Virtual Memory - Part II

CS 537 • Introduction to Operating Systems

Page Table Size

- Where does page table live?
  - virtual memory?
  - or physical memory?
- How big is page table?
  - 32 bit addressing, 4K page; 4 byte table entry
    - 4 MB
  - with 64 bit addressing, this number is huge

Page Table Size

- If page table stored in physical memory, pretty substantial overhead
- Solution
  - track frames instead of pages
  - OR, put the page table in virtual memory
- At some point, something must exist in physical memory or nothing can be found
  - need some structure in physical memory that keeps track of where the page table is
Inverted Page Table

- Instead of a page table, keep a frame table
  - one entry for each frame in the system
  - an entry contains the page number it is mapping
- Table size is now proportional to physical memory
  - page size = 4 KB
  - total memory size = 128 MB
  - table entry = 3 bytes
  - table size = $2^{28} / 2^{12} = 2^6 = 64$ KB
  - less than 1% of memory is needed for the table

Inverted Page Table

Inverted Page Table

- Major flaw with inverted page table
  - must search entire table to find page
  - can’t just index in like regular page table
- Still need to keep around a structure for all of the pages to indicate where they are at on disk
Multilevel Paging

- In physical memory, keep a mini page table
- The entries in this page table refer to the physical locations of the real page table
- Consider a system with a 4 MB page table and 4 KB pages
  - number of pages to hold page table is
    - $2^{23}/2^{12} = 2^{11} = 1K$
  - if each entry in mini table entry is 4 bytes
  - page table in physical memory is 4 KB

Multilevel Paging Addressing

- Address is now broken up into 3 parts
  - outer page index
  - inner page index
  - offset


Multilevel Paging Example

- This is a two-level page table
- Could also have 3 or 4 levels of paging
Effective Access Times

- Doing lots more references to memory
- Effective memory access
  - average time for some random access
- For the two level scheme above
  - assume \( t_{mem} = 100 \text{ ns} \) (time per memory access)
  - \( t_{eff} = 3 \times t_{mem} = 300 \text{ ns} \)
- We have just made our average access three times as long
  - even worse for more levels of indirection

Reducing \( t_{eff} \)

- Memory accesses occur very frequently
  - They must be fast
- Recall that we have 2 tricks
  - indirection and caching
- We used indirection to save space
- We will use caching to save performance

TLB

- Need hardware to make paging fast
- Translation Look-aside Buffer (TLB)
- Hardware device that caches page table entries
- TLB can be manipulated by the operating system
  - special instructions
TLB

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Page Location</th>
<th>X</th>
<th>W</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Table is searched in a **fully-associatively** manner
  - all page numbers are checked for match at same time
- If page match is found and page is valid
  - just combine the offset to the page location
- Otherwise, generate a page fault and have OS search for the page

### TLB

- If page is found in TLB
  - TLB hit
- If page is not found in TLB
  - TLB miss
- TLB hit rates are typically about 90%
  - locality of reference

### TLB Example

![TLB Example Diagram]
Effective Access Time

- Assumptions
  - $t_{mem\text{Hit}} = 100$ ns (memory access time on hit)
  - $t_{mem\text{Miss}} = 300$ ns (memory access time on miss)
  - $P_{hit} = 0.90$ (TLB hit percentage)
- Calculating effective access time
  - $t_{eff} = P_{hit} \cdot t_{mem\text{Hit}} + (1 - P_{hit}) \cdot t_{mem\text{Miss}}$
  - $t_{eff} = 0.90 \cdot 100 + 0.10 \cdot 300 = 120$ ns
- Average time is 20% longer than best case
  - if hit rates are high, TLB works great

Important Observations

- OS does not get involved at all if page is cached in the TLB
- If page not in cache, OS does get involved
- Access time increases drastically for a TLB miss
  - this is partially due to extra memory references
  - partially due to extra instructions the OS must run

TLB Fault Handler

- On a TLB miss:
  1. trap to operating system
  2. save registers and process state
  3. check if page in memory
     - if it is, go to step 5
     - if it is not go to step 4
  4. do a page fault
  5. make the appropriate entry in the TLB
  6. restore process registers and process state
  7. reexecute the line of code that generated the fault
- All of the software steps above take 10’s of microseconds
- The page fault could take 10’s of milliseconds
Page Fault Handler

- On a page fault
  1. find the offending page on disk
  2. select a frame to read the page into
  3. write the page currently in the frame to disk
     (this may or may not be necessary, more on this later)
  4. read the page on disk into the frame
  5. modify the page table to reflect change
- Notice the possibility for two disk ops
  - one write, one read
  - may be able to avoid one of these

Effective Access Time

- Assumptions
  - \( t_{\text{readhit}} = 100 \text{ ns} \)
  - \( t_{\text{readMiss}} = 25 \text{ ms} = 25,000,000 \text{ ns} \)
  - \( P_{\text{hit}} = 0.99 \)
- Effective access time
  - \( t_{\text{eff}} = 0.99 \times 100 + 0.10 \times 25,000,000 \)
  - \( t_{\text{eff}} = 2,500,099 \approx 2.5 \text{ ms} \)
- This access time would be completely unacceptable to performance

Effective Access Time

- Some simple math
  - \( t_{\text{eff}} = (1 - P_{\text{miss}}) \times t_{\text{readhit}} + P_{\text{miss}} \times t_{\text{readMiss}} \)
  - \( P_{\text{miss}} = (t_{\text{hit}} - t_{\text{readhit}}) / (t_{\text{readMiss}} - t_{\text{readhit}}) \)
- For an effective access of 120 ns
  - \( P_{\text{miss}} = (120 - 100) / (25,000,000 + 100) \)
  - \( P_{\text{miss}} = 0.0000008 \)
- That means 1 miss per 1,250,000 accesses!
- Obviously, it is crucial that the page hit rates be very high
Multiple Processes

- There is usually a separate page table for each process.
- When a process is swapped in, so is its page table.
  - It's part of the process's state.
- 2 options when dealing with the TLB.
  - Flush it.
    - Can be expensive.
  - Consider part of process state.
    - More data to save and restore.

Page Sharing

- Another nice feature of paging is the ability of processes to share pages.
- Map different pages in different processes to the same physical frame.
  - Shared data, shared code, etc.
- If read only pages can still be considered separate memory for each process.

Copy-on-Write

- Clever trick to help with performance and still implement separate memory / process.
  - Mark a shared page as read only.
  - If any process tries to write it, generates a fault.
  - OS can recognize page as being shared.
  - OS then copies the page to a new frame and updates page tables and TLB if necessary.
  - OS then returns control to writing process which is now allowed to write.
- Can greatly improve performance.
  - Consider the `fork()` system call.
Issues with Paging

- Notice that process is restarted from the instruction that caused the exception.
- Consider an architecture that allows the state of a machine to change during the instruction:
  - an increment or a decrement
  - what happens if we increment R2 and then try to write to R3 and take a page fault
  - now R2 is different and restoring the instruction will give incorrect results.
- Either don’t allow these types of instructions or provide a way to deal with it.