Lecture #23
Virtual Memory

2005-11-21

Lecturer PSOE, new dad Dan Garcia

IBM reveals Cell processor →
One Power Processing Elt, (update of PowerPC), eight “Synergistic Processing Element” (SPE) cores. External chip BW is 76.8GB/s, 10x faster than any other chip; DRAM BW 25.6GB/s

www.popsci.com/popsci/whatsnew/4df68ca927d05010vgnvcm1000004eecbcddrcrd.html

Garcia, Fall 2005 © UCB
Cache Review

• Caches are NOT mandatory:
  • Processor performs arithmetic, memory stores data
  • Caches simply make data transfers go faster

• Each Memory Hierarchy level subset of next higher level

• Caches speed up due to temporal locality: store data used recently

• Block size > 1 wd spatial locality speedup:
  Store words next to the ones used recently

• Cache design choices:
  • size of cache: speed v. capacity
  • direct-mapped v. associative
  • choice of N for N-way set assoc
  • block replacement policy
  • 2nd level cache? 3rd level cache?
  • Write through v. write back?

• Use performance model to pick between choices, depending on programs, technology, budget, ...
Another View of the Memory Hierarchy

Thus far

Next: Virtual Memory

Upper Level
Faster
Larger
Lower Level
Memory Hierarchy Requirements

• If Principle of Locality allows caches to offer (close to) speed of cache memory with size of DRAM memory, then recursively why not use at next level to give speed of DRAM memory, size of Disk memory?

• While we’re at it, what other things do we need from our memory system?
Memory Hierarchy Requirements

• Share memory between multiple processes but still provide protection – don’t let one program read/write memory from another

• Address space – give each program the illusion that it has its own private memory
  
  • Suppose code starts at address 0x40000000. But different processes have different code, both residing at the same address. So each program has a different view of memory.
Virtual Memory

• Called “Virtual Memory”

• Also allows OS to share memory, protect programs from each other

• Today, more important for protection vs. just another level of memory hierarchy

• Each process thinks it has all the memory to itself

• Historically, it predates caches
Virtual to Physical Addr. Translation

• Each program operates in its own virtual address space; only one program running
• Each is protected from the other
• OS can decide where each goes in memory
• Hardware (HW) provides virtual physical mapping
Analogy

• Book title like virtual address

• Library of Congress call number like physical address

• Card catalogue like page table, mapping from book title to call #

• On card for book, in local library vs. in another branch like valid bit indicating in main memory vs. on disk

• On card, available for 2-hour in library use (vs. 2-week checkout) like access rights
Simple Example: Base and Bound Reg

- $base$
- $bound$
- $base+$

User C

User B

User A

OS

∞

Enough space for User D, but discontinuous ("fragmentation problem")

- Want discontinuous mapping
- Process size $>>$ mem
- Addition not enough!

⇒ use Indirection!
Mapping Virtual Memory to Physical Memory

- Divide into equal sized chunks (about 4 KB - 8 KB)
- Any chunk of Virtual Memory assigned to any chunk of Physical Memory ("page")

64 MB Physical Memory
Paging Organization (assume 1 KB pages)

**Physical Address**  | **Page is unit of mapping**  | **Virtual Address**
--- | --- | ---
0  | page 0  | 0
1024  | page 1  | 1024
...  | ...  | ...
7168  | page 7  | 2048

**Physical Memory**  | **Page also unit of transfer from disk to physical memory**  | **Virtual Memory**
--- | --- | ---
31744  | page 31  | 31744
Virtual Memory Mapping Function

- Cannot have simple function to predict arbitrary mapping
- Use table lookup of mappings

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Offset</th>
</tr>
</thead>
</table>
- Use table lookup ("Page Table") for mappings: Page number is index
- Virtual Memory Mapping Function
  - Physical Offset = Virtual Offset
  - Physical Page Number = PageTable[Virtual Page Number]
    (P.P.N. also called "Page Frame")
Address Mapping: **Page Table**

**Virtual Address:**
- `page no. offset`

**Page Table**
- **Base Reg**
- **index into page table**

**Page Table**
- **V A.R. P. P. A.**
- **Val-id Access Rights Physical Page Address**
- **...**

**Physical Memory Address**

Page Table located in physical memory
Page Table

• A page table is an operating system structure which contains the mapping of virtual addresses to physical locations
  - There are several different ways, all up to the operating system, to keep this data around

• Each process running in the operating system has its own page table
  - “State” of process is PC, all registers, plus page table
  - OS changes page tables by changing contents of Page Table Base Register
Requirements revisited

• Remember the motivation for VM:
  • Sharing memory with protection
    • Different physical pages can be allocated to different processes (sharing)
    • A process can only touch pages in its own page table (protection)
  • Separate address spaces
    • Since programs work only with virtual addresses, different programs can have different data/code at the same address!

• What about the memory hierarchy?
### Page Table Entry (PTE) Format

- Contains either Physical Page Number or indication not in Main Memory
- OS maps to disk if Not Valid \((V = 0)\)

<table>
<thead>
<tr>
<th>Page Table</th>
<th>(V)</th>
<th>A.R.</th>
<th>P. P.N.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Val-id</td>
<td>Access Rights</td>
<td>Physical Page Number</td>
<td></td>
</tr>
<tr>
<td>(V)</td>
<td>A.R.</td>
<td>P. P.N.</td>
<td></td>
</tr>
<tr>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td></td>
</tr>
</tbody>
</table>

- If valid, also check if have permission to use page: **Access Rights** (A.R.) may be Read Only, Read/Write, Executable
Paging/Virtual Memory Multiple Processes

User A: Virtual Memory

∞
Stack

Static
Code

0

A Page Table

64 MB

Physical Memory

User B: Virtual Memory

∞
Stack

Static
Code

0

B Page Table

User A: Virtual Memory

∞
Stack

Static
Code

0

A Page Table

64 MB

Physical Memory

User B: Virtual Memory

∞
Stack

Static
Code

0

B Page Table

Garcia, Fall 2005 © UCB
## Comparing the 2 levels of hierarchy

<table>
<thead>
<tr>
<th>Cache Version</th>
<th>Virtual Memory vers.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block or Line</td>
<td>Page</td>
</tr>
<tr>
<td>Miss</td>
<td>Page Fault</td>
</tr>
<tr>
<td>Block Size: 32-64B</td>
<td>Page Size: 4K-8KB</td>
</tr>
<tr>
<td>Placement:</td>
<td>Fully Associative</td>
</tr>
<tr>
<td>Direct Mapped,</td>
<td></td>
</tr>
<tr>
<td>N-way Set Associative</td>
<td></td>
</tr>
<tr>
<td>Replacement:</td>
<td>Least Recently Used</td>
</tr>
<tr>
<td>LRU or Random</td>
<td>(LRU)</td>
</tr>
<tr>
<td>Write Thru or Back</td>
<td>Write Back</td>
</tr>
</tbody>
</table>
Notes on Page Table

• Solves Fragmentation problem: all chunks same size, so all holes can be used

• OS must reserve “Swap Space” on disk for each process

• To grow a process, ask Operating System
  • If unused pages, OS uses them first
  • If not, OS swaps some old pages to disk
  • (Least Recently Used to pick pages to swap)

• Each process has own Page Table

• Will add details, but Page Table is essence of Virtual Memory
Why would a process need to “grow”? 

• A program’s **address space** contains 4 regions:
  - **stack**: local variables, grows downward
  - **heap**: space requested for pointers via `malloc()`; resizes dynamically, grows upward
  - **static data**: variables declared outside main, does not grow or shrink
  - **code**: loaded when program starts, does not change

For now, OS somehow prevents accesses between stack and heap (gray hash lines).
Administrivia

• Dan’s wed OH moved to Tu @ 1pm
• Project 4 is out, due next Fri: GUI Cache sim
• Labs this week are take-home
  • You can get checked off in a later lab
• Wed lecture cancelled (don’t show up)
  • Instead we’ll put up a recorded webcast by Prof. Patterson so you can spend time with your family. You’re responsible for the content!
  • Webcast and Notes synchronized!
    wla.berkeley.edu/videosmildemo2/patterson.ram
  • Just the Webcast
    rtsp://webcast.berkeley.edu:554/bibs/older1/f2001/cs3/20020506.rm
  • Just the Notes:
    inst.eecs.berkeley.edu/~wla/dave_patterson.pdf
• Don Corleone (from “The Godfather”):
  “Do you spend time with your family? Good. Because a man that doesn't spend time with his family can never be a real man.”
Virtual Memory Problem #1

- Map every address ⇒ 1 indirection via Page Table in memory per virtual address ⇒ 1 virtual memory accesses = 2 physical memory accesses ⇒ SLOW!

- Observation: since locality in pages of data, there must be locality in virtual address translations of those pages

- Since small is fast, why not use a small cache of virtual to physical address translations to make translation fast?

- For historical reasons, cache is called a Translation Lookaside Buffer, or TLB
Translation Look-Aside Buffers (TLBs)

- TLBs usually small, typically 128 - 256 entries
- Like any other cache, the TLB can be direct mapped, set associative, or fully associative

On TLB miss, get page table entry from main memory
### Typical TLB Format

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>Dirty</th>
<th>Ref</th>
<th>Valid</th>
<th>Access Rights</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- TLB just a cache on the page table mappings
- TLB access time comparable to cache (much less than main memory access time)
- **Dirty**: since use write back, need to know whether or not to write page to disk when replaced
- **Ref**: Used to help calculate LRU on replacement
  - Cleared by OS periodically, then checked to see if page was referenced
What if not in TLB?

• **Option 1**: Hardware checks page table and loads new Page Table Entry into TLB

• **Option 2**: Hardware traps to OS, up to OS to decide what to do
  - MIPS follows Option 2: Hardware knows nothing about page table
What if the data is on disk?

• We load the page off the disk into a free block of memory, using a DMA (Direct Memory Access — very fast!) transfer
  • Meantime we switch to some other process waiting to be run

• When the DMA is complete, we get an interrupt and update the process’s page table
  • …so when we switch back to the task, the desired data will be in memory
What if we don’t have enough memory?

• We chose some other page belonging to a program and transfer it onto the disk if it is dirty
  • If clean (disk copy is up-to-date), just overwrite that data in memory
  • We chose the page to evict based on replacement policy (e.g., LRU)

• And update that program’s page table to reflect the fact that its memory moved somewhere else

• If continuously swap between disk and memory, called Thrashing
Peer Instruction

A. Locality is important yet different for cache and virtual memory (VM): temporal locality for caches but spatial locality for VM

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

B. Cache management is done by hardware (HW), page table management by the operating system (OS), but TLB management is either by HW or OS

C. VM helps both with security and cost
Peer Instruction (1/3)

• 40-bit virtual address, 16 KB page

<table>
<thead>
<tr>
<th>Virtual Page Number (? bits)</th>
<th>Page Offset (? bits)</th>
</tr>
</thead>
</table>

• 36-bit physical address

<table>
<thead>
<tr>
<th>Physical Page Number (? bits)</th>
<th>Page Offset (? bits)</th>
</tr>
</thead>
</table>

• Number of bits in Virtual Page Number/ Page offset, Physical Page Number/Page offset?

1: 22/18 (VPN/PO), 22/14 (PPN/PO)
2: 24/16, 20/16
3: 26/14, 22/14
4: 26/14, 26/10
5: 28/12, 24/12
Peer Instruction (1/3) Answer

• 40-bit virtual address, 16 KB ($2^{14}$ B)

  Virtual Page Number (26 bits)   Page Offset (14 bits)

• 36-bit physical address, 16 KB ($2^{14}$ B)

  Physical Page Number (22 bits)   Page Offset (14 bits)

• Number of bits in Virtual Page Number/Page offset, Physical Page Number/Page offset?

  1: 22/18 (VPN/PO), 22/14 (PPN/PO)
  2: 24/16, 20/16
  3: 26/14, 22/14
  4: 26/14, 26/10
  5: 28/12, 24/12
Peer Instruction (2/3): 40b VA, 36b PA

- 2-way set-assoc. TLB, 256 “slots”, 40b VA:
  - TLB Tag (? bits)
  - TLB Index (? bits)
  - Page Offset (14 bits)

- TLB Entry: Valid bit, Dirty bit, Access Control (say 2 bits), Virtual Page Number, Physical Page Number
  - V  D  Access (2 bits)  TLB Tag (? bits)  Physical Page No. (? bits)

- Number of bits in TLB Tag / Index / Entry?
  1: 12 / 14 / 38 (TLB Tag / Index / Entry)
  2: 14 / 12 / 40
  3: 18 / 8 / 44
  4: 18 / 8 / 58
Peer Instruction (2/3) Answer

- **2-way set-assoc data cache, 256 \(2^8\) “slots”, 2 TLB entries per slot => 8 bit index**

<table>
<thead>
<tr>
<th>TLB Tag (18 bits)</th>
<th>TLB Index (8 bits)</th>
<th>Page Offset (14 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Page Number (26 bits)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **TLB Entry: Valid bit, Dirty bit, Access Control (2 bits), Virtual Page Number, Physical Page Number**

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Access (2 bits)</th>
<th>TLB Tag (18 bits)</th>
<th>Physical Page No. (22 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>12 / 14 / 38 (TLB Tag / Index / Entry)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>14 / 12 / 40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>18 / 8 / 44</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>18 / 8 / 58</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CS61C L23 Virtual Memory (35) Garcia, Fall 2005 © UCB
Peer Instruction (3/3)

- 2-way set-assoc, 64KB data cache, 64B block

```
| Cache Tag (? bits) | Cache Index (? bits) | Block Offset (? bits) |
```

Physical Page Address (36 bits)

- Data Cache Entry: Valid bit, Dirty bit, Cache tag + ? bits of Data

```
| V | D | Cache Tag (? bits) | Cache Data (? bits) |
```

- Number of bits in Data cache Tag / Index / Offset / Entry?

1: 12 / 9 / 14 / 87 (Tag/Index/Offset/Entry)
2: 20 / 10 / 6 / 86
3: 20 / 10 / 6 / 534
4: 21 / 9 / 6 / 87
5: 21 / 9 / 6 / 535
Peer Instruction (3/3) Answer

• 2-way set-associative data cache, 64K/1K ($2^{10}$) "slots", 2 entries per slot => 9 bit index

Physical Page Address (36 bits)

Cache Tag (21 bits)  Cache Index (9 bits)  Block Offset (6 bits)

Data Cache Entry: Valid bit, Dirty bit, Cache tag + 64 Bytes of Data

1: 12 / 9 / 14 / 87 (Tag/Index/Offset/Entry)
2: 20 / 10 / 6 / 86
3: 20 / 10 / 6 / 534
4: 21 / 9 / 6 / 87
5: 21 / 9 / 6 / 535

Cache Data (64 Bytes = 512 bits)
And in conclusion…

- Manage memory to disk? Treat as cache
  - Included protection as bonus, now critical
  - Use Page Table of mappings for each user vs. tag/data in cache
  - TLB is cache of Virtual→Physical addr trans

- Virtual Memory allows protected sharing of memory between processes

- Spatial Locality means Working Set of Pages is all that must be in memory for process to run fairly well
BONUS | 4 Qs for any Memory Hierarchy

• Q1: Where can a block be placed?
  • One place (direct mapped)
  • A few places (set associative)
  • Any place (fully associative)

• Q2: How is a block found?
  • Indexing (as in a direct-mapped cache)
  • Limited search (as in a set-associative cache)
  • Full search (as in a fully associative cache)
  • Separate lookup table (as in a page table)

• Q3: Which block is replaced on a miss?
  • Least recently used (LRU)
  • Random

• Q4: How are writes handled?
  • Write through (Level never inconsistent w/lower)
  • Write back (Could be “dirty”, must have dirty bit)
BONUS I Q1: Where block placed in upper level?

- Block 12 placed in 8 block cache:
  - Fully associative
  - Direct mapped
  - 2-way set associative
    - Set Associative Mapping = Block # Mod # of Sets

**Diagram:**
- Fully associative: block 12 can go anywhere
- Direct mapped: block 12 can go only into block 4 (12 mod 8)
- Set associative: block 12 can go anywhere in set 0 (12 mod 4)
BONUS I Q2: How is a block found in upper level?

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Block offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Index</td>
</tr>
</tbody>
</table>

- **Direct indexing (using index and block offset), tag compares, or combination**
- **Increasing associativity shrinks index, expands tag**

Set Select

Data Select
**BONUS | Q3: Which block replaced on a miss?**

- Easy for Direct Mapped
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

<table>
<thead>
<tr>
<th>Miss Rates Associativity: 2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 KB</td>
<td>LRU</td>
<td>Ran</td>
</tr>
<tr>
<td></td>
<td>5.2%</td>
<td>5.7%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
</tr>
</tbody>
</table>
BONUS I Q4: What to do on a write hit?

• **Write-through**
  - update the word in cache block and corresponding word in memory

• **Write-back**
  - update word in cache block
  - allow memory word to be “stale”

=> add ‘dirty’ bit to each line indicating that memory be updated when block is replaced

=> OS flushes cache before I/O !!!

• Performance trade-offs?
  - WT: read misses cannot result in writes
  - WB: no writes of repeated writes
BONUS | Three Advantages of Virtual Memory

1) Translation:
   • Program can be given consistent view of memory, even though physical memory is scrambled
   • Makes multiple processes reasonable
   • Only the most important part of program ("Working Set") must be in physical memory
   • Contiguous structures (like stacks) use only as much physical memory as necessary yet still grow later
BONUS: Three Advantages of Virtual Memory

2) Protection:
   • Different processes protected from each other
   • Different pages can be given special behavior
     - (Read Only, Invisible to user programs, etc).
   • Kernel data protected from User programs
   • Very important for protection from malicious programs ⟹ Far more “viruses” under Microsoft Windows
   • Special Mode in processor (“Kernel mode”) allows processor to change page table/TLB

3) Sharing:
   • Can map same physical page to multiple users (“Shared memory”)
BONUS I Why Translation Lookaside Buffer (TLB)?

• Paging is most popular implementation of virtual memory (vs. base/bounds)

• Every paged virtual memory access must be checked against Entry of Page Table in memory to provide protection

• Cache of Page Table Entries (TLB) makes address translation possible without memory access in common case to make fast
• User program view of memory:
  - Contiguous
  - Start from some set address
  - Infinitely large
  - Is the only running program

• Reality:
  - Non-contiguous
  - Start wherever available memory is
  - Finite size
  - Many programs running at a time
BONUS I Virtual Memory Overview (2/4)

• Virtual memory provides:
  • illusion of contiguous memory
  • all programs starting at same set address
  • illusion of ~ infinite memory
    \((2^{32} \text{ or } 2^{64} \text{ bytes})\)
  • protection
• Implementation:
  • Divide memory into “chunks” (pages)
  • Operating system controls page table that maps virtual addresses into physical addresses
  • Think of memory as a cache for disk
  • TLB is a cache for the page table
• Let’s say we’re fetching some data:
  • Check TLB (input: VPN, output: PPN)
    - hit: fetch translation
    - miss: check page table (in memory)
      – Page table hit: fetch translation
      – Page table miss: page fault, fetch page from disk to memory, return translation to TLB
  • Check cache (input: PPN, output: data)
    - hit: return value
    - miss: fetch value from memory