Constructive Computer Architecture

Virtual Memory: From Address Translation to Demand Paging

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

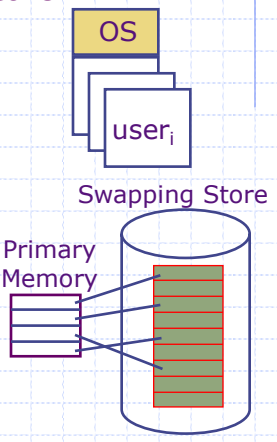
Contributors to the course material

- Arvind, Rishiyur S. Nikhil, Joel Emer, Muralidaran Vijayaraghavan
- Staff and students in 6.375 (Spring 2013), 6.S195 (Fall 2012), 6.S078 (Spring 2012)
  - Asif Khan, Richard Ruhler, Sang Woo Jun, Abhinav Agarwal, Myron King, Kermin Fleming, Ming Liu, Li-Shiuan Peh
- External
  - Prof Amey Karkare & students at IIT Kanpur
  - Prof Jihong Kim & students at Seoul Nation University
  - Prof Derek Chiou, University of Texas at Austin
  - Prof Yoav Etsion & students at Technion

Modern Virtual Memory Systems

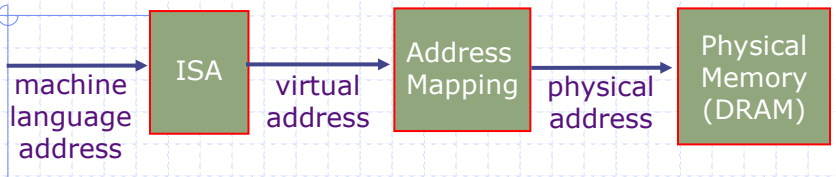
Illusion of a large, private, uniform store

- Protection & Privacy
  - Each user has one private 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 of VM is address translation on each memory reference

Names for Memory Locations



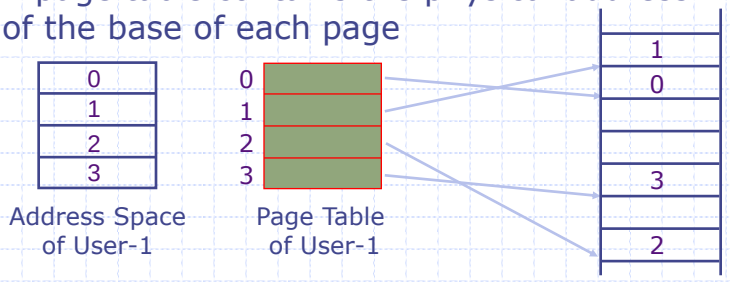
- Machine language address
  - as specified in machine code
- Virtual address
  - ISA specifies translation of machine code address into virtual address of program variable (sometime called *effective* address)
- Physical address
  - operating system specifies mapping of virtual address into name for a physical memory location

Paged Memory Systems

- Processor generated address can be interpreted as a pair <page number, offset>

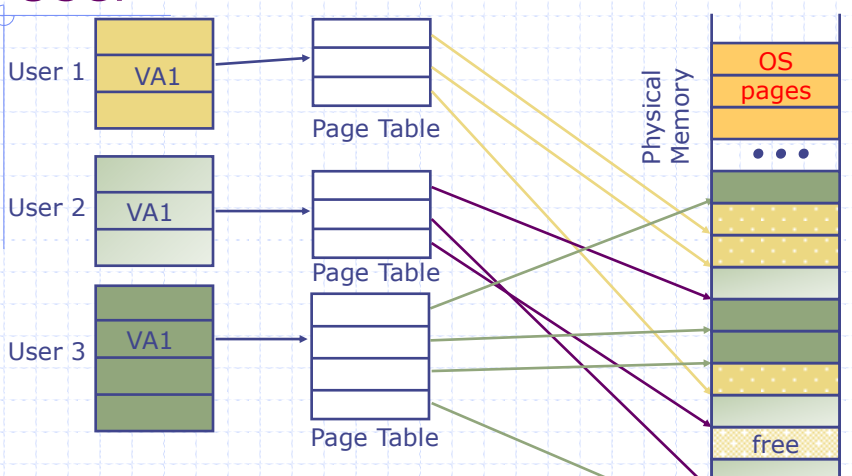
page number	offset

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



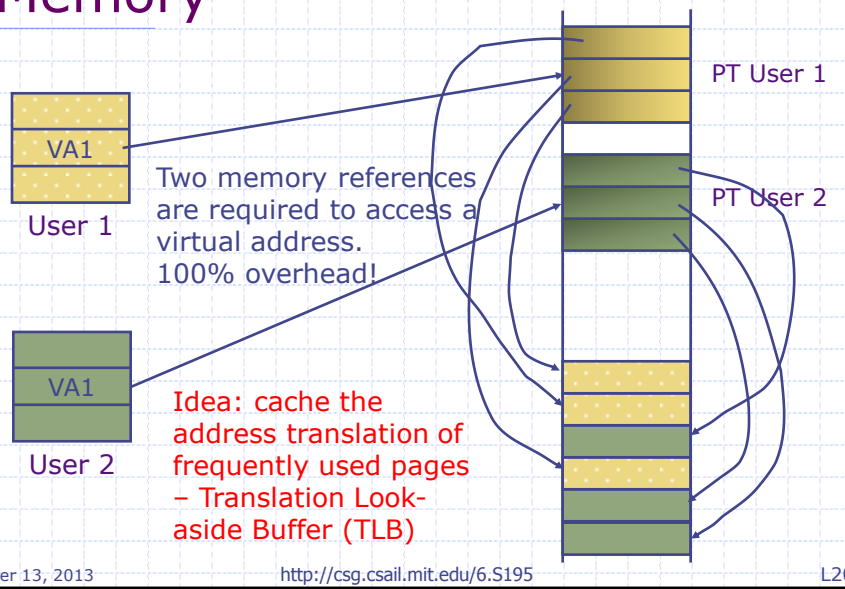
Page tables make it possible to store the pages of a program non-contiguously

Private Address Space per User



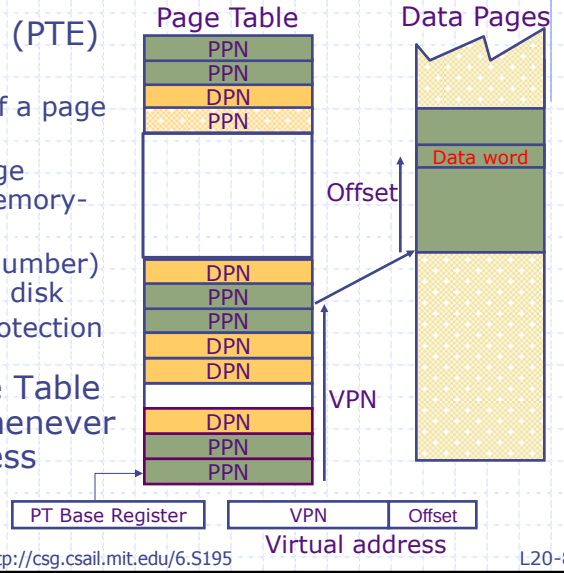
- Each user has a page table
- Page table contains an entry for each user page

Page Tables in Physical Memory



Linear Page Table

- Page Table Entry (PTE) contains:
  - A bit to indicate if a page exists
  - 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
- OS sets the Page Table Base Register whenever active user process changes



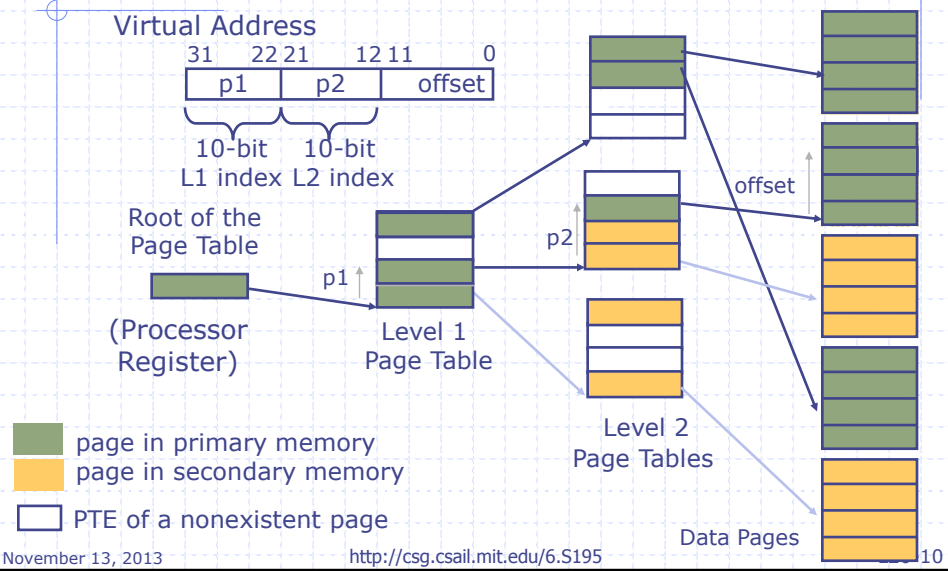
Size of Linear Page Table

- With 32-bit addresses, 4-KB pages & 4-byte PTEs
  - 2^{20} PTEs, i.e, 4 MB page table per user
  - 4 GB of swap space needed to back up the full virtual address space
- Larger Pages can reduce the overhead *but cause*
  - Internal fragmentation (Not all memory in a page is 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!)

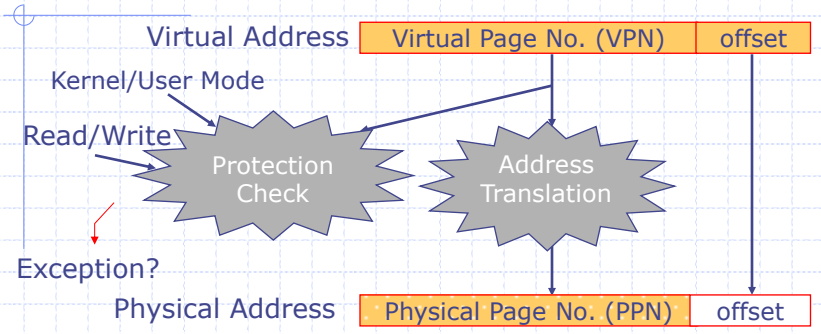
Any *"saving grace"* ?

Page tables are sparsely populated and hence hierarchical organization can help

Hierarchical Page Table



Address Translation & Protection



- Every instruction access and data access needs address translation and protection checks
- Address translation is very expensive!
  - In a one-level page table, each reference becomes two or more memory accesses

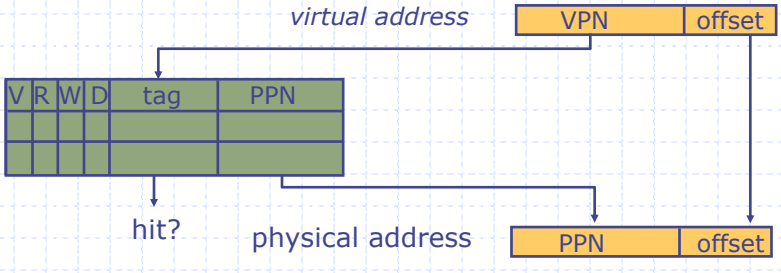
A good VM design needs to be fast and space efficient

Translation Lookaside Buffers (TLB)

Cache address translations in TLB

TLB hit ⇒ *Single Cycle Translation*

TLB miss ⇒ *Page Table Walk to refill*



TLB Designs

- Typically 32-128 entries, usually 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
- Random or FIFO replacement policy
- Process ID information in TLB?
- 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

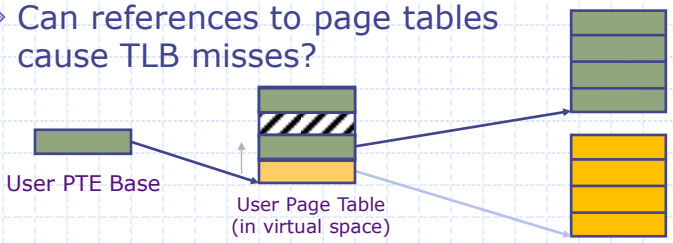
  TLB Reach = *64 entries * 4 KB = 256 KB*

Handling a TLB Miss

- Software (MIPS, Alpha)
  - TLB miss causes an exception and the operating system walks the page tables and reloads TLB
  - A privileged "untranslated" addressing mode is used for PT walk
- Hardware (SPARC v8, x86, PowerPC)
  - A memory management unit (MMU) walks the page tables and reloads the TLB
  - If a missing (data or PT) page is encountered during the TLB reloading, MMU gives up and signals a Page-Fault exception for the original instruction

Translation for Page Tables

- Can references to page tables cause TLB misses?

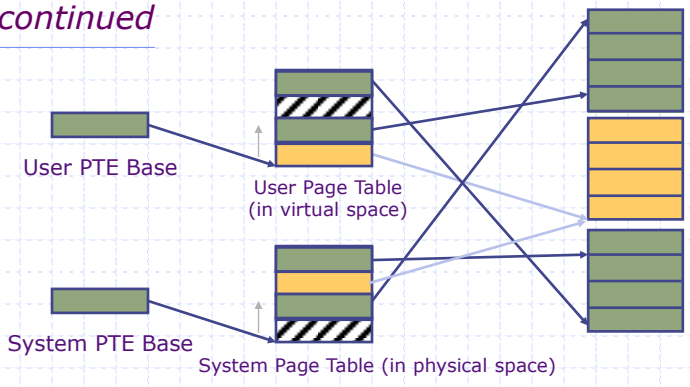


- User VA translation causes a TLB miss
- *Page table walk:* User PTE Base and appropriate bits from VA are used to obtain virtual address (VP) for the page table entry
- Suppose we get a TLB miss when we try to translate VP?

Must know the physical address of the page table

Translation for Page Tables

continued



- On a TLB miss during a VP translation, OS adds System PTE Base to bits from VP to find physical address of page table entry for the VP
- A program that traverses the page table needs a "no translation" addressing mode

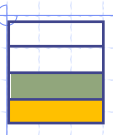
Handling a Page Fault

- When the referenced page is not in DRAM:
  - The missing page is located (or created)
  - It is brought in from disk, and page table is updated

    Another job may be run on the CPU while the first job waits for the requested page to be read from disk
  - If no free pages are left, a page is swapped out

    approximate LRU replacement policy
- Since it takes a long time (msecs) to transfer a page, page faults are handled completely in software (OS)
  - Untranslated addressing mode is essential to allow kernel to access page tables

Swapping a Page of a Page Table



A PTE in primary memory contains primary or secondary memory addresses

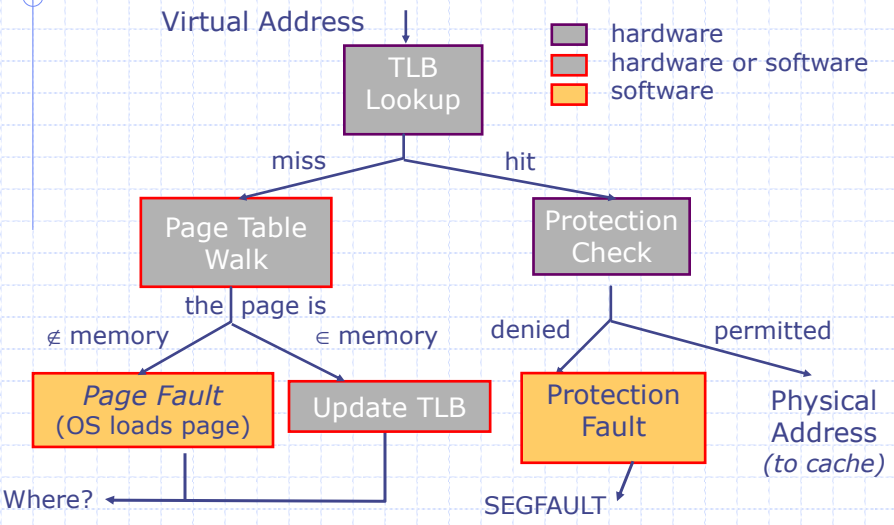
A PTE in secondary memory contains *only* secondary memory addresses

⇒ a page of a PT can be swapped out only if none its PTE's point to pages in the primary memory

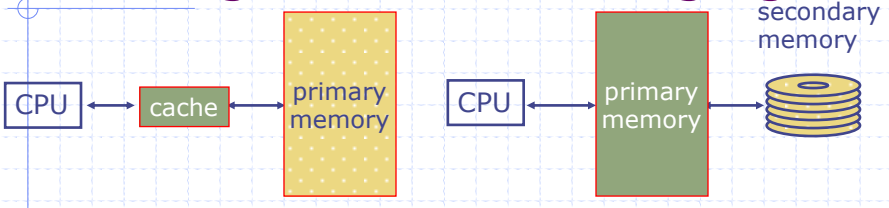
Why?

Don't want to cause a page fault during translation when the data is in memory

Address Translation:
putting it all together



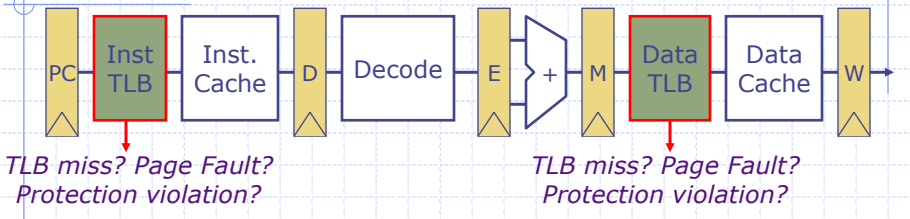
Caching vs. Demand Paging



Caching
- cache entry
- cache block (~32 bytes)
- cache miss rate (1% to 20%)
- cache hit (~1 cycle)
- cache miss (~100 cycles)
- a miss is handled in *hardware*

Demand paging
- page frame
- page (~4K bytes)
- page miss rate (<0.001%)
- page hit (~100 cycles)
- page miss (~5M cycles)
- a miss is handled mostly in *software*

Address Translation in CPU Pipeline

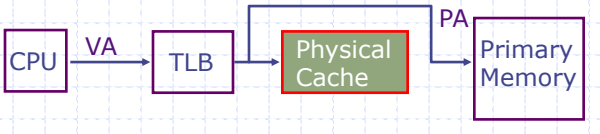


TLB miss? Page Fault? Protection violation?

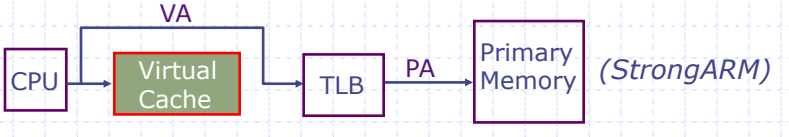
TLB miss? Page Fault? Protection violation?

- Software handlers need a *restartable* exception on page fault or protection violation
- Handling a TLB miss needs a *hardware* or *software* mechanism to refill TLB
- Need mechanisms to cope with the additional latency of a TLB:
  - slow down the clock
  - √ pipeline the TLB and cache access
  - virtual address caches
  - parallel TLB/cache access

Physical or Virtual Address Caches?



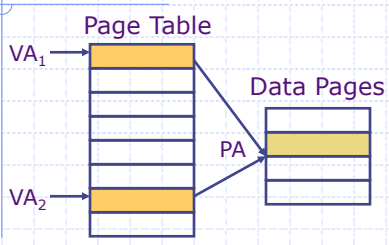
Alternative: place the cache before the TLB



(StrongARM)

- one-step process in case of a hit (+)
- cache needs to be flushed on a context switch unless address space identifiers (ASIDs) included in tags (-)
- *aliasing problems* due to the sharing of pages (-)

Aliasing in Virtual-Address Caches



Two virtual pages share one physical page

Tag	Data
VA_1	1st Copy of Data at PA
VA_2	2nd Copy of Data at PA

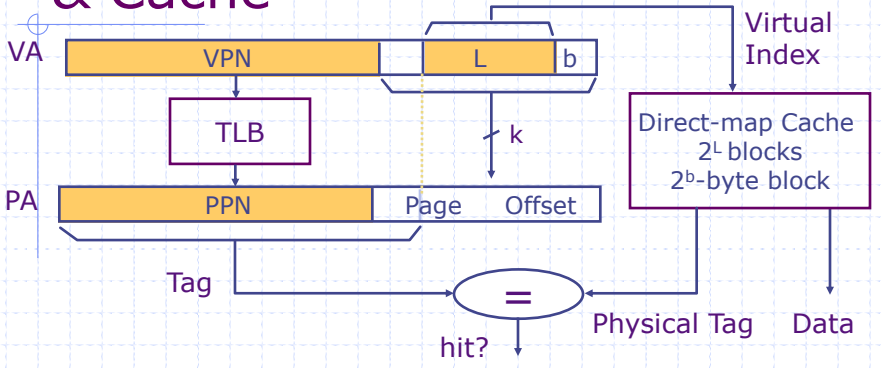
Virtual cache can have two copies of same physical data. Writes to one copy not visible to reads of other!

General Solution: *Disallow aliases to coexist in cache*

Software (i.e., OS) solution for direct-mapped cache

VAs of shared pages must agree in cache index bits; this ensures all VAs accessing same PA will conflict in direct-mapped cache (early SPARCs)

Concurrent Access to TLB & Cache



Index L is available without consulting the TLB

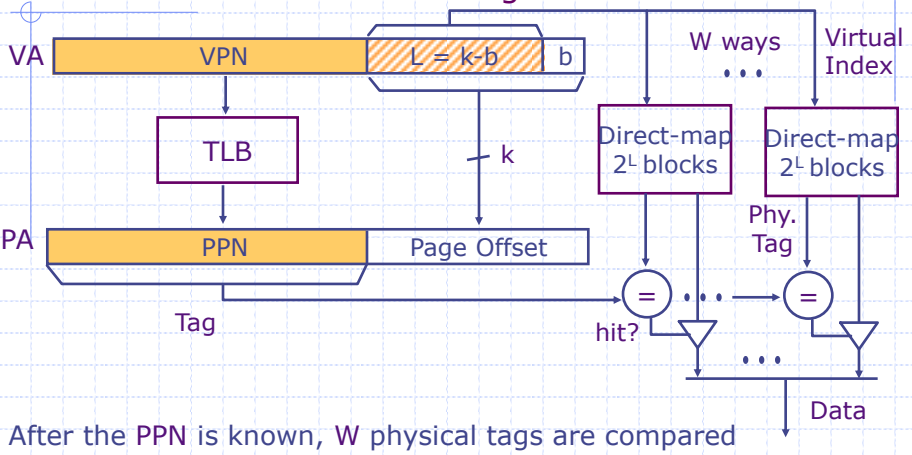
⇒ *cache and TLB accesses can begin simultaneously*

Tag comparison is made after both accesses are completed

Cases: $L + b = k$ $L + b < k$

$L + b > k$ *what happens here?* Partially VA cache!

Virtual-Index Physical-Tag Caches: Associative Organization



After the PPN is known, W physical tags are compared

Allows cache *size* to be greater than 2^{L+b} bytes

We change the cache interface minimally and assume that the Address translation is done as part of the memory system

A memory request will return a 2-tuple <mem-reponse, mException>

Coding is straightforward but we do not have adequate testing infrastructure: requires implementing at least rudimentary TLB-miss and page-fault handlers