Lecture#16: VM, thrashing, Replacement, Cache state

****

Review -- 1 min

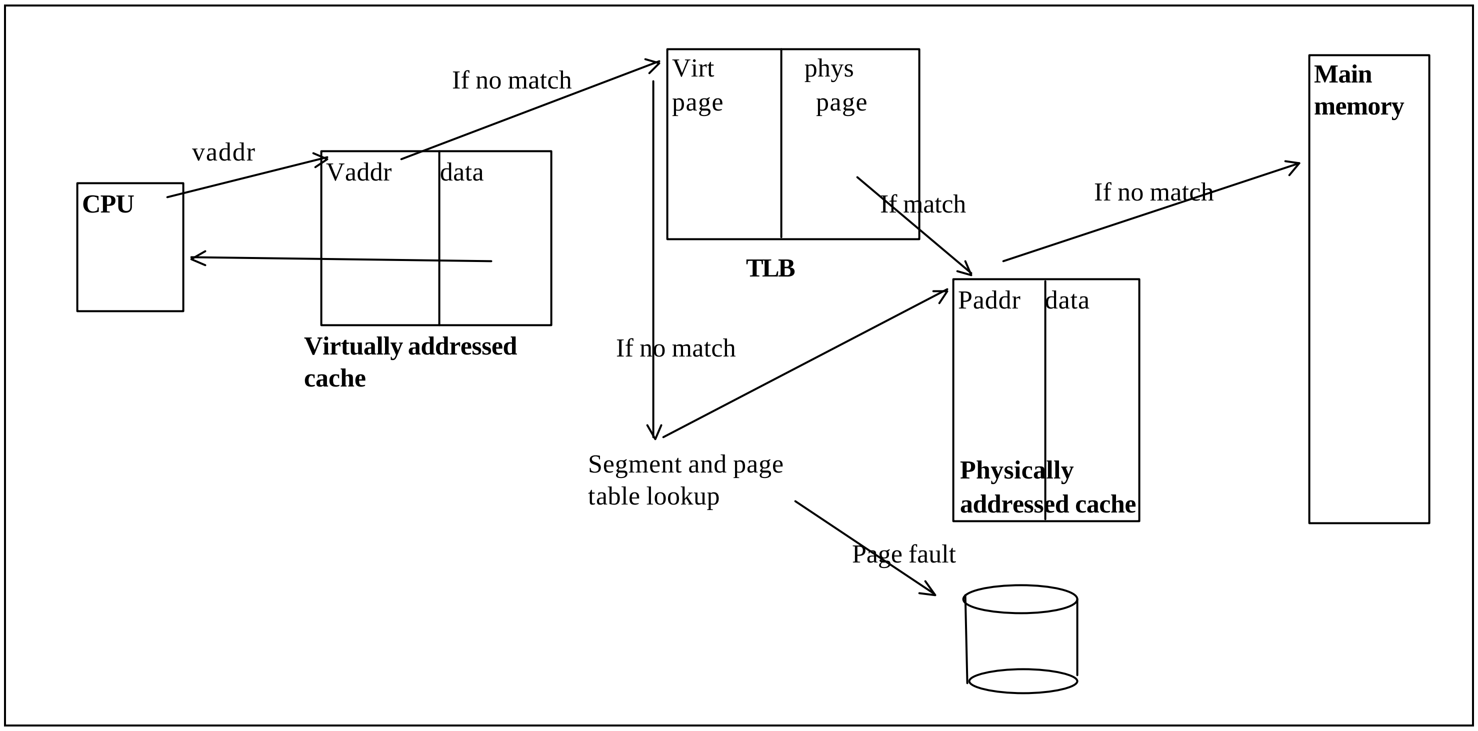
****

Multi-level translation

- tree: multi-level page table, paged paging, paged segmentation, …
- hash table: inverted page table
- combination: hashed page table

Paging to disk

- simple mechanism (once we have everything else)



AMAT (average memory access time)
AMAT = T_L1 + P_L1miss * T_L1miss
T_L1Miss = T_TLB + P_TLB_miss * T_TLB_miss
+ T_L2 + P_L2miss * T_mem
T_TLB_miss = # references * T_L2 + P_L2miss * T_mem
T_mem = T_DRAM + P_DRAM_Miss * T_Disk

T_L1: a few cycles
T_L2: ~O(10) cycles

T_DRAM: ~O(100 cycles)
T_TLB_miss: ~O(100-1000cycles)
T_Disk: ~O(10^6 cycles)

****
Outline - 1 min
****
Paging to disk:

- writing and sharing – dirty bit and core map
- performance

Replacement
State bits

****
Preview - 1 min
****
I/O – file systems

****
Lecture - 35 min

[finish thrashing, working set, swapping]

****
Admin - 3 min

HW 3 available
Project 2 due ~~Monday Feb 16~~ Wednesday Feb 18

****
Lecture - 35 min
****

1. Cache replacement policies

TLB – fully associative – can replace any entry on a miss
hardware → random
Virtual memory cache – replace any page

software → more flexibility
high miss penalty → very important to maximize hit rate

Replacement policy is an issue for any caching system

1.1 Random

Typical solution for TLBs. Easy to implement in HW

1.2 FIFO

Throw out oldest page. Be fair – let every page live in memory for the same amount of time, then toss it.

Bad because throws out heavily used pages instead of those that are not frequently used

1.3 MIN (aka OPT)

Replace page that won't be used for the longest time into the future

MIN is optimal – you can prove that no replacement policy has fewer misses than MIN.

What's the problem with MIN?

Principle: Useful to identify optimal policy even if it is unrealistic to implement

- **Useful comparison**. Can often simulate it → useful for comparison studies (policy X is 1% worse than OPT → probably not worth while to spend a lot of time optimizing X further)
- **Improves understanding**. Provides insight to fundamental problem and requirements on solution → helps you implement simple, effective, practical algorithm

1.4 LRU

Replace page that hasn't been used for the longest time

If induction works, LRU is a good approximation to MIN. Actually, people don't use even LRU, they approximate it.

1.5 Example

Suppose we have 3 page frames and 4 virtual pages with the reference string ABCABDADBCB (virtual page references)

1.5.1 FIFO

reference phys slot	A	B	C	A	B	D	A	D	B	C	B
1	A			√		D		√		C	
2		B			√		A				
3			C						B		√

1.5.2 MIN

reference phys slot	A	B	C	A	B	D	A	D	B	C	B
1	A			√			√			C	
2		B			√				√		√
3			C			D		√			

1.5.3 LRU

Same as MIN for this pattern
Won't always be this way

QUESTION: When will LRU perform badly?
When next reference is to the least recently used page

Reference string ABCDABCDABCD
LRU

reference phys slot	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	

3			C			B			A			D

Same behavior with FIFO! What about MIN?

MIN

reference phys slot	A	B	C	D	A	B	C	D	A	B	C	D
1	A				√				√	B		
2		B				√	C				√	
3			C	D				√				√

1.5.4 Does adding memory always reduce the number of page faults?

Yes for LRU, MIN

No for FIFO (Belady’s anomoly)

reference phys slot	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					√
2		B			A			√		C		
3			C			B			√		D	

reference phys slot	A	B	C	D	A	B	E	A	B	C	D	E
1	A				√		E				D	
2		B				√		A				E
3			C						B			
4				D						C		

With FIFO, contents of memory can be completely different with different number of page frames

By contrast, with LRU or MIN, contents of memory with X pages is subset of contents with X+1 pages. So with LRU or MIN, having more pages never hurts. **Stack property**

2. Implementing LRU

2.1 Perfect

Timestamp page on each reference
Keep list of pages ordered by time of reference

On every memory reference – move page to front of LRU list
Too much work per mem reference

2.2 Clock

Approximate LRU (approx to approx of MIN)

Replace **an** old page, not **the** oldest page

Clock algorithm: arrange physical pages in a circle, with a clock hand

1. Keep a **use bit** per physical page (usually in coremap)
2. Set use bit when page brought into TLB

If bit isn't set, means not referenced for a long time

3. On page fault

    Advance clock hand (not real time)
    check use bit
    1 → clear, go on
    2 → replace page

Will it always find a page or loop indefinitely?
Even if all use bits are set, it will eventually loop around, clearing all use bits → FIFO

What if hand is moving slowly?
Not many page faults and/or find page quickly

What if hand is moving quickly?
Lots of page faults and/or lots of reference bits set

One way to view clock: crude partitioning of pages into two groups – young and old. Why not partition into more than 2 groups?

2.3 Nth chance

Nth chance algorithm – don't throw page out until hand has swept by N times

OS keeps counter per page -- # sweeps

On page fault, OS checks use bit
1→ clear use and also clear counter, go on
0 → increment counter; if < N go on else replace page

How do we pick N?

Large N → better approximate LRU
Small N → more efficient; otherwise might have to look a long way to find a free page

Dirty pages have to be written back to disk when replaced. Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing.

Common approach (e.g., Solaris)
- clean pages – use N = 1
- dirty pages – use N = 2 (and write back when N = 1)

3. State per page table entry

To summarize, many machines maintain 4 bits per page table entry

use – set when page referenced; cleared by clock algorithm
modified – set when page is modified; cleared when page written to disk
valid – OK for program to reference this page
read-only OK for program to read page, but not to modify it (e.g. don't allow modification of code page)

3.1 Do we really need a HW "modified" bit?

Suppose you are given a machine w/ where the TLB doesn't have a "modified" bit. How can you tell which pages are dirty?

Can emulate modified bit (e.g. BSD UNIX -- HW page tables). Keep two sets of books:
(i) pages user program can access w/o taking fault
(ii) pages in memory

Set I is a subset of set ii.

SW-loaded TLB case:
When loading TLB , if page is clean, mark it "read only". On write, trap to OS. OS sees that SW page table allows writes → set SW page table modified bit and update TLB entry as read/write

HW-loaded TLB/HW page table case:
Keep 2 page tables: HW page table and SW page table. Initially, mark all pages as read-only in HW page table. On write, trap to OS. OS sets (SWpt) modified bit and marks (HWpt) page as read-write. When page comes back in from disk (clean), mark HWpt as read-only

3.2 Do we really need a "use" bit?

No. Can emulate it, exactly the same as above

e.g., HW page table:
(i) Mark all pages as invalid, even if in memory
(ii) On read to invalid page, trap to OS
(iii) OS sets use bit, and marks page read-only
(iv) on write, set use and modified bit, and mark page read-write
(v) when clock hand passes by, reset use bit and mark page as invalid

But remember, clock is just approximation of LRU
Can we do better approximation, given that we have to take page fault on some reads and writes to collect use information? Need to identify an old page, not the oldest page!

VAX/VMS didn't have a use or modify bit, so had to come up with some solution.
Idea was to split memory into two parts – mapped and unmapped

i) directly accessible to program (marked as read-write) (managed FIFO)
ii) second-chance list (marked as invalid, but in memory) (managed pure LRU)

On page reference

```
    if mapped, access at full speed
    otherwise page fault:
    if on second chance list, mark read-write
        move first page on FIFO list onto end of second chance
        list (and mark invalid)
    if not on second chance list, bring into memory
    move first page on FIFO list onto end of second chance
    replace first page on second chance list
```

How many pages for second chance list?

    If 0, FIFO
    if all, LRU, but page fault on every page reference

Pick intermediate value
Result:
+ few disk accesses (page only goes to disk if it is unused for along time)
■ increase overhead trapping to OS (sw/hw tradeoff)

3.3 Does sw-loaded TLB need a use bit?

Two options:
1) hardware sets use bit in TLB; when TLB entry is replaced, sofware copies use bit back to page table
2) Sofware manages TLB entries as FIFO list; everything not in TLB is second chance list, managed as strict LRU

4. Fairness

On a page fault, do you consider all pages in one pool or only those of the process that caused the page fault

Global replacement (UNIX) – all pages in one pool

More flexible – if my process needs a lot, and you need a little, I can grab pages from you. Problem – one turkey can ruin whole system (want to favor jobs that need only a few pages!)

Per-process (VMS) – give each a separate pool; for example, a separate clock for each process. Less flexible

Example:
  intermittent interactive job (emacs)
  batch job (compilation)

When compilation is over, emacs page have to be brought back in, and no history information.

****

Summary: Virtual memory

****

Goals of virtual memory:

- Protection, sharing → control bits in TLB, page table
- Relocation → translation VA→PA in TLB, page table
- illusion of infinite memory → paging to disk
- minimal overhead → HW support (TLB) + tree, hash
  - space
  - time