18-447 Lecture 18:
Page Tables and TLBs

James C. Hoe
Department of ECE
Carnegie Mellon University

Housekeeping

- Your goal today
  - see the reality of page tables
  - delve into the many nuts and bolts of VM supports
- Notices
  - Lab 3, due this week
  - Handout #13: Lab 4
- Readings
  - P&H Ch 5
  - required for L19: “Virtual memory in contemporary microprocessors” by Jacob and Mudge (on Blackboard)
Lab 4 Preview

**Secondary Pipeline**

- IF/ID
- ID/EX
- EX/MEM
- MEM/WB

**Primary Pipeline**

- IMEM
- PC
- Decode
- Reg File
- ALU and address calculation
- DMEM
- RegFile WB
- Exception Unit
- Branch Resolution

**Lab 4 Preview**

- do what it takes to keep the duo-pipeline fed
- do what you like to increase pipeline throughput
- partially graded on performance ranking
  - inst/sec
  - Inst/sec/Watt
Just one more thing: How large is the page table?

- A page table holds mapping from VPN to PPN
- Suppose 64-bit VA and 40-bit PA, how large is the page table? $2^{52}$ entries x ~4 bytes ≈ 16x10^{15} Bytes
  And that is for just one process!!?

How large is the page table?

- Don’t need to keep track of the entire VA space
  - total allocated VA space in a system is $2^{64}$ bytes x # processes, but most of which is not alive
  - system can’t possibly use more memory locations than physically exist (DRAM and swap disk)
- A clever page table should scale “linearly” with size of physical storage and not size of VA space
- Also cannot be too convoluted
  - a page table must be “walkable” by HW
  - a page table is accessed not infrequently
- Two basic schemes in use today
  - hierarchical page tables
  - hashed page tables
Hierarchical Page Table

- Hierarchical page table is a “tree” data structure in DRAM

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L1 idx10</td>
<td>L2 idx10</td>
<td>PO12</td>
</tr>
<tr>
<td></td>
<td>context table</td>
<td>L1 table</td>
<td>L2 table</td>
</tr>
<tr>
<td></td>
<td>descriptor</td>
<td>descriptor</td>
<td>PTE</td>
</tr>
<tr>
<td></td>
<td>PA to base of L1</td>
<td>PA to base of L2</td>
<td>PA to base of page frame (i.e., PPN) or location on swap disk</td>
</tr>
</tbody>
</table>

Exact implementations vary greatly. Next lecture!!

Hierarchical Page Table

- Hierarchical page table is a “tree” graph,
  - for example on previous page
    - L1 table has 1024 decedents (L2 tables)
    - each L2 table has 1024 decedents (physical page frames)
  - more levels can be used to accommodate larger VA
  - assume 4-byte descriptors and PTEs, each table is 4KByte (size of page frames) such that the tables can be demand paged between DRAM and disk
- Hierarchical page table is a “sparse” tree graph
  - if none of the virtual page frames associated with a L2 table is in used, the L2 table does not need to exist (corresponding L1 entry points to null)
  - in general, an entire unused sub-tree can be avoided
  - considering typical size ratio of VA to PA, the tree should be quite sparse

How sparse?
How large is the hierarchical table?

- Assume 32-bit VA with 4 MByte in use
- **Best Case:** one contiguous 4-MByte region in VA aligned on 4MByte boundaries
  - 1K physical page frames
  - needs 1 L2 table + 1 L1 table=2 x 4KBytes,
  - overhead ≈ sizeof(PTE)/page_size per physical page
- **Worst Case:** 1K 4-KByte regions in VA; each is 4MByte aligned
  - 1K physical page frames
  - needs 1K L2 tables (only 1 entry per L2 table in use)
  - 1025 x 4KBytes
  - overhead ≈ 1 page per data page
- Locality says we should be closer to the best case

4 bytes/4Kbytes ≈ 0.1%

---

Hashed Page Table

- Choose an appropriate page table storage overhead
  - at least 1 entry per physical page, but probably more to avoid “hash” conflicts
  - e.g., 1GB DRAM ⇒ 256K frames ⇒ 256K PTEs
- Page table works like a hash table
  - to lookup a translation, hash VPN and PID into a index
    e.g., (VPN⊕PID)%table_size (note: overly simplified)
  - assumes PTE was inserted according to the same hash
  - each entry must be “tagged” by PID and VPN to detect collision

![Hashed Page Table Diagram]
How large is the hashed page table?

- Hashed table size is a function of physical mem. size
- Exact proportion is an engineering choice  
  large enough to reduce hash collisions
- Hashed table is not complete and only tracks DRAM-resident; on a miss, consult a secondary complete table structure
- The original “inverted” page table (a historical note)
  - allocate exactly 1 entry per physical page frame
  - VPN that hashes to entry i can only use physical frame i
  - entry only holds VPN (tag) but not PPN (implied by i)  
    viewing the table out-of-context, it is indexed by PPN and returns VPN

Translation Look-Aside Buffer (TLB)

- Every user memory reference (code or data) requires a translation  
  - how many memory accesses per translation?  
    hierarchical vs. hashed
  - what good is it to hit in the cache if translation takes forever
- TLB: a “cache” of most recently used translations  
  - same type of “tagged” lookup structure as caches and BTBs  
  - given a VPN, returns a PTE (PPN & protections)
  - TLB entry:
    tag: address tag (from VA), PID  
    PTE: PPN, protection bits  
    misc: valid, dirty, etc.
  - similar design considerations as caches  
    capacity, block size, associativity, replacement policy
Direct-Mapped TLB (bad example)

```
<table>
<thead>
<tr>
<th>PID</th>
<th>VPN</th>
<th>PO</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>idx</td>
<td></td>
</tr>
</tbody>
</table>
```

- Tag Bank
- PTE Bank
- valid
- hit?
- PTE

TLB Design

- **Separate I and D-TLB, multi-level TLBs make sense as in caches**
  - **C**: if the L1 I-cache is 64KB, what’s the I-TLB size?
    - should cover the same 64KB footprint
    - a minimum of 16 TLB entries $\times$ some safety factor $(2^{\sim}8)$
    - in the old days 32~64 entries; nowadays a few hundred
  - **B**: after accessing a page, how likely is it to access the next page? (coarse grain spatial locality)
    - typically one PTE per TLB entry
    - MIPS stores 2 consecutive pages’ translations per entry
  - **a**: what associativity to minimize collision?
    - in the old days, fully-associative is the norm
    - nowadays, 2~4-way-associative is more common

Why?
On a TLB Miss

- Most address translation resolved in ~1 cycle in the TLB
- On a TLB miss
  - must “walk” the page table to determine translation
  - walk usually done by HW (MIPS walks in SW)
  - can take 100’s of cycles to complete
  - if PTE is found and page is in memory, then replace TLB with new PTE and continue
  - if PTE is found but the page is on disk, then trigger “page fault” exception to initiate kernel handler for demand paging
  - if PTE is not found, trigger “segmentation fault” exception to initiate kernel handler

What to do now?

VA to PA Translation

1. VA to PA Translation
2. TLB lookup
3. Protection check
4. PA to cache
5. "page fault" demand paging
6. "protection violation"
7. 10 msec
8. ~1 pclk
9. ~100’s pclk
10. no yes
11. no yes
How should VM and Cache Interact?

Virtual Caches

- Even with TLB, translation takes time
- Naively, memory access time in the best case is
  \[ \text{TLB hit time} + \text{cache hit time} \]
- Why not access cache with virtual addresses and only translate on a cache miss to DRAM
  make sense if \( \text{TLB hit time} \gg \text{cache hit time} \)
- Virtual caches in SUN SPARC, circa 1990
  - CPU has gotten fast enough that off-chip a SRAM access takes multiple cycles
  - dies size has gotten large enough to integrate L1 caches
  - MMU and TLB still on a separate chip

these conditions no longer hold
Managing Virtual Caches:
Synonyms and Homonyms

- **Homonyms** (same sound different meaning)
  - same EA (in different processes) points to different PAs
  - flush virtual cache between context; or include PID in cache tag
- **Synonyms** (different sound same meaning)
  - different EAs (from the same or different processes) point to the same PA
  - in a virtually addressed cache
    - a PA could be cached twice under different EAs
    - updates to one cached copy would not be reflected in the other cached copy
    - solution: make sure synonyms can’t co-exist in the cache, e.g., OS can forces synonyms to have the same index bits in a direct mapped cache

Virtually-Indexed Physically-Tagged
(a misnomer)

- If $C \leq (\text{page\_size} \times \text{associativity})$, the cache index bits come only from page offset (same in VA and PA)
- If both cache and TLB are on chip
  - index both arrays concurrently using VA bits
  - check cache tag (physical) against TLB output at the end

Only an issue for L1 caches
Large Virtually-Indexed Caches

- If \( C > (\text{page size} \times \text{associativity}) \), the cache index bits include VPN \( \Rightarrow \) Synonyms can cause problems
- Solutions
  - increase associativity
  - increase page size
  - MIPS R10K

R10000’s Virtually Index Cache

- 32KB 2-Way Virtually-Indexed L1
  - needs 10 bits of index and 4 bits of block offset
  - page offset is only 12-bits \( \Rightarrow \) 2 bits of index are VPN[1:0]
- Direct-Mapped Physical L2
  - L2 is inclusive of L1
  - VPN[1:0] is appended to the “tag” of L2
- Given two synonyms VA and VB that differs in VPN[1:0] and both map to the same physical address PA
  - Suppose VA is accessed first so blocks are allocated in L1&L2
  - What happens when VB is referenced?
    1. VB indexes to a different block in L1 and misses
    2. VB translates to PA and goes to the same block as VA in L2
    3. Tag comparison fails (VA[1:0] \( \neq \) VB[1:0])
    4. L2 detects that a synonym is cached in L1 \( \Rightarrow \) VA’s entry in L1 is evicted before VB is allowed to be refilled in L1
Interactions of VM and DMA

- A contiguous block in VA
  - is not necessarily contiguous in PA
  - may not be in memory at all
- Software solutions
  - user must allocate special pages if it is intended for DMA transfer later, or
  - kernel copies from user buffer to pinned, contiguous buffer before DMA
- Smarter DMA engines
  - OS creates in memory a “linked list” of commands for moving non-contiguous blocks
  - DMA engine follows linked list to perform gather/scatter
- Virtually-addressed I/O bus
  - I/O devices refer to contiguous data blocks by VA
  - translate by I/O TLB into PA before accessing main memory
  - can TLB-miss or page-fault (who handles this?)

Read “Virtual memory in contemporary microprocessors” by Jacob and Mudge before coming to next Lecture!!!