CS 162 Operating Systems and Systems Programming

Professor: Anthony D. Joseph

Spring 2001

Lecture 14: Caching and TLBs

14.0 Main Points:

- Cache concept, in general and as applied to translation
- Ways of organizing caches – associativity
- Problems with caching

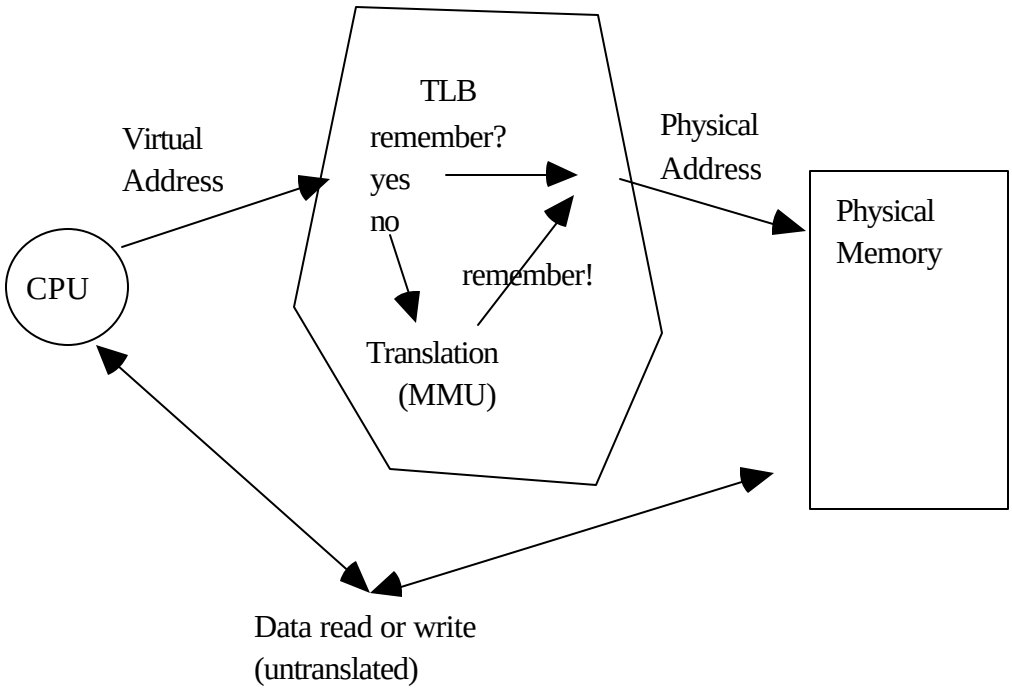
14.1 Cache concept

Cache: copy that can be accessed more quickly than original.

Idea is: make frequent case efficient, infrequent path doesn't matter as much. Caching is a fundamental concept used in lots of places in computer systems. It underlies many of the techniques that are used today to make computers go fast: can cache translations, memory locations, pages, file blocks, file names, network routes, authorizations for security systems, etc.

14.2 Caching applied to address translation

Address translation is on the critical path for every instruction --- can't afford to always do a memory look up (or even worse, an I/O!) to find a page table entry.
Often reference same page repeatedly, why go through entire translation each time?



Translation Buffer, Translation Lookaside Buffer: hardware table of frequently used translations, to avoid having to go through page table lookup in common case. Typically, on chip, so access time of 5 – 10ns, instead of several hundred of ns for main memory.

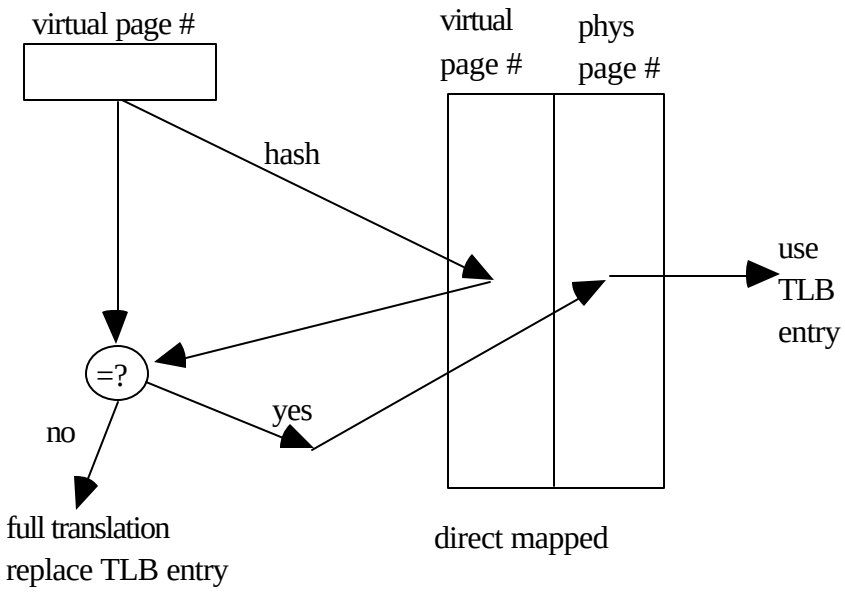
TLB for example from previous lecture (simple paging)

virtual page #	physical page #	control bits
2	1	valid, rw
-	-	invalid
0	4	valid, rw

14.2.1 How do we tell if needed translation is in TLB?

1. Search table in sequential order

2. **Direct mapped:** restrict each virtual page to use specific slot in TLB



As a (bad, as we will see shortly) example, could use the lower bits of virtual page number to index TLB. Compare against the upper bits of virtual page number to check for match.

Note: Do we need to store the entire virtual page number in the TLB? No, as an enhancement we could get away with storing only the upper bits.

Consider a 256 entry TLB, and the following reference stream of virtual page numbers in hex (note these are just page numbers, not entire virtual addresses):

0x621
0x2145
0x621
0x2145
...
0x321
0x2145
0x321
0x621

What happens to the TLB?

What if two pages conflict for the same TLB slot? Ex: program counter and stack.

Thrashing: cache contents tossed out even if still needed

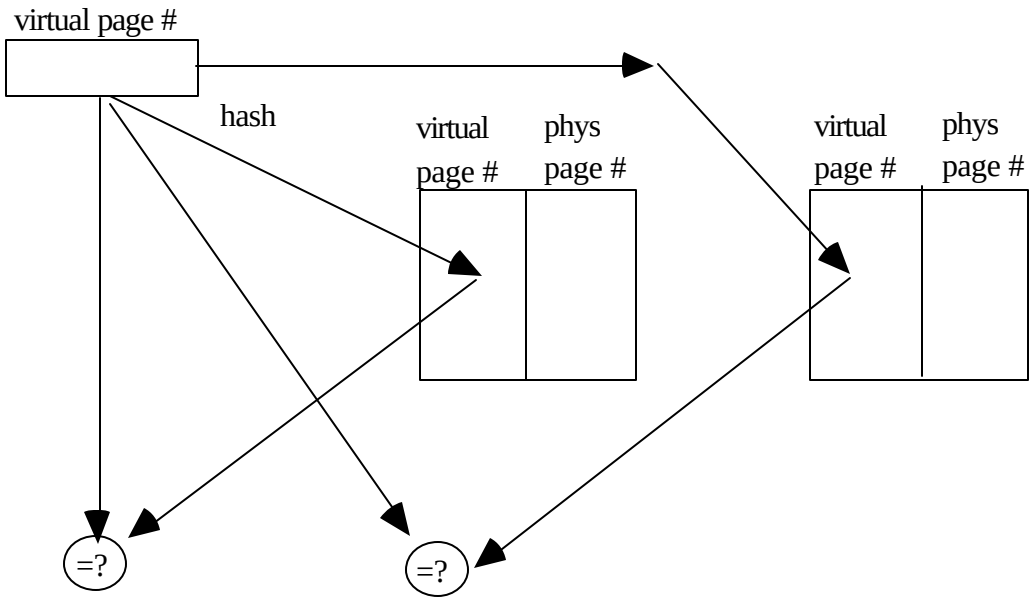
One approach: pick hash function to minimize conflicts

What if use low order bits as index into TLB?

What if use high order bits as index into TLB?

Thus, use selection of high order and low order bits as index.

3. **Set associativity**: arrange TLB (or cache) as N separate banks. Do simultaneous lookup in each bank. In this case, called "N-way set associative cache".

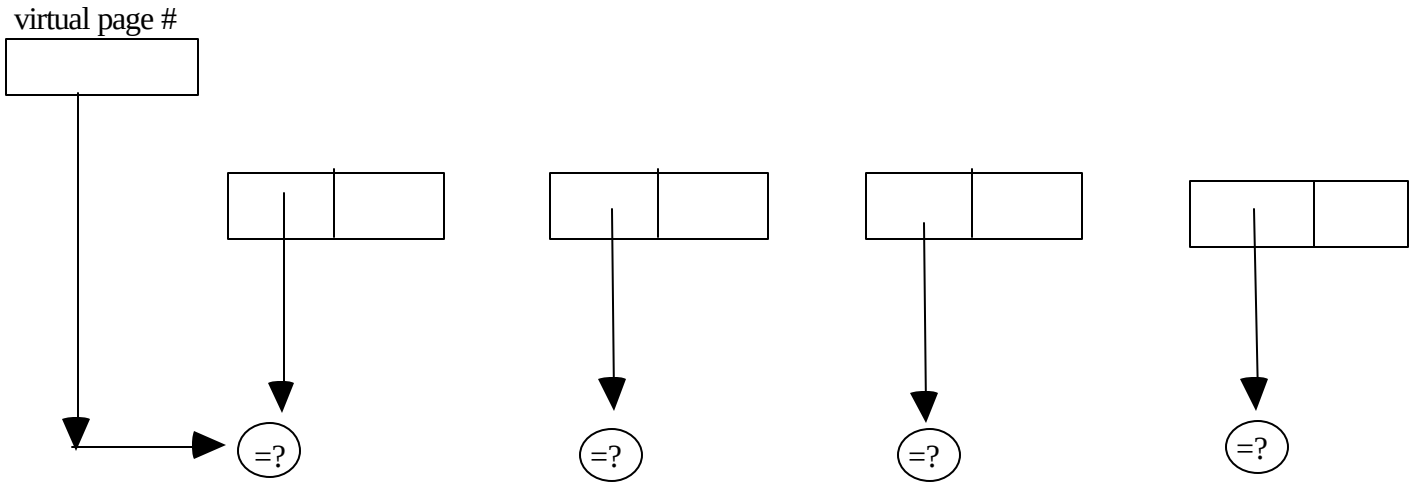


if either match, use TLB entry, otherwise,
translate and replace one of the entries.

Two-way set associative

More set associativity, less chance of thrashing. Translations can be stored, replaced in either bank.

4. **Fully associative**: translation can be stored anywhere in TLB, so check all entries in the TLB in parallel.



if any match, use that TLB entry, otherwise,
translate and replace one of the entries.

Fully associative TLB

One element per bank, one comparator per bank. So parallelism is obtained through the addition of compartor hardware.

Same set of options, whether you are building TLB or any kind of cache. Typically, TLB's are small and fully associative. Hardware caches are larger, and direct mapped or set associative to a small degree.

14.2.2 How do we choose which item to replace?

For direct mapped, never any choice as to which item to replace. But for set associative or fully associative cache, have a choice. What should we do?

Replace least recently used? Random? Most recently used? Defer until next lecture topic. In hardware, often choose item to replace randomly, because it's simple and fast.

In software (for example, for page replacement), typically do something more sophisticated. Tradeoff: spend CPU cycles to try to improve cache hit rate.

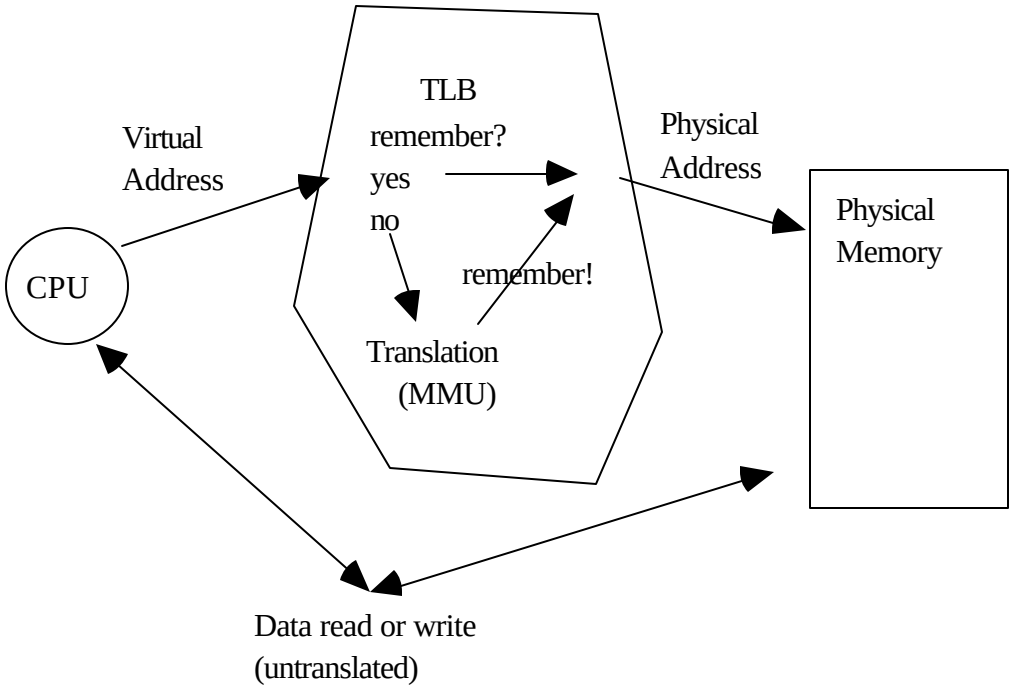
14.2.3 Consistency between TLB and page tables

What happens on context switch?

Have to invalidate entire TLB contents. When new program starts running, will bring in new translations.
Alternatively, include process id tag in TLB comparator. Have to keep TLB consistent with whatever the full translation would give.

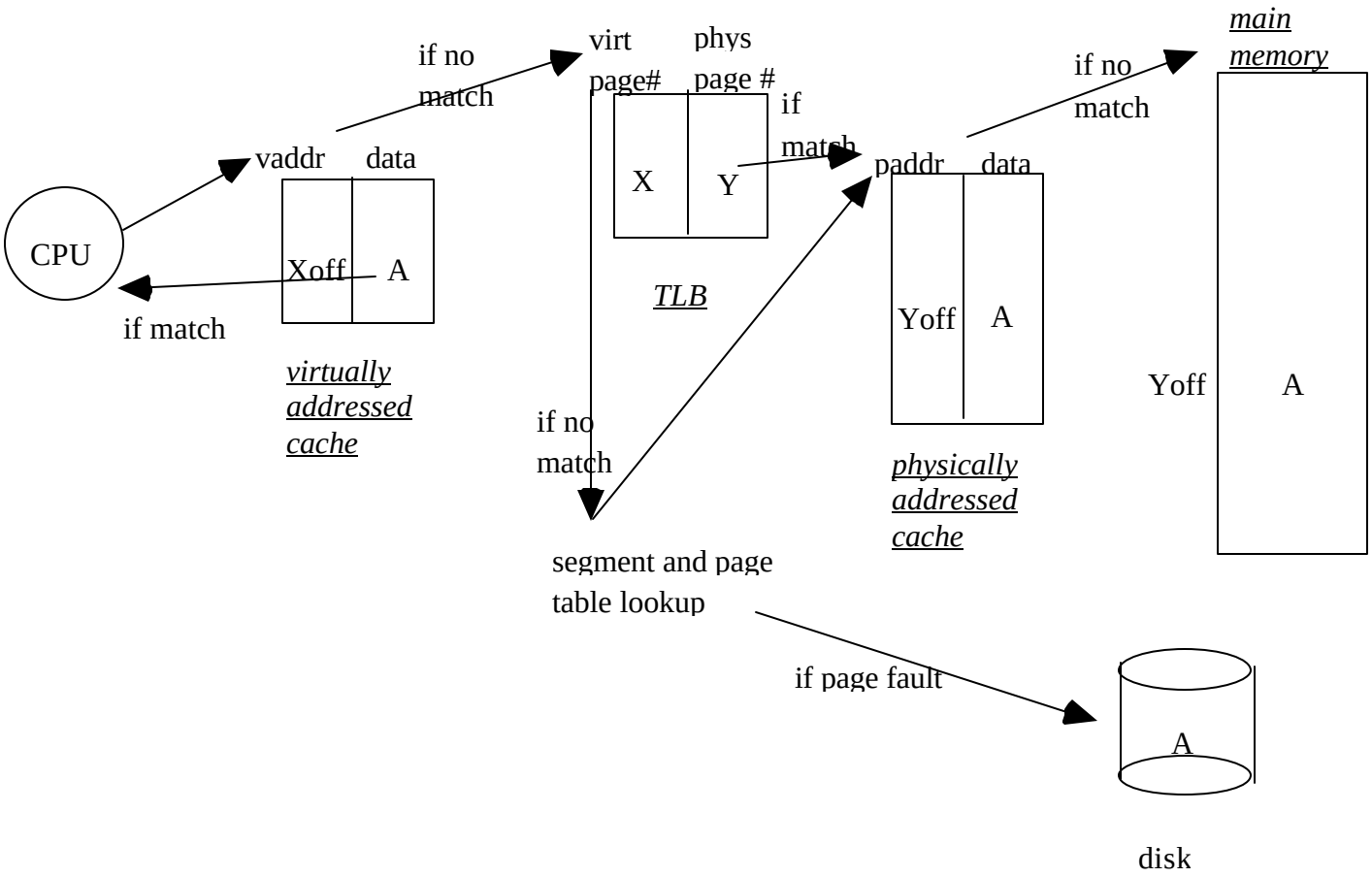
What if translation tables change? For example, to move page from memory to disk, or vice versa. Have to invalidate TLB entry.

14.3 Relationship between TLB and hardware memory caches



Can put a cache of memory values anywhere in this process.

If between translation box and memory, called a "physically addressed cache". Could also put a cache between CPU and translation box: "virtually addressed cache".



Virtual memory is a kind of caching: we're going to talk about using the contents of main memory as a cache for disk.

What about writes? What if CPU modifies a location?

Two options:

- **write-through**: update immediately sent through to next level in memory hierarchy
- **write-back**: (delayed write-through) update kept until item is replaced from cache, then sent to next level.

Since write-back is faster, memory caches typically use write-back; file systems, which need to worry about whether the data being written will survive a machine crash, tend to use write-through.

14.4 Memory Hierarchy

Two principles:

1. The smaller amount of memory needed, the faster that memory can be accessed.
2. The larger amount of memory, the cheaper per byte.

Thus, put frequently accessed stuff in small, fast, expensive memory; use large, slow, cheap memory for everything else.

	Latency	Size	Cost
registers	2.5ns	32-128 bytes	on chip
on-chip cache	5ns	8KB	on chip
off-chip cache	25ns	256KB	$2000/MB
main memory	200ns	256MB	$4/MB
Disk	10ms (10M ns)	30 GB	$0.01MB
robotic tape	10s (10B ns)	100 TB	factor of 3-5 less than disk.

Use caching at each level, to provide illusion of a terabyte, with register access times.

Works because programs aren't random. Exploit **locality**: that computers behave in future like they have in the past.

Temporal locality: will reference same locations as accessed in the recent past

Spatial locality: will reference locations near those accessed in the recent past

14.5 Generic Issues in Caching

Cache hit: item is in the cache
Cache miss: item is not in the cache, have to do full operation

Effective access time =

P(hit) * cost of hit + P(miss) * cost of miss

Can divide cache misses into one of four categories:

1. **Compulsory**: first reference to data will always miss, even if you assume infinite cache
2. **Capacity**: non-compulsory misses due to limited size cache, assuming fully associative and optimal replacement policy
3. **Conflict**: non-compulsory, non-capacity misses, due to limited associativity, assuming optimal replacement policy
4. **Policy**: misses due to non-optimal replacement policy

1. How do you find whether item is in the cache (whether there is a cache hit)?
2. If it is not in cache (cache miss), how do you choose what to replace from cache to make room? (Replacement policy)
3. Consistency – how do you keep cache copy consistent with real version?

14.6 When does caching break down?

Whenever programs don't exhibit enough spatial or temporal locality – what if loop through array that doesn't fit in the cache?

Example: When MIPS was first introduced: 64 item TLB, 4 KB page size. Plus, 1K x 1K x 4 byte video RAM. Video RAM was mapped into virtual memory, to allow graphics programs to write directly to display. What happens?