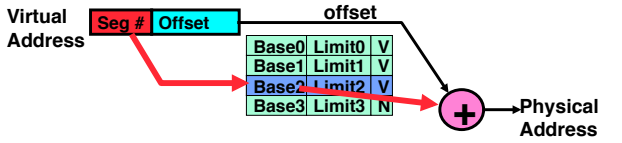
CS162
Operating Systems and Systems Programming
Lecture 10

Caches and TLBs

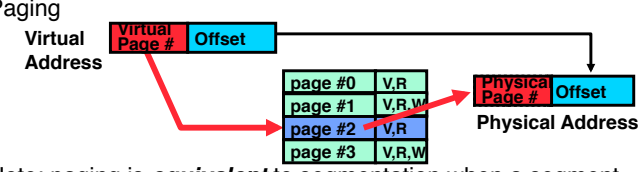
February 22, 2011
Anthony D. Joseph and Ion Stoica
http://inst.eecs.berkeley.edu/~cs162

Recap: Segmentation vs. Paging

- Segmentation:

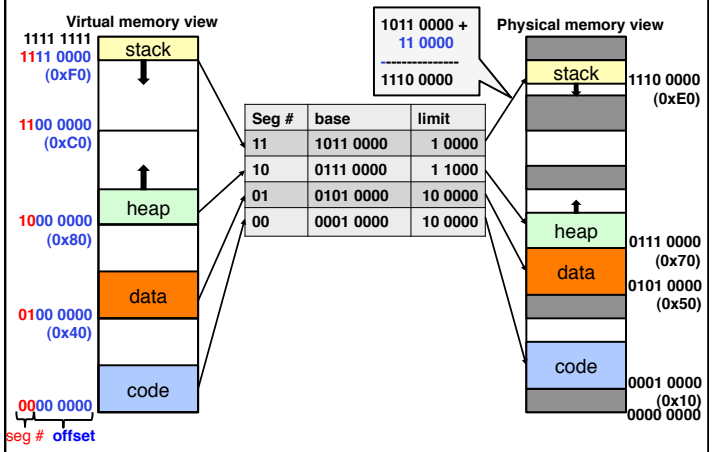


- Paging

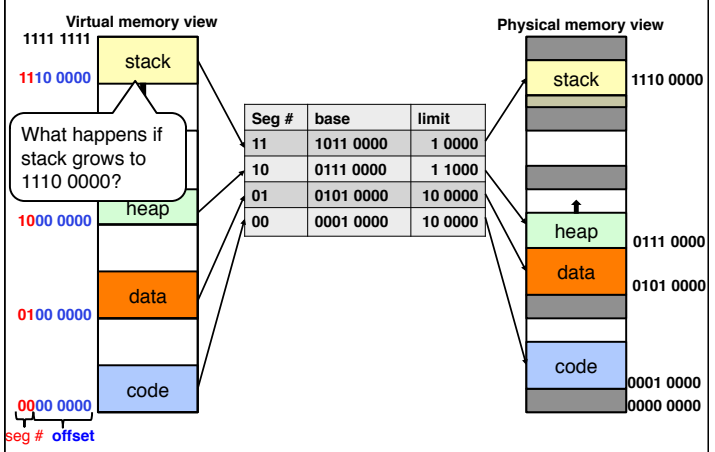


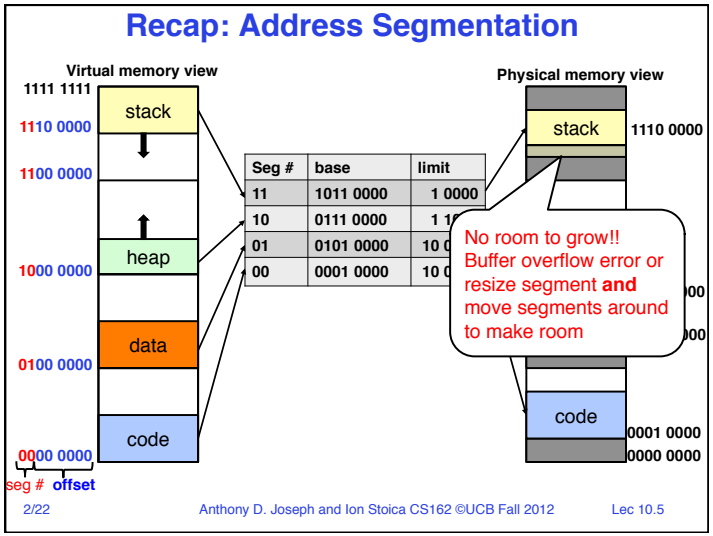
- Note: paging is ***equivalent*** to segmentation when a segment maps onto a page!
  - The offset of the first address in a page is 0

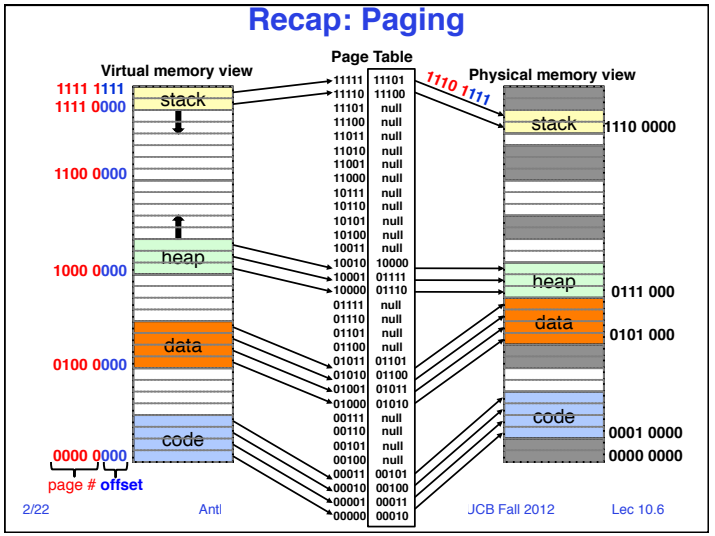
Recap: Address Segmentation

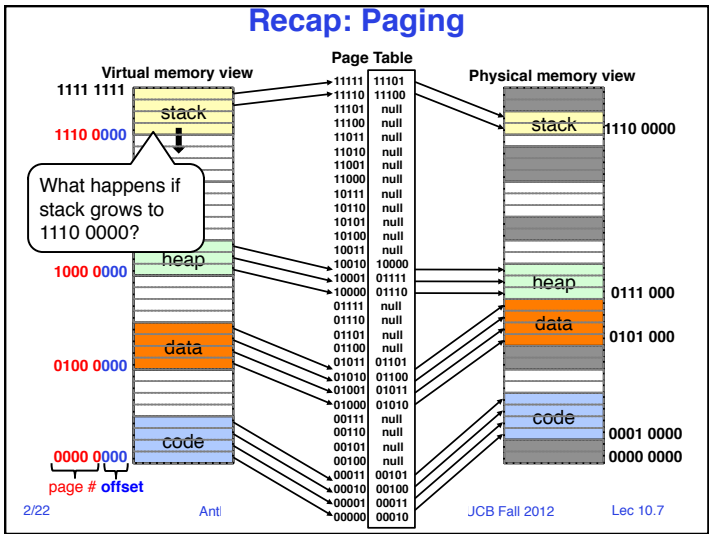


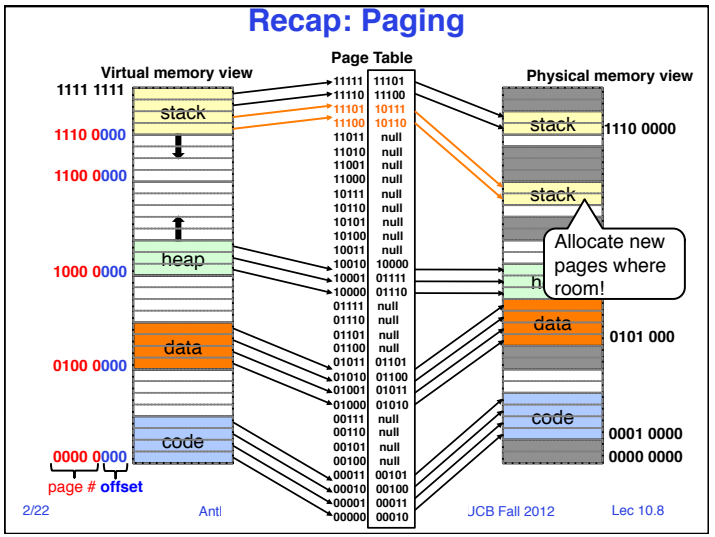
Recap: Address Segmentation



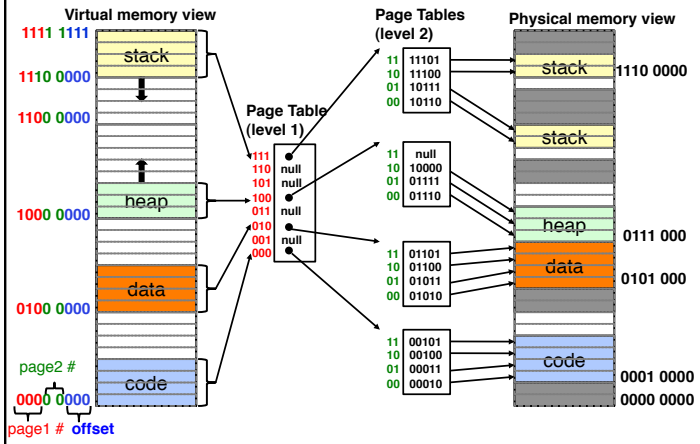

Recap: Address Segmentation
Virtual memory view
1111 1111
1110 0000
1100 0000
1000 0000
0100 0000
0000 0000
stack
heap
data
code
seg # offset
Seg # | base | limit
11 | 1011 0000 | 1 0000
10 | 0111 0000
01 | 0101 0000
00 | 0001 0000
Physical memory view
stack
1110 0000
code
0001 0000
0000 0000
No room to grow!! Buffer overflow error or resize segment and move segments around to make room
2/22
Anthony D. Joseph and Ion Stoica CS162 ©UCB Fall 2012
Lec 10.5


Recap: Paging
Virtual memory view
Page Table
Physical memory view
1111 1111
1111 0000
1100 0000
1000 0000
0100 0000
0000 0000
stack
heap
data
code
page # offset
1110 1111
1110 0000
0111 000
0101 000
0001 0000
0000 0000
2/22
Anti
JCB Fall 2012
Lec 10.6

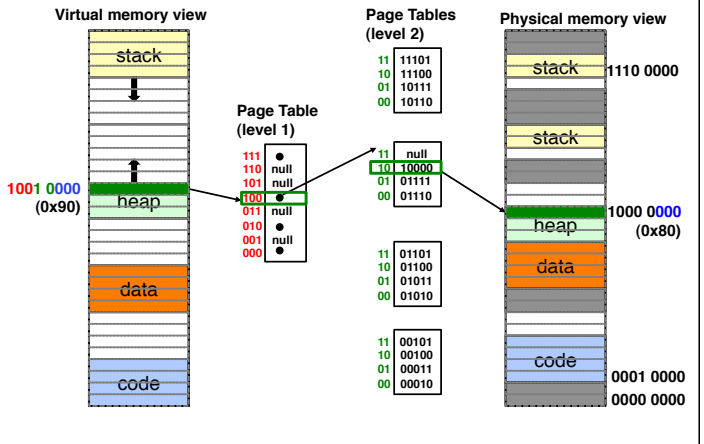

Recap: Paging
Virtual memory view
Page Table
Physical memory view
What happens if stack grows to 1110 0000?
stack
heap
data
code
page # offset
1110 0000
0111 000
0101 000
0001 0000
0000 0000
2/22
Anti
JCB Fall 2012
Lec 10.7


Recap: Paging
Virtual memory view
Page Table
Physical memory view
Allocate new pages where room!
stack
heap
data
code
page # offset
1110 0000
0101 000
0001 0000
0000 0000
2/22
Anti
JCB Fall 2012
Lec 10.8

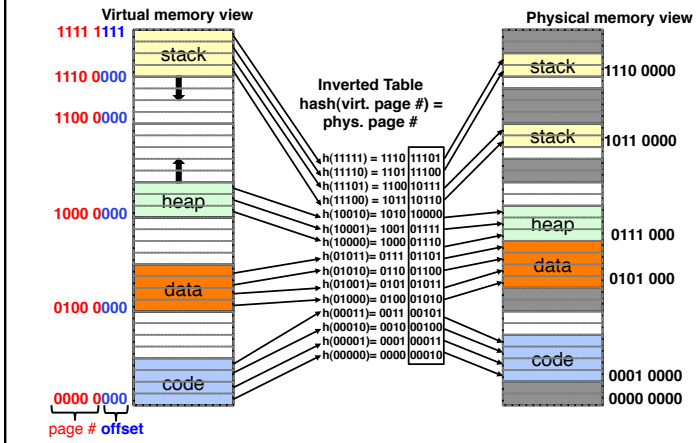
Recap: Two-Level Paging



Recap: Two-Level Paging



Recap: Inverted Table



Address Translation Comparison

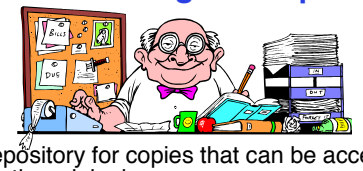
	Advantages	Disadvantages
Segmentation	Fast context switching: Segment mapping maintained by CPU	External fragmentation
Paging (single-level page)	No external fragmentation	Large table size ~ virtual memory
Paged segmentation	Table size ~ # of pages in physical memory	Multiple memory references per page access
Two-level pages		
Inverted Table	Table size ~ # of pages in physical memory	Hash function more complex

Goals for Rest of Today's Lecture

- Caching
  - Misses
  - Organization
- Translation Look aside Buffers (TLBs)

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from lecture notes by Kubiatowicz.

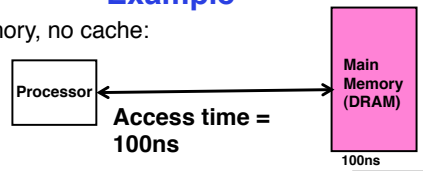
Caching Concept



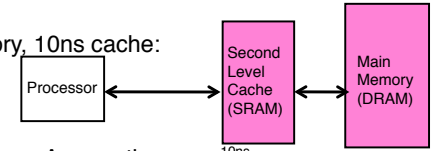
- Cache: a repository for copies that can be accessed more quickly than the original
  - Make frequent case fast and infrequent case less dominant
- Caching at different levels
  - Can cache: memory locations, address translations, pages, file blocks, file names, network routes, etc…
- Only good if:
  - Frequent case frequent enough and
  - Infrequent case not too expensive
- Important measure: Average Access time =
  (Hit Rate x Hit Time) + (Miss Rate x Miss Time)

Example

- Data in memory, no cache:



- Data in memory, 10ns cache:

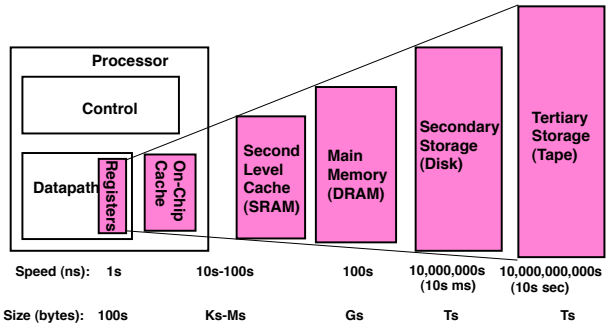


Average Access time =
(Hit Rate x HitTime) + (Miss Rate x MissTime)

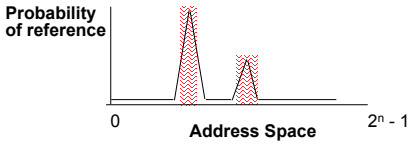
- HitRate + MissRate = 1
- HitRate = 90% → Average Access Time = 19ns
- HitRate = 99% → Average Access Time = 10.9ns

Review: Memory Hierarchy

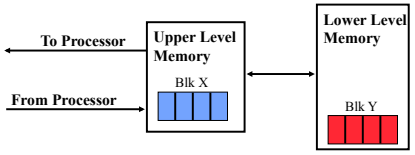
- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology



Why Does Caching Help? Locality!



- Temporal Locality (Locality in Time):
  - Keep recently accessed data items closer to processor
- Spatial Locality (Locality in Space):
  - Move contiguous blocks to the upper levels

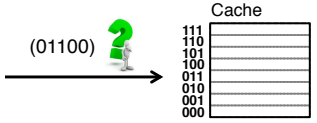


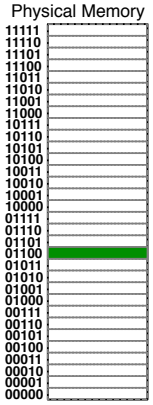
Sources of Cache Misses

- Compulsory (cold start): first reference to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: When running "billions" of instruction, Compulsory Misses are insignificant
- Capacity:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- Conflict (collision):
  - Multiple memory locations mapped to same cache location
  - Solutions: increase cache size, or increase associativity
- Two others:
  - Coherence (Invalidation): other process (e.g., I/O) updates memory
  - Policy: Due to non-optimal replacement policy

Caching Questions

- 8 byte cache
- 32 byte memory
- 1 block = 1 byte
- Assume CPU accesses 01100

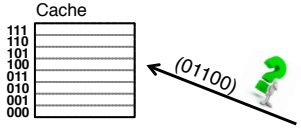


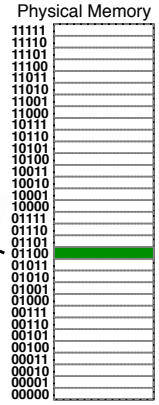


1. How do you know whether byte @ 01100 is cached?

Caching Questions

- 8 byte cache
- 32 byte memory
- 1 block = 1 byte
- Assume CPU accesses 01100

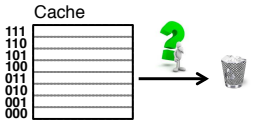


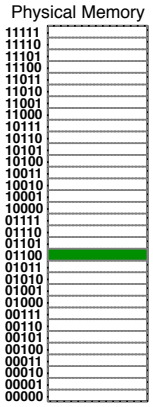


1. How do you know whether byte @ 01100 is cached?
2. If not, at which location in the cache do you place the byte?

Caching Questions

- 8 byte cache
- 32 byte memory
- 1 block = 1 byte
- Assume CPU accesses 01100

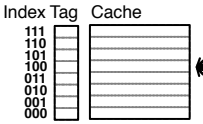


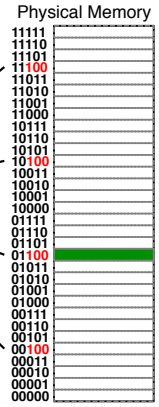


1. How do you know whether byte @ 01100 is cached?
2. If not, at which location in the cache do you place the byte?
3. If cache full, which cached byte do you evict?

Simple Example: Direct Mapped Cache

- Each byte (block) in physical memory is cached to a single cache location
  - Least significant bits of address (last 3 bits) index the cache
  - (00100),(01100),(10100),(11100) cached to 100

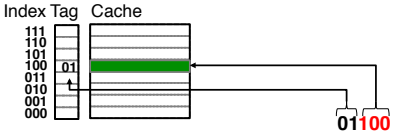


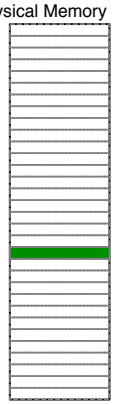


- How do you know which byte is cached?
  - Cache stores the most significant two bits (i.e., **tag**) of the cached byte

Simple Example: Direct Mapped Cache

- Each byte (block) in physical memory is cached to a single cache location
  - Least significant bits of address (last 3 bits) index the cache
  - (00100),(01100),(10100),(11100) cached to 100

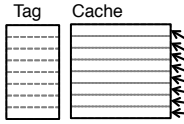


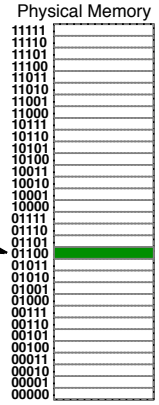


1. How do you know whether (01100) is cached?
   - Check **tag** associated with index 100
2. At which cache location do you place (01100)?
   - 100
3. If cache full, which cached byte do you evict?
   - 100

Simple Example: Fully Associative Cache

- Each byte can be stored at any location in the cache

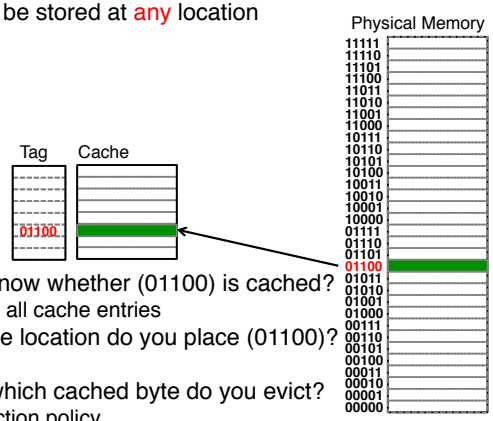




- How do you know which byte is cached?
  - Tag store entire address of cached byte

Simple Example: Fully Associative Cache

- Each byte can be stored at any location in the cache



1. How do you know whether (01100) is cached?
   - Check **tag** of all cache entries
2. At which cache location do you place (01100)?
   - Any
3. If cache full, which cached byte do you evict?
   - Specific eviction policy

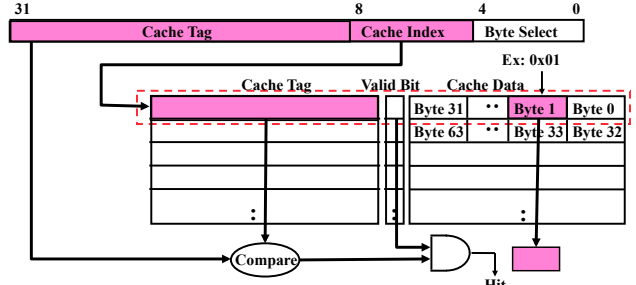
Announcements

- Evaluations and final design documents are due today @ 11:59pm
  - Evaluation anonymous, so please be as open as possible in your comments → it would help us to catch group issues early
- Piazza is part of your class participation
  - We will also make a conscious effort to improve our own response time
- New upgraded scripts for autograder
  - It should be more robust from now on
- Please fill the course survey at https://www.surveymonkey.com/s/DZ5Y8XM
  - It's anonymous, so please be open!

5min Break

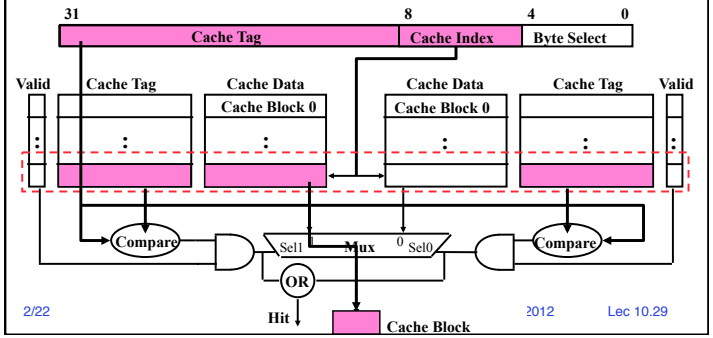
Direct Mapped Cache

- Cache index selects a cache block
- "Byte select" selects byte within cache block
  - Example: Block Size=32B blocks
- Cache tag fully identifies the cached data
- Data with same "cache index" shares the same cache entry
  - Conflict misses



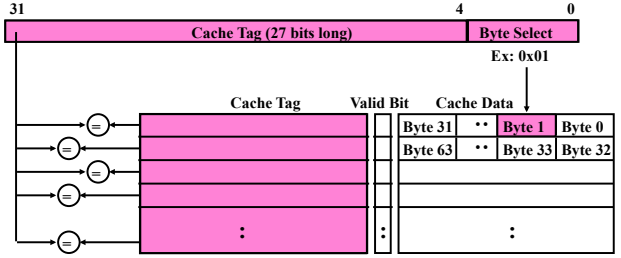
Set Associative Cache

- N-way set associative: N entries per Cache Index
  - N direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - Two tags in the set are compared to input in parallel
  - Data is selected based on the tag result



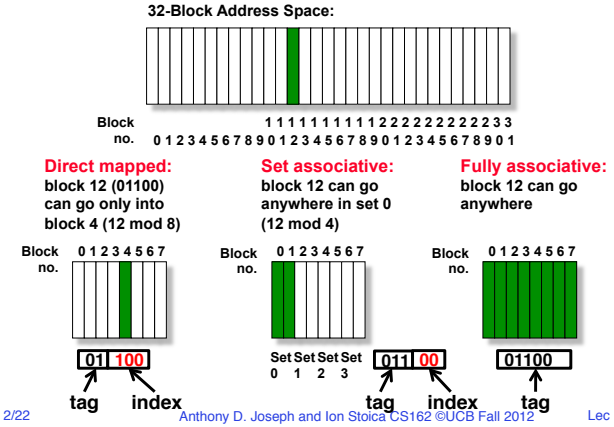
Fully Associative Cache

- Fully Associative: Every block can hold any line
  - Address does not include a cache index
  - Compare Cache Tags of all Cache Entries in Parallel
- Example: Block Size=32B blocks
  - We need N 27-bit comparators
  - Still have byte select to choose from within block



Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache



Which block should be replaced on a miss?

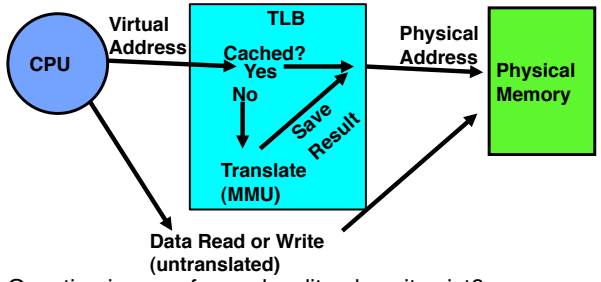
- Easy for Direct Mapped: Only one possibility
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

	2-way		4-way		8-way	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

What happens on a write?

- Write through: The information is written both to the block in the cache and to the block in the lower-level memory
- Write back: The information is written only to the block in the cache.
  - Modified cache block is written to main memory only when it is replaced
  - Question is block clean or dirty?
- Pros and Cons of each?
  - WT:
    - » PRO: read misses cannot result in writes
    - » CON: processor held up on writes unless writes buffered
  - WB:
    - » PRO: repeated writes not sent to DRAM
      processor not held up on writes
    - » CON: More complex
      Read miss may require writeback of dirty data

Caching Applied to Address Translation



- Question is one of page locality: does it exist?
  - Instruction accesses spend a lot of time on the same page (since accesses sequential)
  - Stack accesses have definite locality of reference
  - Data accesses have less page locality, but still some…
- Can we have a TLB hierarchy?
  - Sure: multiple levels at different sizes/speeds

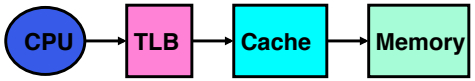
What Actually Happens on a TLB Miss?

- Hardware traversed page tables:
  - On TLB miss, hardware in MMU looks at current page table to fill TLB (may walk multiple levels)
    - » If PTE valid, hardware fills TLB and processor never knows
    - » If PTE marked as invalid, causes Page Fault, after which kernel decides what to do afterwards
- Software traversed Page tables
  - On TLB miss, processor receives TLB fault
  - Kernel traverses page table to find PTE
    - » If PTE valid, fills TLB and returns from fault
    - » If PTE marked as invalid, internally calls Page Fault handler
- Most chip sets provide hardware traversal
  - Modern operating systems tend to have more TLB faults since they use translation for many things

What happens on a Context Switch?

- Need to do something, since TLBs map virtual addresses to physical addresses
  - Address Space just changed, so TLB entries no longer valid!
- Options?
  - Invalidate TLB: simple but might be expensive
    - » What if switching frequently between processes?
  - Include ProcessID in TLB
    - » This is an architectural solution: needs hardware
- What if translation tables change?
  - For example, to move page from memory to disk or vice versa…
  - Must invalidate TLB entry!
    - » Otherwise, might think that page is still in memory!

What TLB organization makes sense?



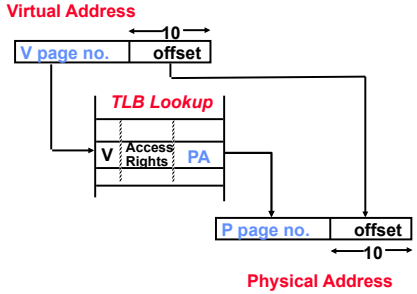
- Needs to be really fast
  - Critical path of memory access
  - Seems to argue for Direct Mapped or Low Associativity
- However, needs to have very few conflicts!
  - With TLB, the Miss Time extremely high!
  - This argues that cost of Conflict (Miss Time) is much higher than slightly increased cost of access (Hit Time)
- Thrashing: continuous conflicts between accesses
  - What if use low order bits of page as index into TLB?
    - » First page of code, data, stack may map to same entry
    - » Need 3-way associativity at least?
  - What if use high order bits as index?
    - » TLB mostly unused for small programs

TLB organization: include protection

- How big does TLB actually have to be?
  - Usually small: 128-512 entries
  - Not very big, can support higher associativity
- TLB usually organized as fully-associative cache
  - Lookup is by Virtual Address
  - Returns Physical Address + other info
- What happens when fully-associative is too slow?
  - Put a small (4-16 entry) direct-mapped cache in front
  - Called a "TLB Slice"
- When does TLB lookup occur?
  - Before cache lookup?
  - In parallel with cache lookup?

Reducing translation time further

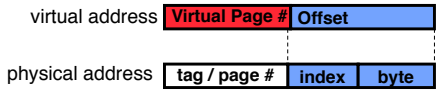
- As described, TLB lookup is in serial with cache lookup:



- Machines with TLBs go one step further: they overlap TLB lookup with cache access.
  - Works because offset available early

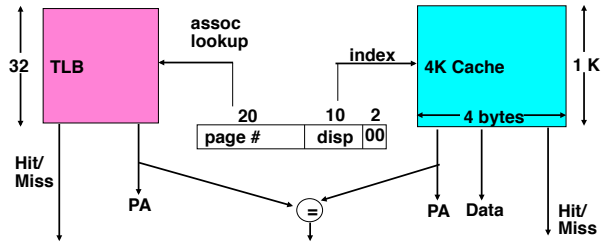
Overlapping TLB & Cache Access (1/2)

- Main idea:
  - Offset in virtual address exactly covers the "cache index" and "byte select"
  - Thus can select the cached byte(s) in parallel to perform address translation



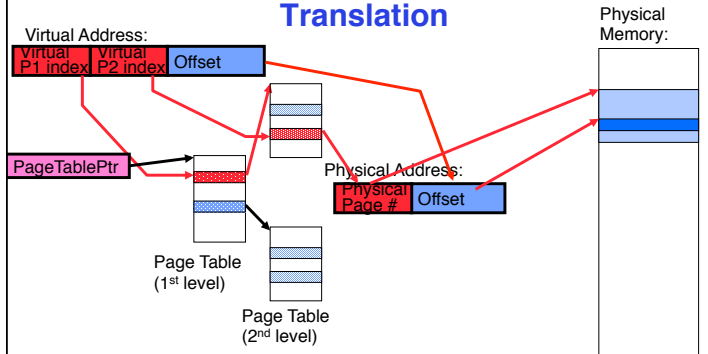
Overlapping TLB & Cache Access (1/2)

- Here is how this might work with a 4K cache:

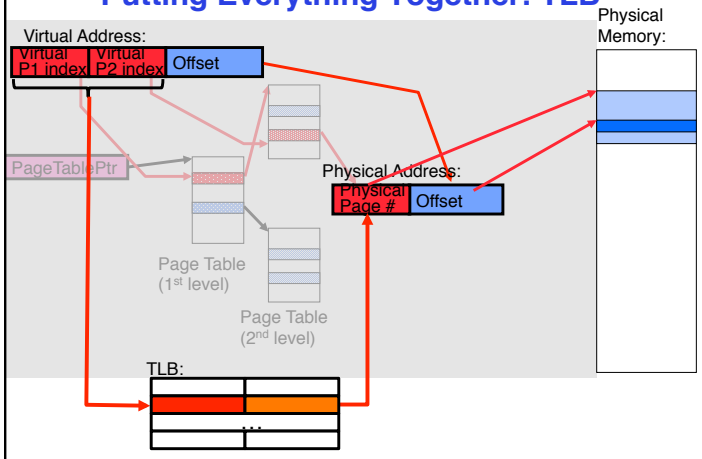


- What if cache size is increased to 8KB?
  - Overlap not complete
  - Need to do something else. See CS152/252

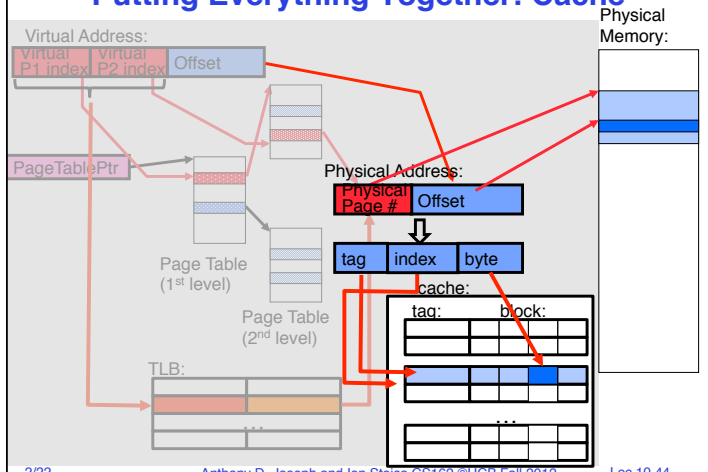
Putting Everything Together: Address Translation



Putting Everything Together: TLB



Putting Everything Together: Cache



Summary (1/2)

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time.
    - » Temporal Locality: Locality in Time
    - » Spatial Locality: Locality in Space
- Three (+1) Major Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Conflict Misses: increase cache size and/or associativity
  - Capacity Misses: increase cache size
  - Coherence Misses: Caused by external processors or I/O devices

Summary (2/2)

- Cache Organizations:
  - Direct Mapped: single block per set
  - Set associative: more than one block per set
  - Fully associative: all entries equivalent
- TLB is cache on address translations
  - Fully associative to reduce conflicts
  - Can be overlapped with cache access