COMP3211 03S2 Lecture 12

Virtual Memory

Adapted from

CS152: Computer Architecture and Engineering
Dave Patterson (www.cs.berkeley.edu/~pattrsn)

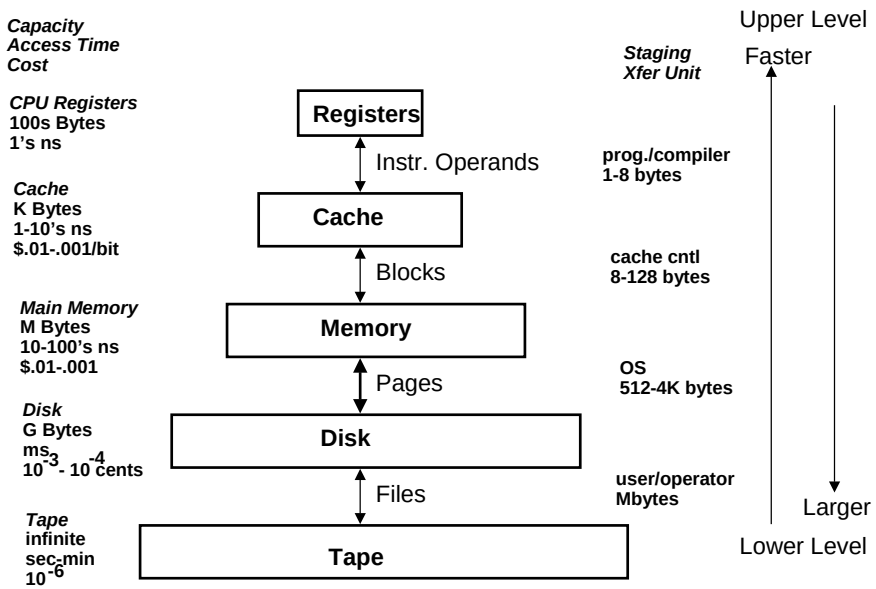
Virtual Memory: Key Ideas (1/2)

1. **Virtual address space, divided into pages, is much larger than physical address space, which is divided into similarly sized blocks known as frames.**
2. **A process, running on the processor refers to data using virtual addresses, which must therefore be translated into physical addresses to access actual memory locations.**
3. **Virtual page numbers (the high order address bits) are translated into physical frame numbers using a page table that is stored in physical memory.**
4. **To speed up translation, a translation lookaside buffer (TLB) – a small associative cache – is used to store recent page-frame translations.**

Virtual Memory: Key Ideas (2/2)

5. **The data cache is indexed/tagged using physical addresses.**
6. **Since TLB access is done using the page number of the virtual address, and since the direct-mapped or n-way set associative D-cache is accessed using the offset of the address within the page, these accesses can be done in parallel.**
7. **Hardware must trap to the operating system if a page is not resident in memory so that it can be loaded from disk – this may result in another page having to be written back to disk.**

Recall: Levels of the Memory Hierarchy



Why virtual memory?

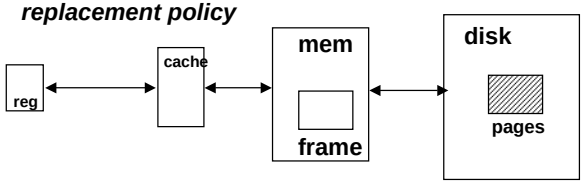
- **Generality**
  - ability to run programs larger than size of physical memory
- **Storage management**
  - allocation/deallocation of variable sized blocks is costly and leads to (external) fragmentation
- **Protection**
  - regions of the address space can be R/O, Ex, . . .
- **Flexibility**
  - portions of a program can be placed anywhere, without relocation
- **Storage efficiency**
  - retain only most important portions of the program in memory
- **Concurrent I/O**
  - execute other processes while loading/dumping page
- **Expandability**
  - can leave room in virtual address space for objects to grow.
- **Performance**
  - Observe: impact of multiprogramming, impact of higher level languages

Recall: 4 Questions for Memory Hierarchy

- Q1: Where can a block be placed in the upper level? *(Block placement)*
- Q2: How is a block found if it is in the upper level? *(Block identification)*
- Q3: Which block should be replaced on a miss? *(Block replacement)*
- Q4: What happens on a write? *(Write strategy)*

Basic Issues in Virtual Memory System Design

- size of information blocks that are transferred from secondary (Disk) to main storage ([M]emory)
- missing item fetched from secondary memory only on the occurrence of a fault → *demand load policy*
- which region of M is to hold the new block → *placement policy*
- block of information brought into M, and M is full, then some region of M must be released to make room for the new block → *replacement policy*



Paging Organization

virtual and physical address space partitioned into blocks of equal size

page frames

pages

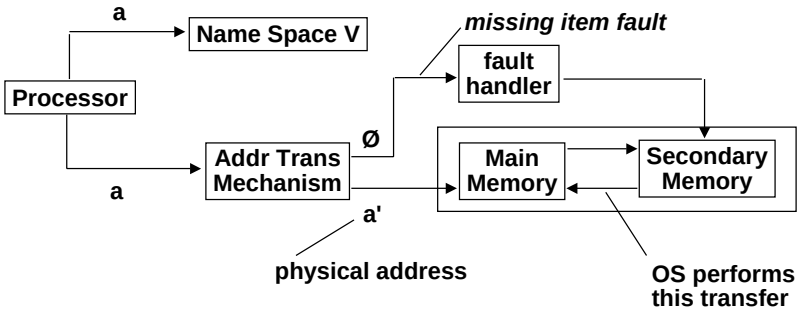
Address Map ([B]lock [I]dentification)

V = {0, 1, . . . , n - 1} virtual address space
M = {0, 1, . . . , m - 1} physical address space, n > m

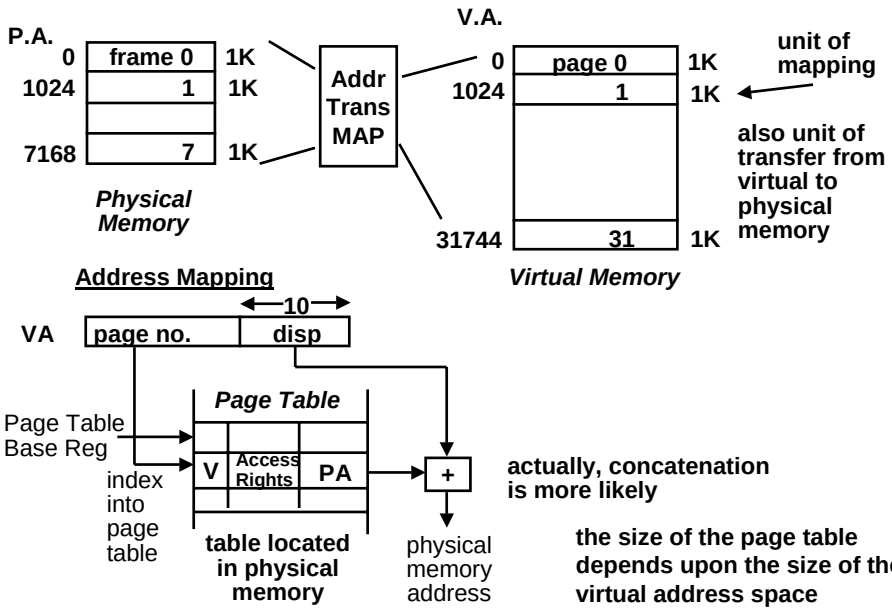
MAP: V → M U {Ø} address mapping function

MAP(a) = a' if data at virtual address a is present in physical address a' and a' in M

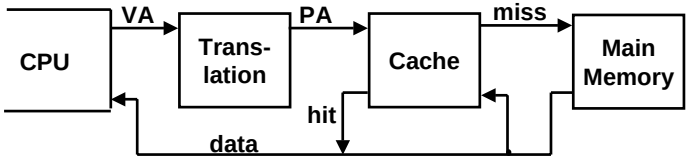
= Ø if data at virtual address a is not present in M



Paging Organization (BI)



Virtual Address and a Cache (BI)

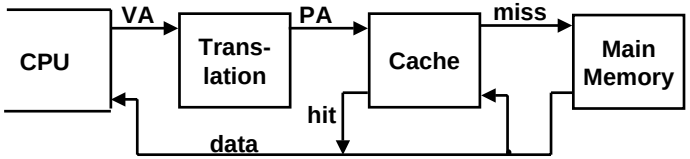


It takes an extra memory access to translate VA to PA

This makes cache access very expensive, and this is the "innermost loop" that you want to go as fast as possible!!

But why access cache with PA at all?

Virtual Address and a Cache (BI)



It takes an extra memory access to translate VA to PA

This makes cache access very expensive, and this is the "innermost loop" that you want to go as fast as possible!!

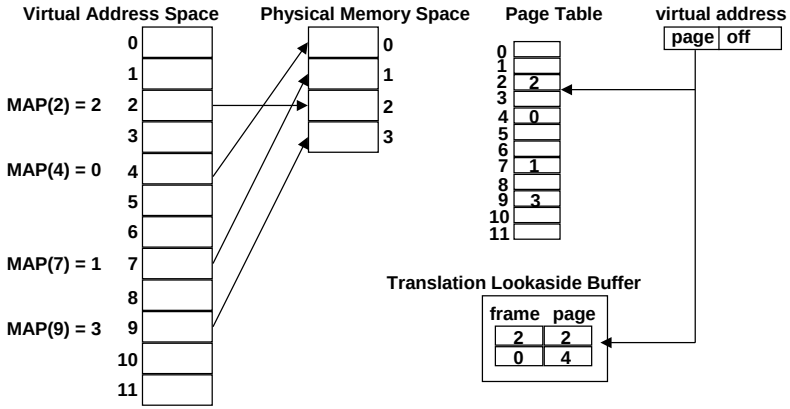
But why access cache with PA at all? VA caches have a problem!

synonym / alias problem: two different virtual addresses map to same physical address => two different cache entries holding data for the same physical address! (This happens when pages are shared between programs but use different virtual addresses to access the one data object)

for update: must update all cache entries with same physical address or memory becomes inconsistent

Making address translation practical: TLB (BI)

- Virtual memory => memory acts like a cache for the disk
- Page table maps virtual page numbers to physical frames
- Translation Look-aside Buffer (TLB) introduced to provide a cache of recent translations
- Benefit: TLB access time comparable to cache access time

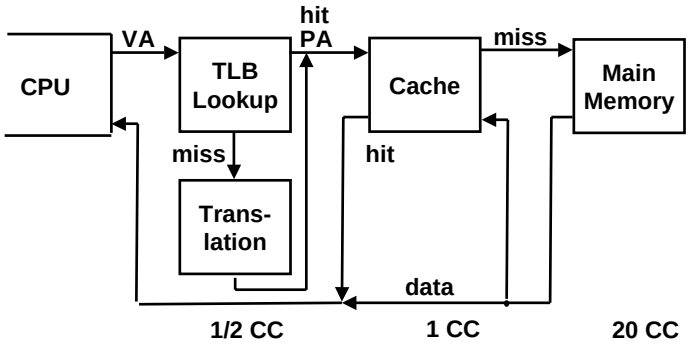


Translation Look-Aside Buffers (BI)

Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped

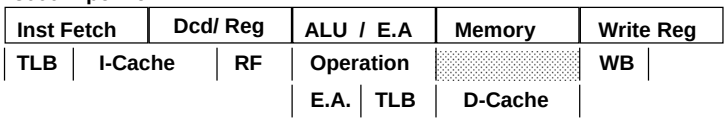
TLBs are usually small, typically not more than 128 - 256 entries even on high-end machines. This permits fully associative lookup on these machines. Most mid-range machines use small n-way set associative organizations.

Translation with a TLB



Constraints on TLB organization (BI)

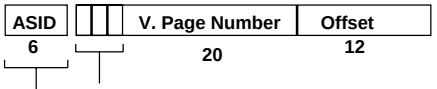
MIPS R3000 Pipeline



TLB

64 entry, on-chip, fully associative, software TLB fault handler

Virtual Address Space



0xx User segment (caching based on PT/TLB entry)
100 Kernel physical space, cached
101 Kernel physical space, uncached
11x Kernel virtual space

Allows context switching among
64 user processes without TLB flush

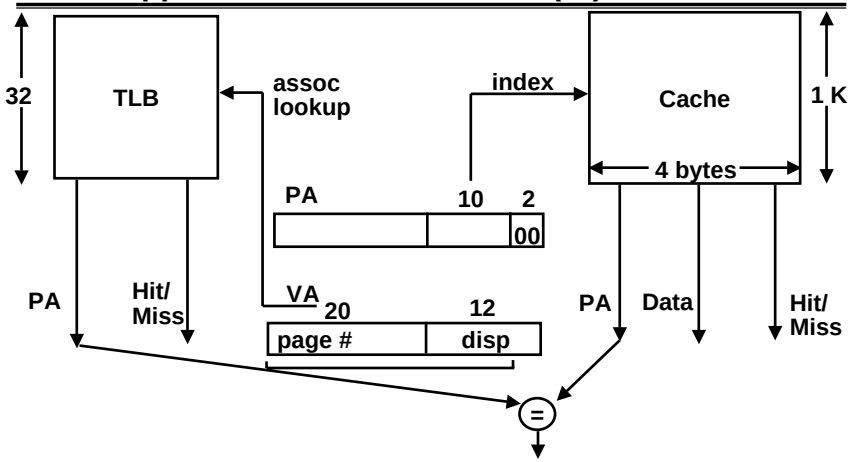
Reducing Translation Time (BI)

Machines with TLBs go one step further to reduce # cycles/cache access

They overlap the cache access with the TLB access

Works because high order bits of the VA are used to look in the TLB while low order bits are used as index into cache

Overlapped Cache & TLB Access (BI)



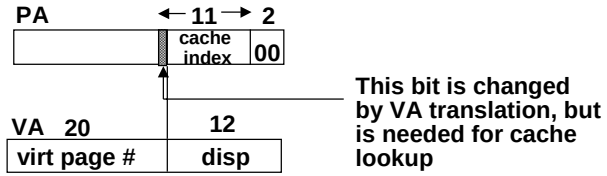
IF cache hit AND (cache tag = PA) THEN deliver data to CPU
ELSE IF cache miss AND TLB hit THEN
access memory with the PA from the TLB
ELSE do standard VA translation

Problems With Overlapped TLB Access (BI)

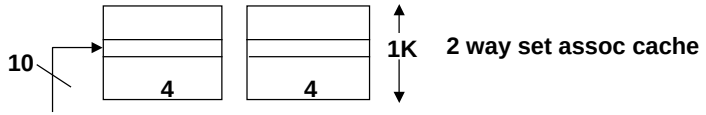
Overlapped access only works as long as the address bits used to index into the cache *do not change* as the result of VA translation

This usually limits things to small caches, large page sizes, or high n-way set associative caches if you want a large cache

Example: suppose everything the same except that the cache is increased to 8 K bytes instead of 4 K:



Solutions:
go to 8K byte page sizes (I.e., extend the offset) & reduce the # of pages;
go to 2 way set associative cache (I.e., increase associativity); or
SW guarantee VA[12]=PA[12]



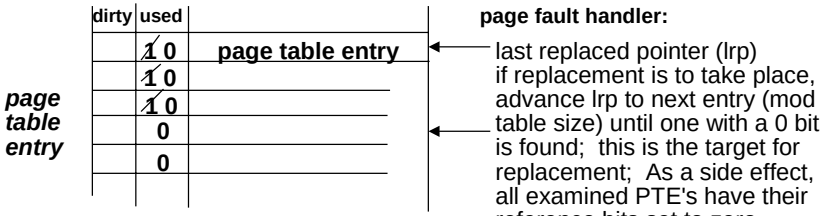
Page Fault: What happens when you miss? ([R]eplacement [P]olicy)

- Not talking about TLB miss
  - TLB is HW's attempt to make page table lookup fast (on average)
- Page fault means that page is not resident in memory (i.e., MAP(a) = ∅)
- Hardware cannot remedy the situation
- Therefore, hardware must trap to the operating system so that it can remedy the situation
  - pick a page to discard (possibly writing it to disk)
  - load the page in from disk
  - update the page table
  - resume to program so HW will retry and succeed!
- What is in the page fault handler?
  - see Operating Systems course
- What can HW do to help it do a good job?

Page Replacement: Not Recently Used (1-bit LRU, Clock) (RP & [W]riteback [P]olicy)

Associated with each page is a reference flag such that
ref flag = 1 if the page has been referenced in recent past
= 0 otherwise

-- if replacement is necessary, choose any page frame such that its reference bit is 0. This is a page that has not been referenced in the recent past



page fault handler:

last replaced pointer (lrp)
if replacement is to take place, advance lrp to next entry (mod table size) until one with a 0 bit is found; this is the target for replacement; As a side effect, all examined PTE's have their reference bits set to zero.

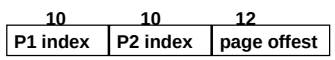
Or search for a page that is both not recently referenced AND not dirty.

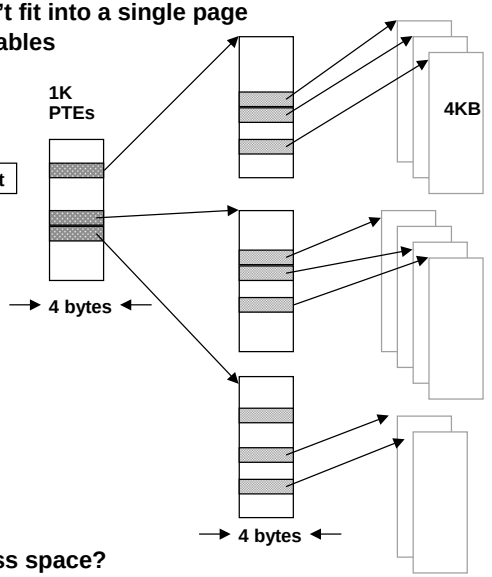
Architecture part: support dirty and used bits in the page table
=> may need to update PTE on any instruction fetch, load, store

Large Address Spaces (BI)

Problem: Page table doesn't fit into a single page
Solution: Two-level Page Tables

32-bit address:





° 4 GB virtual address space
° 4 MB of PTE2
– paged, holes
° 4 KB of PTE1

What about a 48-64 bit address space?

Optimal Page Size

° Minimize wasted storage
- small page minimizes internal fragmentation
- small page increase size of page table

° Minimize transfer time
- large pages (multiple disk sectors) amortize access cost
- sometimes transfer unnecessary info
- sometimes prefetch useful data
- sometimes discards useless data early

° General trend toward larger pages because
- big cheap RAM
- increasing mem / disk performance gap
- larger address spaces

Summary #1/ 4:

° The Principle of Locality:
- Program likely to access a relatively small portion of the address space at any instant in time.
  - Temporal Locality: Locality in Time
  - Spatial Locality: Locality in Space

° Three Major Categories of Cache Misses:
- Compulsory Misses: sad facts of life. Example: cold start misses.
- Conflict Misses: increase cache size and/or associativity. Nightmare Scenario: ping pong effect!
- Capacity Misses: increase cache size

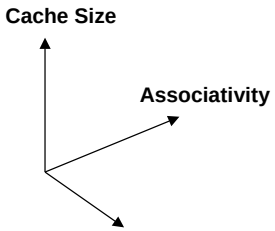
° Cache Design Space
- total size, block size, associativity
- replacement policy
- write-hit policy (write-through, write-back)
- write-miss policy

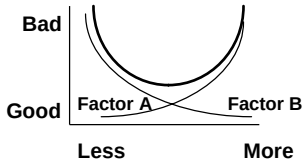
Summary #2 / 4: The Cache Design Space

° Several interacting dimensions
- cache size
- block size
- associativity
- replacement policy
- write-through vs write-back
- write allocation

° The optimal choice is a compromise
- depends on access characteristics
  - workload
  - use (I-cache, D-cache, TLB)
- depends on technology / cost

° Simplicity often wins





Summary #3 / 4 : TLB, Virtual Memory

° Caches, TLBs, Virtual Memory all understood by examining how they deal with 4 questions:
1. Where can block be placed?
2. How is block found?
3. What block is replaced on miss?
4. How are writes handled?

° Page tables map virtual address to physical address

° TLBs are important for fast translation

° TLB misses are significant in processor performance: (most systems can't access all of 2nd level cache without TLB misses!)

Summary #4 / 4: Memory Hierarchy

- Virtual memory was controversial when introduced: can SW automatically manage 64KB across many programs?
  - 1000X DRAM growth removed the controversy
- Today VM allows many processes to share single memory without having to swap all processes to disk; VM protection is more important than memory hierarchy
- Today CPU time is a function of (ops, cache misses) vs. just f(ops): What does this mean for Compilers, Data structures, Algorithms?