CS 152 Computer Architecture and Engineering

Lecture 11 - Virtual Memory and Caches

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152
Today is a review of last two lectures

• Translation/Protection/Virtual Memory
• This is complex material - often takes several passes before the concepts sink in
• Try to take a different path through concepts today
VM features track historical uses:

• Bare machine, only physical addresses
  – One program owned entire machine

• Batch-style multiprogramming
  – Several programs sharing CPU while waiting for I/O
  – Base & bound: translation and protection between programs (not virtual memory)
  – Problem with external fragmentation (holes in memory), needed occasional memory defragmentation as new jobs arrived

• Time sharing
  – More interactive programs, waiting for user. Also, more jobs/second.
  – Motivated move to fixed-size page translation and protection, no external fragmentation (but now internal fragmentation, wasted bytes in page)
  – Motivated adoption of virtual memory to allow more jobs to share limited physical memory resources while holding working set in memory

• Virtual Machine Monitors
  – Run multiple operating systems on one machine
  – Idea from 1970s IBM mainframes, now common on laptops
    » e.g., run Windows on top of Mac OS X
  – Hardware support for two levels of translation/protection
    » Guest OS virtual -> Guest OS physical -> Host machine physical
Bare Machine

- In a bare machine, the only kind of address is a physical address
Logical address is what user software sees. Translated to physical address by adding base register.
[Can fold addition of base register into (base+offset) calculation using a carry-save adder (sum three numbers with only a few gate delays more than adding two numbers)]
Memory Fragmentation

As users come and go, the storage is “fragmented”. Therefore, at some stage programs have to be moved around to compact the storage.
Paged Memory Systems

- Processor generated address can be interpreted as a pair \(<\text{page number}, \text{offset}>\)

- A page table contains the physical address of the base of each page

Page tables make it possible to store the pages of a program non-contiguously.
Private Address Space per User

- Each user has a page table
- Page table contains an entry for each user page
Linear Page Table

- Page Table Entry (PTE) contains:
  - A bit to indicate if a page exists
  - PPN (physical page number) for a memory-resident page
  - DPN (disk page number) for a page on the disk
  - Status bits for protection and usage
- OS sets the Page Table Base Register whenever active user process changes
Page Tables in Physical Memory

User 1

VA1

User 2

VA1

PT User 1

PT User 2
Size of Linear Page Table

With 32-bit addresses, 4-KB pages & 4-byte PTEs:

⇒ $2^{20}$ PTEs, i.e, 4 MB page table per user
⇒ 4 GB of swap needed to back up full virtual address space

Larger pages?

• Internal fragmentation (Not all memory in a page is used)
• Larger page fault penalty (more time to read from disk)

What about 64-bit virtual address space???

• Even 1MB pages would require $2^{44}$ 8-byte PTEs (35 TB!)

*What is the “saving grace”?*

*sparsity of virtual address usage*
Hierarchical (Two-Level) Page Table

Virtual Address

31 22 21 12 11 0

p1 p2 offset

10-bit 10-bit
L1 index L2 index

Root of the Current Page Table

(Processor Register)

Level 1 Page Table

Level 2 Page Tables

offset

p1

p2

Data Pages

page in primary memory
page in secondary memory
PTE of a nonexistent page
Two-Level Page Tables in Physical Memory

Virtual Address Spaces

User 1

VA1

User 2

VA1

User 1

Level 1 PT

User 1

Level 1 PT

User 2

User2/VA1

User1/VA1

Level 2 PT

User 2

Physical Memory
Address Translation & Protection

- Every instruction and data access needs address translation and protection checks

A good VM design needs to be fast (~ one cycle) and space efficient
Translation Lookaside Buffers

Address translation is very expensive!
In a two-level page table, each reference becomes several memory accesses

Solution: *Cache translations in TLB*

- TLB hit $\Rightarrow$ *Single Cycle Translation*
- TLB miss $\Rightarrow$ *Page Table Walk to refill*

<table>
<thead>
<tr>
<th>V</th>
<th>R</th>
<th>W</th>
<th>D</th>
<th>tag</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Virtual address*

- (VPN = virtual page number)
- (PPN = physical page number)

*Physical address*
Handling a TLB Miss

Software (MIPS, Alpha)
TLB miss causes an exception and the operating system walks the page tables and reloads TLB. A privileged "untranslated" addressing mode used for walk

Hardware (SPARC v8, x86, PowerPC)
A memory management unit (MMU) walks the page tables and reloads the TLB

If a missing (data or PT) page is encountered during the TLB reloading, MMU gives up and signals an exception for the original instruction
Page-Based Virtual Memory Machine
(Hardware Page Table Walk)

- Assumes page tables held in untranslated physical memory
CS152 Administrivia

• Tuesday Mar 9, Quiz 2
  – Cache and virtual memory lectures, L6-L11, PS 2, Lab 2
Virtual Memory

• More than just translation and protection
• Use disk to extend apparent size of main memory
• Treat DRAM as cache of disk contents
• Only need to hold active working set of processes in DRAM, rest of memory image can be swapped to disk
• Inactive processes can be completely swapped to disk (except usually the root of the page table)
• Combination of hardware and software used to implement this feature
• (ATLAS was first implementation of this idea)
Page Fault Handler

• When the referenced page is not in DRAM:
  – The missing page is located (or created)
  – It is brought in from disk, and page table is updated
    
    *Another job may be run on the CPU while the first job waits for the requested page to be read from disk*

  – If no free pages are left, a page is swapped out
    
    *Pseudo-LRU replacement policy*

• Since it takes a long time to transfer a page (msecs), page faults are handled completely in software by the OS
  – Untranslated addressing mode is essential to allow kernel to access page tables
Caching vs. Demand Paging

**Caching**
- cache entry
- cache block (~32 bytes)
- cache miss rate (1% to 20%)
- cache hit (~1 cycle)
- cache miss (~100 cycles)
- a miss is handled in *hardware*

**Demand paging**
- page frame
- page (~4K bytes)
- page miss rate (<0.001%)
- page hit (~100 cycles)
- page miss (~5M cycles)
- a miss is handled mostly in *software*
Address Translation: putting it all together

Virtual Address

Restart instruction

TLB Lookup

Page Table Walk

Page Fault (OS loads page)

Update TLB

Protection Fault

Protection Check

hit

miss

the page is

∉ memory

∈ memory

denied

permitted

Physical Address (to cache)

 SEGFAULT

hardware

hardware or software

software

February 25, 2010

CS152, Spring 2010

23
Address Translation in CPU Pipeline

- Software handlers need *restartable* exception on TLB fault
- Handling a TLB miss needs a *hardware* or *software* mechanism to refill TLB
- Need mechanisms to cope with the additional latency of a TLB:
  - slow down the clock
  - pipeline the TLB and cache access
  - virtual address caches
  - parallel TLB/cache access
Virtual Address Caches

- one-step process in case of a hit (+)
- cache needs to be flushed on a context switch unless address space identifiers (ASIDs) included in tags (-)
- aliasing problems due to the sharing of pages (-)
- maintaining cache coherence (-) (see later in course)

Alternative: place the cache before the TLB

(StrongARM)
Aliasing in Virtual-Address Caches

Two virtual pages share one physical page

Virtual cache can have two copies of same physical data. Writes to one copy not visible to reads of other!

General Solution: *Disallow aliases to coexist in cache*

Software (i.e., OS) solution for direct-mapped cache

VAs of shared pages must agree in cache index bits; this ensures all VAs accessing same PA will conflict in direct-mapped cache (early SPARC's)
Concurrent Access to TLB & Cache

Index L is available without consulting the TLB

⇒ *cache and TLB accesses can begin simultaneously*

Tag comparison is made after both accesses are completed

**Cases:** \( L + b = k \) \( L + b < k \) \( L + b > k \)
Virtual-Index Physical-Tag Caches: Associative Organization

Is this scheme realistic?

After the PPN is known, $2^a$ physical tags are compared.
Concurrent Access to TLB & Large L1

The problem with L1 > Page size

Can $VA_1$ and $VA_2$ both map to $PA$?
A solution via **Second Level Cache**

Often, a common L2 cache backs up both Instruction and Data L1 caches

L2 is “inclusive” of both Instruction and Data caches
Anti-Aliasing Using L2: *MIPS R10000*

- Suppose VA1 and VA2 both map to PA and VA1 is already in L1, L2 (VA1 ≠ VA2)
- After VA2 is resolved to PA, a collision will be detected in L2.
- VA1 will be purged from L1 and L2, and VA2 will be loaded ⇒ *no aliasing*!
Virtually-Addressed L1: Anti-Aliasing using L2

Physically-addressed L2 can also be used to avoid aliases in virtually-addressed L1.
Atlas Revisited

- One PAR for each physical page
- PAR’s contain the VPN’s of the pages \textit{resident in primary memory}
- \textit{Advantage}: The size is proportional to the size of the primary memory
- \textit{What is the disadvantage?}
Hashed Page Table: Approximating Associative Addressing

- Hashed Page Table is typically 2 to 3 times larger than the number of PPN’s to reduce collision probability.
- It can also contain DPN’s for some non-resident pages (*not common*).
- If a translation cannot be resolved in this table then the software consults a data structure that has an entry for every existing page.
Acknowledgements

• These slides contain material developed and copyright by:
  – Arvind (MIT)
  – Krste Asanovic (MIT/UCB)
  – Joel Emer (Intel/MIT)
  – James Hoe (CMU)
  – John Kubiatowicz (UCB)
  – David Patterson (UCB)

• MIT material derived from course 6.823
• UCB material derived from course CS252