21 Lecture: More VM

Outline:
Announcements
From last time: Virtual Memory (Fotheringham, 1961)
  What it’s all about
Translation Logistics(HERE)
  Where to keep the page table?
  Possible approaches to dealing with size
Translation Lookaside Buffers
What about really big memories?
Limited memory size
Inverted page tables
Page replacement algorithms
  Optimal
  Random
  Not Recently Used
  FIFO
  Second Chance
  Clock (optimized Second Chance)
LRU
  Not Frequently Used

21.1 Announcements

• Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed</td>
<td>Jun 5 23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri</td>
<td>Jun 7 23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat</td>
<td>Jun 8 10:10</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

• Remember, your SecretKeepers shouldn’t interact with stdin or stdout (stderr, is ok for error messages). They only interact with the filesystem through messages.

21.2 From last time: Virtual Memory (Fotheringham, 1961)

transparent It’s there, but you can’t see it.

virtual You can see it, but it isn’t there.

21.2.1 What it’s all about

The way it works

• Programs exist in a virtual address space

• Virtual memory is divided into pages

• Physical memory is divided into frames
• Non-resident pages are stored in backing store (swap space) and brought in as needed.

• Virtual addresses are translated into physical addresses by the MMU (Memory Management Unit) using page tables

• Page tables contain:
  – virtual page number (index)
  – cache enabled (C)
  – modified bit (M)
  – referenced bit (R)
  – protection bits (rwx?)
  – present bit(P)
  – physical frame number

• Attempting to reference an unmapped page causes a page fault.

VM translation is illustrated in Figure 33

What it provides

1. expanded memory
2. relocation
3. isolation

The last one, isolation is important to consider: no process can refer to another process’s memory. So how can processes communicate?

  Only through the kernel because the kernel has access to physical memory. (At least part of it—the pager—must have access. (How does it do that? It’s using VM hardware, too.))

21.3 Translation Logistics(HERE)

Consider: page tables are large and must be accessed quickly:

  For 4KB Pages, 27 bits per entry:

<table>
<thead>
<tr>
<th>C</th>
<th>M</th>
<th>R</th>
<th>r</th>
<th>w</th>
<th>x</th>
<th>P</th>
<th>physical page number</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
</tr>
</tbody>
</table>

At one word (32 bits) per Page table entry:

<table>
<thead>
<tr>
<th>Virtual Memory size:</th>
<th>4Gb ($2^{32}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Size:</td>
<td>4k ($2^{12}$)</td>
</tr>
<tr>
<td>No. Pages:</td>
<td>1048576 ($2^{20}$)</td>
</tr>
<tr>
<td>P.T. Size</td>
<td>4MB</td>
</tr>
</tbody>
</table>

Remember, that’s per process because every process has its own virtual address space.
Figure 33: Virtual Memory translation in the MMU (4k pages)
21.3.1 Where to keep the page table?

- In the MMU? (fast translation, slow loading)
  
  With a 400 MHz memory bus, to move 4MB would take 0.0026s, and that’s just to move the data ignoring any instruction overhead. Of course, you’d have to move the old table out, too, so, that’d be 5.2 msec per context switch. Given that a quantum is 100msec, that’s a minimum of 5% overhead!

- in Memory (slower translation, possibly faster loading)

21.3.2 Possible approaches to dealing with size

- restrict virtual memory
- page the page tables (multi-level page tables?)
- Even so, many references per instruction:
  
  TLB: a small, associative page-table cache.

21.3.3 Translation Lookaside Buffers

- Small (32–64 entry) associative cache of page table entries
- Management (hardware vs. software.)

  **hardware management** A TLB miss causes an ordinary page lookup and the MMU replaces a TLB entry.

  **software management** A TLB miss causes a trap. This has to be handled efficiently, but with reasonably large TLBs the miss rate can be kept low.

  Why do this? This leads to a much simpler (i.e. faster) TLB.

  Idea: Keep a cache of recently evicted TLB entries around and check them first. (Consider the case where a multi-level page table must be searched.)

21.3.4 What about really big memories?

<table>
<thead>
<tr>
<th>Memory size:</th>
<th>$2^{24}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Size:</td>
<td>4k (2^{12})</td>
</tr>
<tr>
<td>No. Pages:</td>
<td>$2^{52}$ entries, 4,503,599,627,370,496</td>
</tr>
<tr>
<td>P.T. Size</td>
<td>8B * $2^{24}$ is a lot (36,028,797,018,963,968 ≈ 36PB)</td>
</tr>
</tbody>
</table>

21.3.5 Limited memory size

Always an option

(Modern linux only uses 47 bits of it.)

21.3.6 Inverted page tables

Keep track of the frames, not the pages: Slow, but bounded in size. (For 1 Gb, must search 256k page table entries, 1MB on every memory reference?)

  The TLB saves us.

  Example: Order card catalog in the library by call number (where it is stored) vs. title (what it is).
21.4 Page replacement algorithms


21.4.1 Optimal

Have a list of future references and evict the page that won’t be used for the longest time.

21.4.2 Random

Pick a page, any page.

21.4.3 Not Recently Used

Divide the pages into categories based on R(reference) and M(modify) bits: (Clear R on clock tick)

1. not referenced, not modified
2. not referenced, modified
3. referenced, not modified
4. referenced, modified

Evict a random page from the lowest numbered category.
Rationale?

21.4.4 FIFO

Pure round-robin first in, first out. Evict the oldest page in memory. (Problem: might be main())

21.4.5 Second Chance

Like FIFO, but if the head of the queue was referenced, clear the referenced bit and send it to the back of the list.
Reasonable, but has a lot of manipulation.

21.4.6 Clock (optimized Second Chance)

Second chance, but moves a pointer rather than manipulating the list.

21.4.7 LRU

Least Recently Used — best approximator of Optimal, but difficult:

- requires list manipulation

- can be approximated by a counter that’s updated in the page table at every reference. (requires hardware)
21.4.8 Not Frequently Used

Simulate LRU: At every clock tick, add the Reference bit into a counter in the page table. Replace the page with the lowest counter on a page fault.

Problem: NFU Never forgets.

Soln: Modify NFU with aging: Right-shift the counter and add the bit in at the high end.