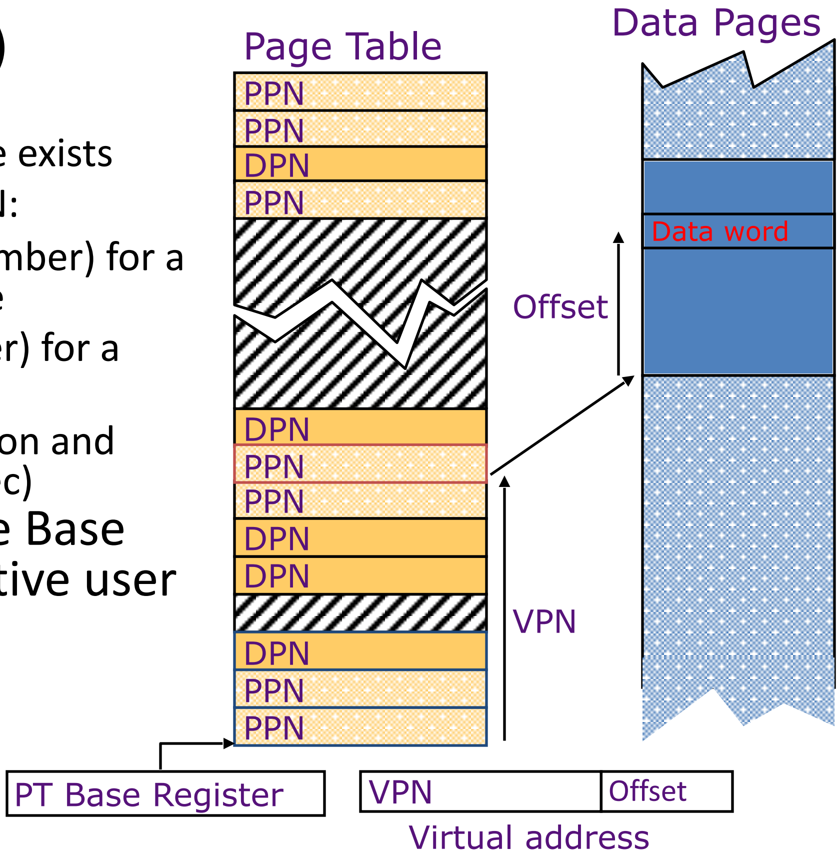


Administrivia

- Upcoming Lecture Schedule
  - 4/17: VM (today)
  - 4/19: I/O: DMA, Disks, Networking
  - 4/24: Dependability: Parity, ECC, RAID
    - Last day of new material
  - 4/26: Summary, What's Next?
- HW 5 (final homework!) tomorrow, due Friday of last week of classes
- Project 5 (WSC related) will be released soon
  - You'll get at ~1.5 weeks to complete it - due Monday RRR week

Linear (simple) Page Table

- Page Table Entry (PTE) contains:
  - 1 bit to indicate if page exists
  - And either PPN or DPN:
  - PPN (physical page number) for a memory-resident page
  - DPN (disk page number) for a page on the disk
  - Status bits for protection and usage (read, write, exec)
- OS sets the Page Table Base Register whenever active user process changes



Suppose an instruction references a memory page that isn't in DRAM?

- We get a exception of type "<u>page fault</u>"
- Page fault handler does the following:
  1. If virtual page doesn't yet exist, assign it an unused page in DRAM, or if page exists …
  2. Initiate transfer of the page contents we're requesting from disk to DRAM, assigning to an unused DRAM page
  3. If no unused page is left, a *page currently in DRAM is selected to be replaced* (based on usage - LRU)
  4. The replaced page is written (back) to disk, page table entry that maps that VPN->PPN is marked with DPN
  5. Page table entry of the (virtual) page we're requesting is updated with a (now) valid PPN
- Following the page fault, re-execute the instruction

Size of Linear Page Table

With 32-bit memory addresses, 4-KB pages:

$\Rightarrow$ $2^{32} / 2^{12} = 2^{20}$ virtual pages per user, assuming 4-Byte PTEs,

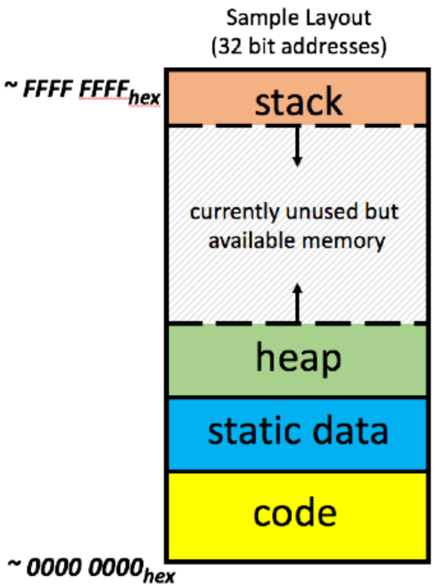
$\Rightarrow$ 2^{20} PTEs, i.e, 4 MB page table per user!

Larger pages?

- Internal fragmentation (Not all memory in page gets used)
- Larger page fault penalty (more time to read from disk)

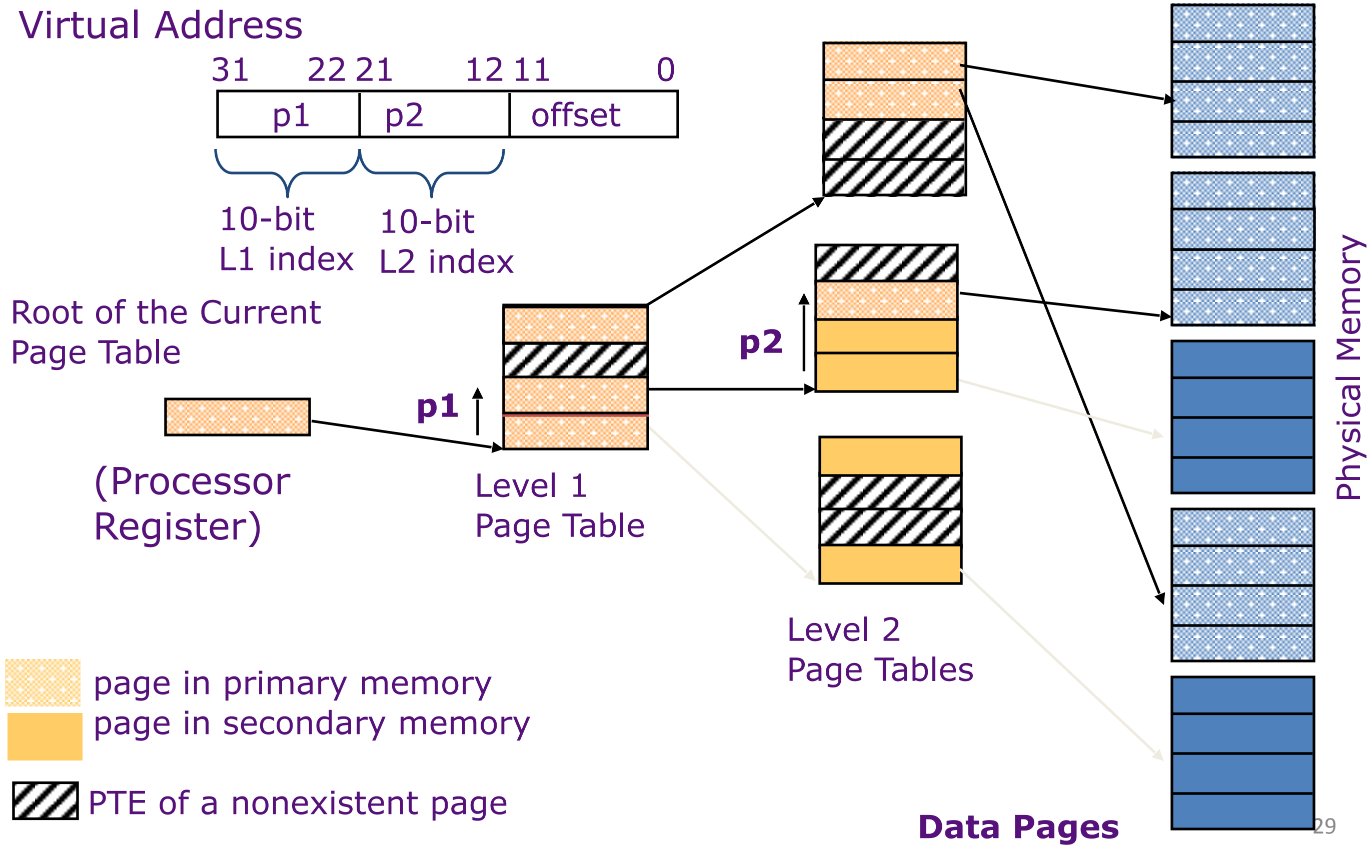
What about 64-bit virtual address space???

- Even 1MB pages would require 2^{44} 8-Byte PTEs (35 TB!)

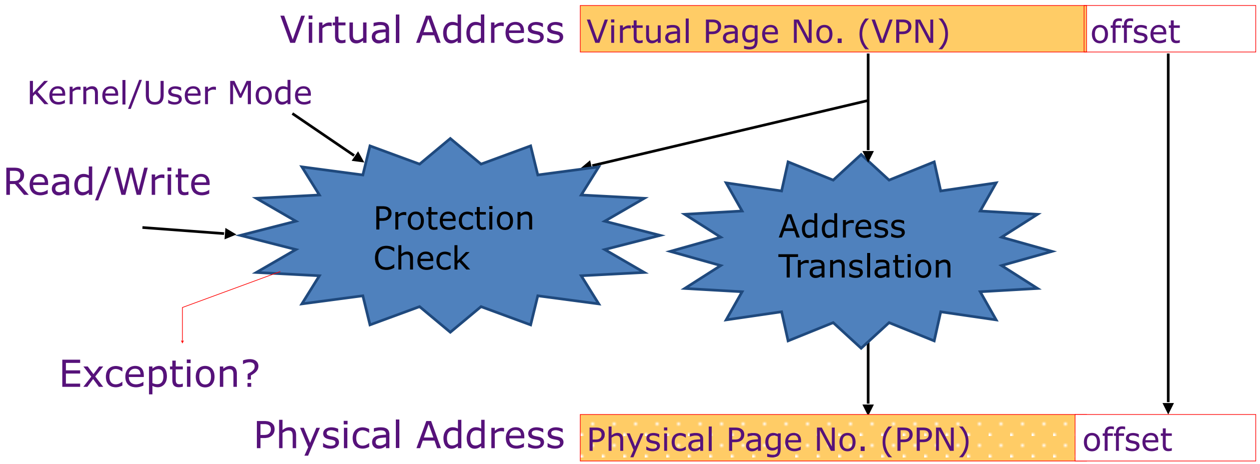


What is the "saving grace" ? Most processes only use a set of high address (stack), and a set of low address (instructions, heap)

Hierarchical Page Table – exploits sparcity of virtual address space use



Address Translation & Protection



- Every instruction and data access needs address translation and protection checks

A good VM design needs to be fast (~ one cycle) and space efficient

Translation Lookaside Buffers (TLB)

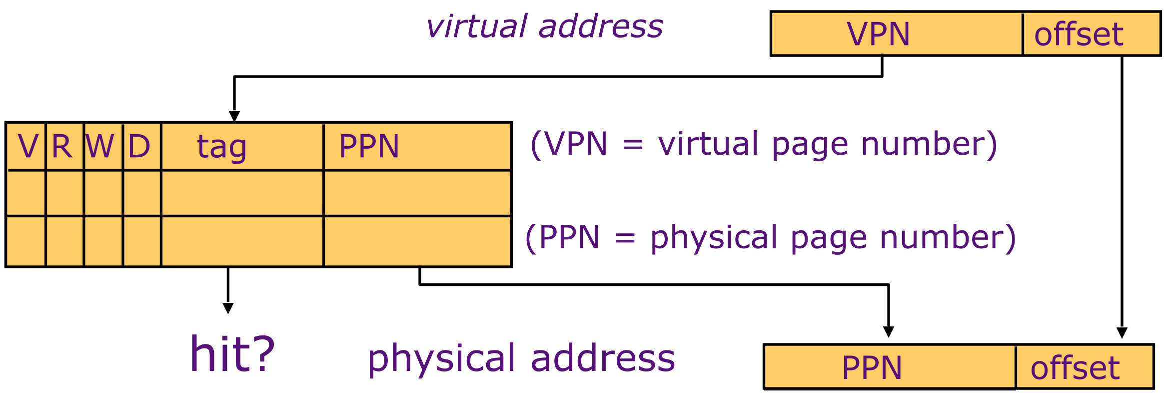
Address translation is very expensive!

In a two-level page table, each reference becomes several memory accesses

Solution: *Cache some translations in TLB*

TLB hit $\Rightarrow$ *Single-Cycle Translation*

TLB miss $\Rightarrow$ *Page-Table Walk to refill*



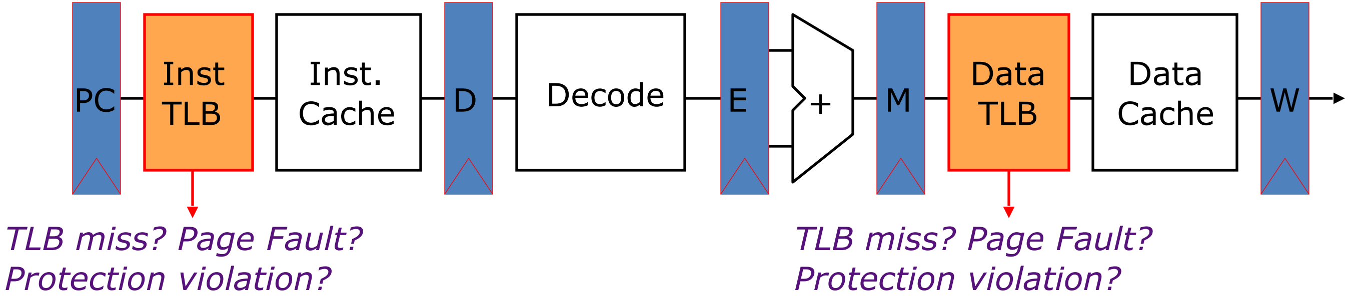
TLB Designs

- Typically 32-128 entries, sometimes fully associative
  - Each entry maps a large page, hence less spatial locality across pages => more likely that two entries conflict
  - Sometimes larger TLBs (256-512 entries) are 4-8 way set-associative
  - Larger systems sometimes have multi-level (L1 and L2) TLBs
- Random or FIFO replacement policy
- "TLB Reach": Size of largest virtual address space that can be simultaneously mapped by TLB

Example: 64 TLB entries, 4KB pages, one page per entry

TLB Reach = ____?

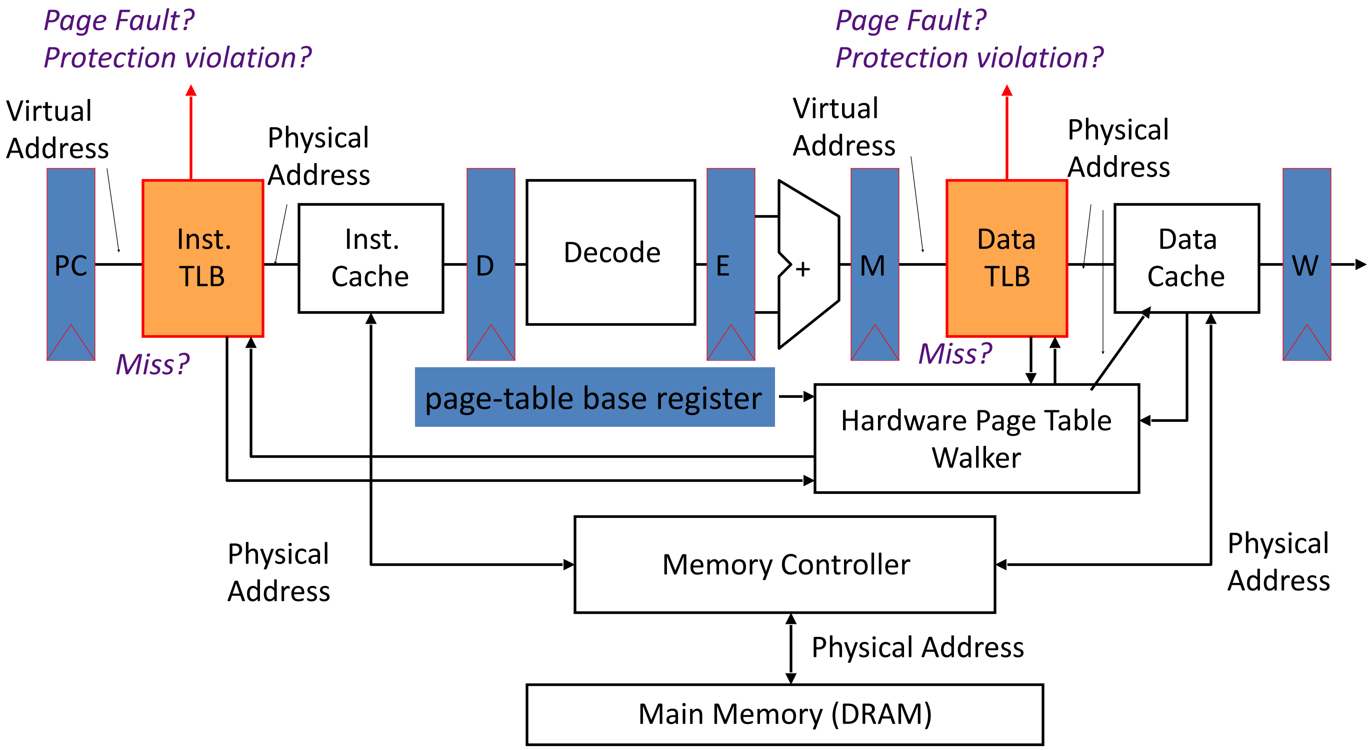
VM-related exceptions in pipeline



- Handling a TLB miss needs a hardware or software mechanism to refill TLB
  - usually done in hardware now
- Handling a page fault (e.g., page is on disk) needs a *precise* trap so software handler can easily resume after retrieving page
- Handling protection violation may abort process

Page-Based Virtual-Memory Machine

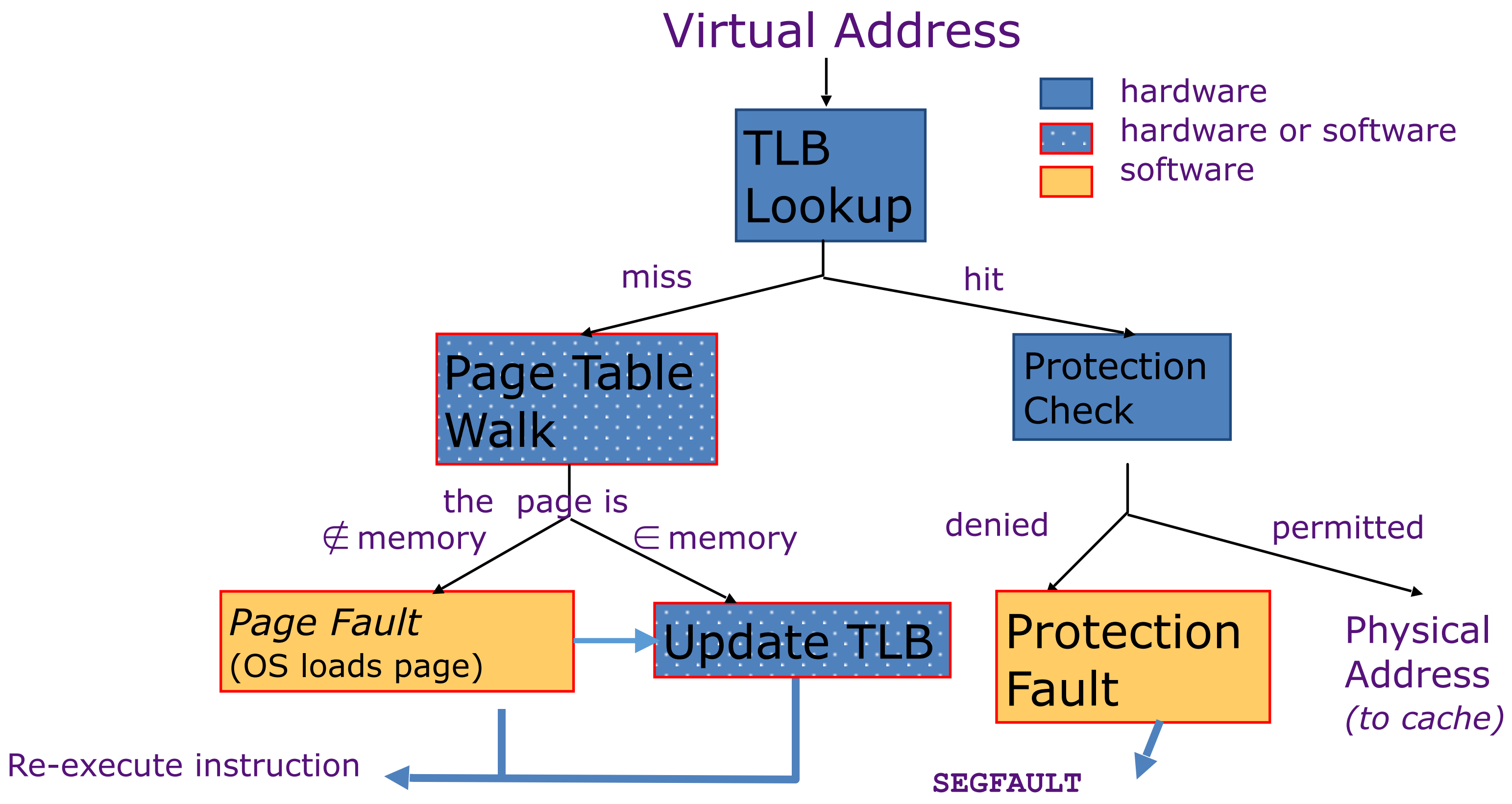
(Hardware Page-Table Walk)



- Assumes page tables held in untranslated physical memory

Address Translation:

putting it all together



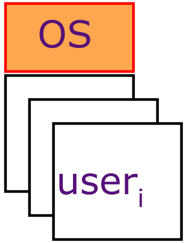
Summary: Virtual Memory Systems

Illusion of a large, private, uniform store

Protection & Privacy

several users, each with their private address space and one or more shared address spaces

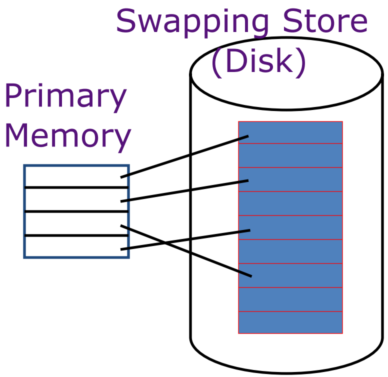
page table ≡ name space



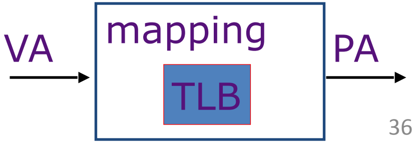
Demand Paging

Provides the ability to run programs larger than the primary memory

Hides differences in machine configurations



The price is address translation on each memory reference



Clicker Question

Let’s try to extrapolate from caches… Which one is false?

A. # offset bits in V.A. = log2(page size)

B. # offset bits in P.A. = log2(page size)

C. # VPN bits in V.A. = log2(# of physical pages)

D. # PPN bits in P.A. = log2(# of physical pages)

E. A single-level page table contains a PTE for every possible VPN in the system

And, in Conclusion …

- Virtual and physical addresses
  - Program → virtual address
  - DRAM → physical address
- Paged Memory
  1. Facilitates virtual → physical address translation
  2. Provides isolation & protection
  3. Extends available memory to include disk
- Implementation issues
  - Hierarchical page tables
  - Caching page table entries (TLB)