Virtual Memory

Today's topics:

Virtual memory
deeper look at memory hierarchy & management
TLB's for increased speed and protection
a few examples of approaches to date
Midterm Review
topics you should pay attention to

Main Memory Organization

- Familiar optimizations
  - wider memory
    - make a main memory transaction look like a cache line
    - handled primarily by the memory controller
  - bus width
    - actually a standard
    - wider & slower JEDEC
    - skinnier and faster RAMBUS
  - pipeline
    - with synchronous DRAM's
    - pipeline extended into DIMM and DRAM chips
  - interleaved or phased memory
    - n slow banks – interleave return on higher bandwidth path/bus
    - ultimately the trick being used in DDR1, 2, 3, ...
  - optimize for sequential memory accesses
    - capitalize on spatial locality similar to caches

Memory Hierarchy

- 3 physical memory types
  - caches – on chip SRAM
  - main memory – off chip DRAM
    - fronted by a memory controller
  - disk – either SSD or HD
    - magnetic or slow NVRAM
    - details later for new thing SUPER SLOW

Common principle
- similar to multi-level caches
- miss here?
  - dig deeper

Hierarchy Options
Striping/Interleaving

- Different for disks
  - Ignore this for now
- For caches and main memory
  - exploit concurrency in banks

<table>
<thead>
<tr>
<th>Word address</th>
<th>Bank 0</th>
<th>Bank 1</th>
<th>Bank 2</th>
<th>Bank 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

Virtual Memory

- Large virtual address space
  - mapping mechanism to physical main memory
    - e.g. 64 bit virtual address space
    - smaller physical address
  - 20-40 bit common now
- Multiple process management
  - each process has a "private" and "protected" virtual address space
    - trick is how to manage the private/protected illusion so it's true
      - for caches: virtual indexed and tagged via address spaces
    - between DRAM and disk
      - miss becomes a page or TLB fault
      - TLB is just a cache of recently used page table entries
      - block becomes a page or segment

Page Relocation

- Page table allows contiguous virtual addresses to be mapped in a non-contiguous fashion in main memory

Difficulties

- TLB is a cache
  - usually highly associative
    - conflict miss penalty is huge since miss
      - to main memory (~300 cycles) or disk (~10 msec)
- Main memory has 2 masters
  - cache line sized blocks move up in the hierarchy
  - page sized blocks move down in the hierarchy
  - memory controller has to keep it straight
    - used to be on the Northbridge chipset
    - now moving on chip
      - 2 of them on Nehalem for example
Line vs. Page Differences

- **Replacement**
  - page fault handled by OS
  - time to access disk + context switch is large
  - hence more exotic replacement (LRU’ish) policy is tractable

- **Capacity**
  - cache size choice is unrelated to either physical or virtual address size
  - physical address size specifies maximum main memory size
    - smaller is OK but mask exists to based on existing configuration
  - virtual address size specifies the minimum swap space size
    - multiply by how many processes you’d like to be partially resident

- **What’s on the disk**
  - SWAP partition
  - File system partitions

2 VM Styles: Main Memory

- **Pages are fixed size**
  - super-page options exist to increase TLB reach

- **Segments**
  - variable sized – hence base pointer and offset addressing

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Page</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block/Address</td>
<td>Line - contains page and offset</td>
<td>Non - provides large main size hence used 32-bit and offset address works</td>
</tr>
<tr>
<td>Programmed valid</td>
<td>No</td>
<td>Sufficient yes</td>
</tr>
<tr>
<td>Replacement</td>
<td>Serial - due to fixed size</td>
<td>Hard - need to find contiguous space - if SC necessary on virtual memory</td>
</tr>
<tr>
<td>Memory efficiency</td>
<td>Internal fragmentation - varied part of page</td>
<td>External Fragmentation - due to variable-size blocks</td>
</tr>
<tr>
<td>Disk Efficiency</td>
<td>Yes - adjust page size to balance access and translation time</td>
<td>Not always - segment size varies</td>
</tr>
</tbody>
</table>

VM’s & Same 4 Questions

- **Placement**
  - lower miss rates vs. complex placement
    - large miss penalty
    - choose low miss rate → place anywhere
      - similar to fully associative cache but on a page granularity in main mem

- **Addressing**
  - pages via a page table
    - VPN → PPN and concatenate page offset
  - page table or TLB cache does translation
  - valid bit needed as a minimum to indicate presence in main mem

- **Segmentation**
  - segment table
    - segment # → offset in segment table
      - pointer to head of segment table required
    - lots of segments → bigger segment table required

VPN → PPN Mapping Basics
Normal Page Tables

- **Size**
  - # entries = number of virtual pages

- **Role**
  - VPN → PPN translation
    - enables page relocation
  - still need status tags
    - valid
    - protection: priv'd, R, W, Xeq, ...

- **Potential problem**
  - 64-bit virtual address space, 34 bit physical address & 4 KB page
    - page table has 2 \(^{32}\) entries
    - YOW that's more than physical memory
    - ideas of how to fix this?

Inverted Page Table

- Make page table reflect what's in physical memory
  - use a hash mechanism
    - create index into inverted page table
    - compare VPN with tag to make sure of the hit
      - similar to caches
  - If you don't find it you go to disk
    - double jeopardy
      - disk access to get the page table
      - disk access to get the page you want
      - plus update the IPT

- The good bit
  - caches miss rarely
  - IPT miss is even more rare

Page Policies

- **Replacement**
  - LRU best but same story - expensive
  - hence "use" bit idea is employed
    - rarer OS wake up makes this closer to LRU than it is for caches
    - strategy
      - spend a few OS cycles to reduce miss rate and horrific page miss penalty

- **Write strategy**
  - always write back – so dirty bit required
    - write-through to disk is silly
  - write buffering works as with caches
    - larger grain size → larger buffer size
    - get the requested one first then do the write when you can

Page Size Dilemma

- **Large pages are good**
  - reduces page table size
  - increases TLB reach
  - amortizes long disk latencies
  - works well when spatial locality is in play

- **Large pages are bad**
  - more internal fragmentation
    - last page of text, heap, and control stack is 50% wasted
  - process start up delay
    - at least the first 3 pages of each type is required
    - big pages → longer transfer time delay
AXP 21064 TLB Example

<table>
<thead>
<tr>
<th>Page Frame Number</th>
<th>Physical PN</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPN Tag</td>
<td>Physical PN</td>
</tr>
</tbody>
</table>

Indicates steps that could be pipeline possibilities - Why?

Improvements

- What's changed
  - Multiple processes' data can co-reside in memory
    - Technology side-effect
      - Larger memories, more processes, higher context switch overhead
    - Virtual address alias problem
      - Use PID's in an option but there are too many of them
      - Use address space numbers instead
        - Similar mechanism discussed for caches
          - 1 per process
          - No need for TLB flush if valid ASN
          - If start a process without a current ASN
            - Remap and flush TLB with evicted ASN
            - No write back needed - just invalidate
        - HP-PA series used this idea from the start

Protection Options

- Base and bound – segmented VM
  - Check that address falls between 2 register values
    - Registers can only be changed by OS/Priv'd instructions
    - PID or ASN idea can be used for finer grain ACL control

- Paged VM
  - Check as part of the VA→PA translation
    - Held in TLB on a page based privilege

- Other options
  - Ring based
    - Multics (late 60's) and now Pentium
      - Inner ring is most priv'd – outer is user
      - Allows greater distinction kernel, OS, loadable module, user
    - Capabilities (ala the ill-fated Intel i432)
      - Key or password based model
      - OS hands them out so difficult to forge
      - Apps can pass them around which could be dangerous
### AXP 21264

- **Both segmented and paged**
  - 64 bit address space (only 48 were used)
    - addr[63:46] = all 0's
      - seg0 for text and heap - grows upward
      - seg1 for stack grows downward
    - addr[15:4] = all 1's
      - kseg - reserved for OS kernel
      - uniformly protected space - no memory management
      - idea is to keep kernel resident w/ no perturbation from seg0 & seg1

- **Advantages**
  - segmentation conserves page table space
  - paging provides VM, protection and relocation
    - paging happens within each segment
    - split page tables
  - best of both worlds
    - best of what?  

---

### 21264 VM Problems

- **Big page table**
  - big memories are slow
  - go hierarchical here too
    - 3 levels of page table
      - each table is 1 page in size
      - 8KB page but support for super pages 16, 32, 64KB in 21364
      - 34 bit physical address
    - Virtual addr = [seg index, lvl1, lvl2, lvl3, offset]
      - mapping
        - LVL1-TBL[lvl1]+lvl2 points to LVL2-TBL entry
        - and so forth
        - LVL3-TBL entry provides PPN
        - PPN+offset = physical address for main memory

---

### 21264 Mapping

- Virtual address
  - seg0 or 1
  - level 1
  - level 1
  - level 1
  - page offset
  - Physical address
    - physical page frame
    - page offset

---

### Concluding Remarks

- **Pentium**
  - both paged and segmented w/ table based translation
    - VA's mapped to segments and physical addresses
      - contains protection bits - 4 level ring based
      - writable - depends on who calls who
      - allows user code to safe call OS and see shared memory
      - possible Trojan horse problem
      - fix by not allowing OS to provide an indirect reference
  - OS and user stack are separate - hence copy required
  - no parameter passing via registers
    - example of not trusting the OS much (oh that would be MS)
    - lots of other cruft that we'll ignore

- **Things are getting hairier**
  - multiple cores, VM’s, hypervisors
    - open question is what is the right HW support for protection
      - mantra - common case goes to HW, flexibility is best done in SW
Midterm Review 1

• Topics
  • cost and market segment issues
  • quantitative analysis ala HW1 and HW2
  • ISA issues
    » RISC vs. CISC, memory modes
  • Pipelining
    » performance issues, hazards, forwarding, laminarity
  • ILP
    » various optimizations in both HW and SW/compiler
      • know the trade-offs
  • Branch prediction (HW3 should have focused some attention)
    » static vs. dynamic
    » local, global, tournament
    » predict what
      • taken-not taken
      • address
      • instruction

Midterm Review 2

• More topics
  • dynamic issue superscalar
    » Scoreboarding and Tomasulo
      • know how they work
  • static issue superscalar
    » VLIW and EPIC
      • understand group and bundle idea from EPIC
  • ILP limitations
    » there are good reasons for going multi-core and multi-threaded
    » know why and the basic multi-threading approaches
  • Caches
    » we’ve covered the basics
    » conceptual question only is possible
    » understand
      • basic organizations: direct-mapped, set-associative, associative
      • basic tradeoffs and organizations
      • miss types what what helps reduce each type
      • basic memory performance equations