Lecture#17: Replacement, Cache state

Review -- 1 min

Virtual memory – paging to disk
  Provide illusion of infinite
  Use essentially same mechanisms as already had
  Page table
  Core map
  Pageout daemon
  Danger: thrashing

Outline - 1 min

Thrashing -- solutions
Replacement
State bits

Preview - 1 min

Finish virtual memory
I/O – file systems

1. Thrashing
   Thrashing – memory overcommitted – pages tossed out while still needed
Example – one program touches 50 pages (each equally likely); only have 40 physical page frames
If have enough pages – 200ns/ref
If have too few pages – assume every 5th reference \(\rightarrow\) page fault
4 refs x 200ns
1 page fault x 10ms for disk I/O
\(\rightarrow\) 5 refs per 10ms + 800ns = 2ms/ref = \textbf{20000x slowdown}!!!

QUESTION: what hit rate do you need to get less than a 1% slowdown from paging to disk?

1.1 \textbf{Problem: system doesn’t know what it is getting in to}

Log more and more users into system – eventually:
   \hspace{1cm} total number of pages needed \(>\) number of pages available

Picture: jobs/sec v. total system throughput

So, what do you do about this?
1) One process alone too big?
   Change program so it needs less memory or has better locality
   For example, split matrix multiply into smaller sub-matrices that each fit in memory

2) Several jobs?
   \hspace{1cm} figure out needs/process (working set)
   \hspace{1cm} run only groups that fit (balance sets) – kick other processes out of memory
   \hspace{1cm} e.g., suppose you are paging and running at a 10,000x slowdown,
   \hspace{1cm} if kicking half of the jobs out would get you to stop paging and run at full speed, you trade a 1.5x slowdown for a 10,000x slowdown

Remember -- issue here is not total size of process, but rather total number of pages being used at the moment.

How do we figure needs/process out?
1.1.1 Working set (denning, MIT mid 60’s)

Informally – collection of pages a process is using right now

Formally – set of pages job has referenced in last T seconds

How do we pick T?

1 page fault = 10ms
10ms = 2M instructions
→ T needs to be a lot bigger than 1 million instructions

How do you figure out what working set is?

■ replacement policy keeps track of time to last access – use information from it (later in lecture/next lecture)

a) modify clock algorithm so that it seeps at fixed intervals (keep idle time/page; how many seconds since last reference)
b) with second chance list – how many seconds since got put on 2\textsuperscript{nd} chance list

Now you know how many pages each program needs. What do you do?

\textbf{Balance set}

1) if all fits? Done
2) if not? Throw out fat cats; bring them back eventually

What if T too big?
→ waste memory – too few programs fit in memory
What if T too small?
→ thrashing

2. Swapping v. paging

If system low on memory, may be better off moving an entire process to disk and stopping it from running for a while
System stop thrashing -> system runs more efficiently → average (and perhaps all) jobs run faster

2.1.1 Need two levels of scheduling
Upper level decides on swapping
- when
- who
- for how long
Lower levels decide who on ready queue actually runs on CPU

Upper level invoked when there are processes swapped out and whenever we need to load more programs than can fit in main memory
3. Cache replacement policies

TLB – fully associative – can replace any entry on a miss
hardware \( \rightarrow \) random
Virtual memory cache – replace any page
software \( \rightarrow \) more flexibility

Replacement policy is an issue for any caching system

3.1 Random

Typical solution for TLBs. Easy to implement in HW

3.2 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

3.3 MIN

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

3.4 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.
3.5 Example

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

3.5.1 FIFO

<table>
<thead>
<tr>
<th>reference phys slot</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>D</th>
<th>B</th>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>C</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

3.5.2 MIN

<table>
<thead>
<tr>
<th>reference phys slot</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>D</th>
<th>B</th>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.5.3 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

<table>
<thead>
<tr>
<th>reference phys slot</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td>D</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td>A</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>D</td>
<td>A</td>
<td>D</td>
<td>B</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Same behavior with FIFO! What about MIN?

**MIN**

<table>
<thead>
<tr>
<th>reference phys slot</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Yes for LRU, MIN
No for FIFO (Belady’s anomaly)

<table>
<thead>
<tr>
<th>reference phys slot</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>E</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td>D</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td>A</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>B</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>reference phys slot</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>E</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td>E</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td>A</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>D</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

4. Implementing LRU

4.1 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

4.2 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. Hardware keeps a use bit per physical page
2. Hardware sets use bit on each reference (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 $\rightarrow$ clear, go on
   2 $\rightarrow$ 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 $\rightarrow$ 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?

4.3 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
  clean pages – use N = 1
  dirty pages – use N = 2 (and write back when N = 1)

5. 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)

5.1 Do we really need a “modified” bit?
   No. Can emulate it (e.g. BSD UNIX). 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. Initially, mark all pages as read-only. On write, trap to OS. OS sets (SW) modified bit and marks page as read-write

When page comes back in from disk, mark as read-only

5.2 Do we really need a “use” bit?
No. Can emulate it, exactly the same as above

(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 list
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)

5.3 Does sw-loaded TLB need a use bit?

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

5.4 Core map
   page tables map virtual page # \(\rightarrow\) physical page #

Do we need the reverse? Physical page # \(\rightarrow\) virtual page #?

Yes. Clock algorithm runs through page frames. What if it ran through
   page tables
   (i) many more entries
   (ii) what if there is sharing? (e.g., what do you do if a shared page
       becomes “dirty”?

6. 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.