Memory 2: Paging, Caching, and TLBs

Sam Kumar
CS 162: Operating Systems and System Programming
Lecture 16
https://inst.eecs.berkeley.edu/~cs162/su20

Read: A&D Ch 9.1-4
Recall: Address Space

- Definition: **Set of accessible addresses and the state associated with them**
  - $2^{32} = \sim 4$ billion on a 32-bit machine

- What happens when you read or write to an address?
  - Perhaps acts like regular memory
  - Perhaps causes I/O operation
    - (Memory-mapped I/O)
  - Causes program to abort (segfault)?
  - Communicate with another program
  - ...

![Diagram of address space with sections labeled Code, Static Data, Heap, and Stack.]
Recall: Important Aspects of Memory Multiplexing

• Protection
  • Prevent access to private memory of other process or kernel

• Translation
  • Gives uniform view of memory to programs
  • Allows for efficient “tricks”
    • E.g., in implementation of fork()

• Controlled Overlap
  • Read-only data, execute-only shared libraries
  • Inter-process communication
Recall: Interposing on Process Behavior

• OS interposes on process’ I/O operations
  • How? All I/O happens via syscalls.

• OS interposes on process’ CPU usage
  • How? Interrupt lets OS preempt current thread

• Question: How can the OS interpose on process’ memory accesses?
  • Too slow for the OS to interpose every memory access
  • Translation: hardware support to accelerate the common case
  • Page fault: uncommon cases trap to the OS to handle
Recall: Uniprogramming

• No Translation or Protection
  • Application always runs at same place in physical memory since only one application at a time
  • Application can access any physical address
  • Application given illusion of dedicated machine by giving it reality of a dedicated machine
Recall: Primitive Multiprogramming

- Multiprogramming without Translation or Protection
- Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
  - Everything adjusted to memory location where OS put program
  - Translation done by a linker-loader (relocation)
- **No protection!**
Recall: Multiprogramming with Protection

• Can we protect programs from each other without translation?
  • Yes: Base and Bound!
  • Used by, e.g., Cray-1 supercomputer
Recall: General Translation

- Two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)
  - Hardware translator (Memory Management Unit or MMU) converts between the two views
- With translation, every program can be linked/loaded into same region of user address space
Recall: Issues with Simple Base and Bound

- Fragmentation problem over time
- No support for sparse address space
- Hard to do interprocess sharing
  - E.g., to share code
Recall: Segmentation

- Program’s view of memory: multiple separate segments
- Each segment is given a region of contiguous memory
  - Has a base and limit
- Memory address consists of segment ID and offset
Recall: Problems with Segmentation

• Must fit variable-sized chunks into physical memory

• May move processes multiple times to fit everything

• Limited options for swapping to disk

• Fragmentation: wasted space
  • External: free gaps between allocated chunks
  • Internal: don’t need all memory within allocated chunks
Recall: General Address Translation

In the diagram, we have two programs, Prog 1 and Prog 2, each with their own virtual address space. The virtual address space for each program is divided into Code, Data, Heap, and Stack sections.

Translation Map 1 and Translation Map 2 are used to translate virtual addresses to physical addresses. The Translation Map 1 shows the following mappings:

- Code
- Data
- Heap
- Stack

The Translation Map 2 shows:

- Code
- Static Data
- Heap
- Stack

The Physical Address Space contains OS code, OS data, OS heap/stacks, and the user process view of memory.

The diagram also shows the relationship between the virtual and physical address spaces, with the physical addresses ranging from 0x000... to 0xFFF...
Paging: *Fixed-Size* Chunks of Memory

• Divide up physical memory into equal-size chunks called *page frames*
• Divide up virtual memory into equal-size chunks called *pages*
• Key idea: each physical page frame can contain any page

• No external fragmentation!
• Should pages be as big as our previous segments?
  • No: Can lead to lots of internal fragmentation
    • Typically have small pages (1K-16K)
  • Consequently: need multiple pages/segment
Hardware Support for Paging

- **Page Table (One per process)**
  - Contains physical page and permission for each virtual page
    - Permissions include: Valid bits, Read, Write, etc
- **Virtual address mapping**
  - Offset from Virtual address copied to Physical Address
  - Virtual page # is the *index* into the page table
    - Physical page # copied from table into physical address
Simple Page Table Example

Example (4 byte pages)

Virtual Memory

- 0x00 to 0x03:
  - a to d
  - 0000 0000
- 0x04 to 0x07:
  - e to h
  - 0000 0100
- 0x08 to 0x0B:
  - i to l
  - 0000 1000

Page Table

- Page 0:
  - 0000 0000
  - 0x00 to 0x03:
    - 0x00
  - 0x04 to 0x07:
    - 0x01
  - 0x08 to 0x0B:
    - 0x02

Physical Memory

- 0x00 to 0x03:
  - a to d
  - 0000 0000
- 0x04 to 0x07:
  - e to h
  - 0000 0100
- 0x08 to 0x0B:
  - i to l
  - 0000 1000

What is the physical address for...

- Virtual address 0x6?
- Virtual address 0x9?
### Where is page sharing used?

- Kernel data mapped into each process
- Different processes running the same binary
- User-level system libraries
- Shared pages as IPC

---

### Virtual Address (Process A):

<table>
<thead>
<tr>
<th>Page Table Pointer</th>
<th>Virtual Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageTablePtrA</td>
<td>page #0</td>
<td>V,R</td>
</tr>
<tr>
<td></td>
<td>page #1</td>
<td>V,R</td>
</tr>
<tr>
<td></td>
<td>page #2</td>
<td>V,R,W</td>
</tr>
<tr>
<td></td>
<td>page #3</td>
<td>V,R,W</td>
</tr>
<tr>
<td></td>
<td>page #4</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>page #5</td>
<td>V,R,W</td>
</tr>
</tbody>
</table>

---

### Virtual Address (Process B):

<table>
<thead>
<tr>
<th>Page Table Pointer</th>
<th>Virtual Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageTablePtrB</td>
<td>page #0</td>
<td>V,R</td>
</tr>
<tr>
<td></td>
<td>page #1</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>page #2</td>
<td>V,R,W</td>
</tr>
<tr>
<td></td>
<td>page #3</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>page #4</td>
<td>V,R</td>
</tr>
<tr>
<td></td>
<td>page #5</td>
<td>V,R,W</td>
</tr>
</tbody>
</table>

---

**Shared Page**

This physical page appears in both process' address spaces.
Example: Memory Layout for Linux 32-bit*

* Pre-Meltdown patches, more later...

Diagram source: http://static.duartes.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png
Summary: Paging

Virtual memory view:
- Stack: 1111 1111 1111 0000
- Heap: 1100 0000
- Data: 0100 0000
- Code: 0000 0000

Page offset:
- Page #: 0000 0000

Page Table:
- Stack: 1111 1110 1110 1110
- Heap: 1100 0000 0111 0000
- Data: 0101 0000 0111 0000
- Code: 0001 0000 0011 0000

Physical memory view:
- Stack: 1110 0000
- Heap: 0111 0000
- Data: 0101 0000
- Code: 0001 0000

0000 0000
Summary: Paging

What happens if stack grows to 1110 0000?
Summary: Paging

What happens if stack grows to 1110 0000?

Allocate new pages where room!
Summary: Paging

What happens if stack grows to 1110 0000?

Allocate new pages where room!

Challenge: Table size equal to # of pages in virtual memory!
How Big is the Page Table?

• Typical page size: 4 KiB
  • How many bits of the address is that? (remember $2^{10} = 1024$)
  • Ans: $4\text{KiB} = 4 \times 2^{10} = 2^{12} \Rightarrow 12$ bits of the address

• So how big is the simple page table for each process?
  • $2^{32}/2^{12} = 2^{20}$ (that’s about a million entries) x 4 bytes each => 4 MiB
  • When 32-bit machines got started (vax 11/780, intel 80386), this was a lot of memory

• How big is a simple page table on a 64-bit processor (x86_64)?
  • $2^{64}/2^{12} = 2^{52}$ (that’s $4.5 \times 10^{15}$ or 4.5 exa-entries) x 8 bytes each = $36 \times 10^{15}$ bytes or 36 exa-bytes!!!! This is a ridiculous amount of memory!

• Mostly, the address space is sparse, i.e. has holes in it that are not mapped to physical memory
  • So, most of this space is taken up by page tables mapped to nothing
Page Table Discussion

• What provides protection here?
  • Translation (per process) and dual-mode operation!
  • Can’t let process alter its own page table!

• Analysis
  • Pros
    • Simple memory allocation
    • Easy to share
  • Con: What if address space is sparse?
    • E.g., on UNIX, code starts at 0, stack starts at \(2^{31}-1\)
    • With 1K pages, need 2 million page table entries!
  • Con: What if table really big?
    • Not all pages used all the time ⇒ would be nice to have working set of page table in memory

• Simple Page table is way too big!
  • Does it all need to be in memory?
  • How about multi-level paging?
  • or combining paging and segmentation
How to Structure a Page Table

• Page Table is a *map* from VPN to PPN

• Simple page table corresponds to a sparse array
  • VPN is index into table, each entry contains PPN

• What other map structures can you think of?
  • Trees?
  • Hash Tables?
Two-Level Page Table

- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Valid bits on Page Table Entries
  - Don’t need every 2\textsuperscript{nd}-level table
  - Even when exist, 2\textsuperscript{nd}-level tables can reside on disk if not in use
Summary: Two-Level Paging
Summary: Two-Level Paging
x86 Classic 32-bit Address Translation

Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging
### x86-64: Four-Level Page Table!

<table>
<thead>
<tr>
<th>Physical Address:</th>
<th>Virtual Address:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(40-50 bits)</td>
<td>(40-50 bits)</td>
</tr>
</tbody>
</table>

**Page Table Structure**

- **48-bit Virtual Address**
  - Virtual P1 index (9 bits)
  - Virtual P2 index (9 bits)
  - Virtual P3 index (9 bits)
  - Virtual P4 index (9 bits)
  - Offset (12 bits)

- **Page Table Ptr**
- **Physical Page #**
- **12-bit Offset**
- **Physical Address**: (40-50 bits)

**Key Points**

- 4096-byte pages (12 bit offset)
- Page tables are 4k bytes (pageable)
Large 64-bit Address Space

- All current x86-64 processors support 64-bit operations
- 64-bit words but 48-bit addresses

Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging
“Huge Pages” Supported as Well

- Memory is now cheap...

Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-

Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging
Intel Ice Lake (2019): One More Layer

Figure 2-1. Linear-Address Translation Using 5-Level Paging
Multi-Level Translation Analysis

• Pros:
  • Only need to allocate as many page table entries as we need for application
    • In other words, sparse address spaces are easy
  • Easy memory allocation
  • Easy Sharing
    • Share at segment or page level (need additional reference counting)

• Cons:
  • One pointer per page (typically 4K – 16K pages today)
  • Page tables need to be contiguous
    • However, previous example keeps tables to exactly one page in size
  • Two (or more, if >2 levels) lookups per reference
    • Seems very expensive!
Aside: Segments + Pages

• What must be saved/restored on context switch?
  • Contents of top-level segment registers (for this example)
  • Pointer to top-level table (page table)
Aside: x86 Memory with Segmentation

Segment Selector from instruction: `mov eax, gs(0x0)`

2-level page table in 10-10-12 bit address

Combined address is 32-bit “linear” Virtual address

First level called “directory”

Second level called “table”
### IA-64: 64-bit Address: Six Levels???

<table>
<thead>
<tr>
<th>64bit Virtual Address:</th>
<th>7 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>12 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual P1 index</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Offset</td>
</tr>
<tr>
<td>Virtual P2 index</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual P3 index</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual P4 index</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual P5 index</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual P6 index</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Too slow
Too many almost-empty tables
Alternative: Inverted Page Table

• With all previous examples (“Forward Page Tables”)
  • Size of page table is at least proportional to amount of virtual memory allocated to processes
  • Physical memory may be much less
    • Much of process space may be out on disk or not in use

• Answer: use a hash table
  • Called an “Inverted Page Table”
  • Size is independent of virtual address space
  • Directly related to amount of physical memory
  • Very attractive option for 64-bit address spaces
    • PowerPC, UltraSPARC, IA64

• Cons:
  • Complexity of managing hash chains: Often in hardware!
  • Poor cache locality of page table
## Address Translation Comparison

<table>
<thead>
<tr>
<th>Method</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Segmentation</td>
<td>Fast context switching (segment map maintained by CPU)</td>
<td>External fragmentation</td>
</tr>
</tbody>
</table>
| Paging (Single-Level) | No external fragmentation  
Fast and easy allocation               | Large table size (~ virtual memory)  
Internal fragmentation                                  |
| Paged Segmentation | Table size ~ # of pages in virtual memory                                | Multiple memory references per page access               |
| Multi-Level Paging  | Fast and easy allocation                                                  |                                                        |
| Inverted Page Table | Table size ~ # of pages in physical memory                                | Hash function more complex  
No cache locality of page table                         |
Announcements

• Congrats on finishing Quiz 2!

• Project 2 design doc due tonight

• Homework 4 (Page Walk) comes out tonight (or maybe tomorrow)
How to Translate Addresses Fast Enough?

The MMU must translate virtual address to physical address on:

- Every instruction fetch
- Every load
- Every store

More than one translation for EVERY instruction

- Each one requires a page table tree traversal (!)
- How to simplify this???
Where and What is the MMU?

- On every memory reference (I-fetch, Load, Store), MMU reads (multiple levels of) page table entries to get physical frame or FAULT
  - Through the caches to the memory
  - Then read/write the physical location
Recall: Memory Hierarchy

- Large memories are slow
- Small memories are fast

Address Translation needs to occur here
Page table lives here (perhaps cached)

<table>
<thead>
<tr>
<th>Speed (ns)</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3</td>
<td>100Bs</td>
</tr>
<tr>
<td>1</td>
<td>10kB</td>
</tr>
<tr>
<td>3</td>
<td>100kB</td>
</tr>
<tr>
<td>10-30</td>
<td>MBs</td>
</tr>
<tr>
<td>100</td>
<td>GBs</td>
</tr>
<tr>
<td>100,000</td>
<td>100GBs</td>
</tr>
<tr>
<td>10,000,000</td>
<td>10TBs</td>
</tr>
</tbody>
</table>
Recall: Caches

• **Cache**: a repository for copies that can be accessed more quickly than the original
  • Make frequent case fast and infrequent case less dominant

• Caching underlies many techniques used to make computers fast
  • Can cache: memory locations, address translations, pages, file blocks, file names, network routes, etc...

• Key measure: Average Access time =
  \[(\text{Hit Rate} \times \text{Hit Time}) + (\text{Miss Rate} \times \text{Miss Time})\]
Recall: Caching Memory

- Average Memory Access Time (AMAT)
  - \(\text{AMAT} = (\text{Hit Rate} \times \text{HitTime}) + (\text{Miss Rate} \times \text{MissTime})\)
  - Where HitRate + MissRate = 1
  - HitRate = 90% => AMAT = (0.9 x 1) + (0.1 x 101)=11.1 ns
  - HitRate = 99% => AMAT = (0.99 x 1) + (0.01 x 101)=2.01 ns
  - MissTime_{L1} includes HitTime_{L1}+MissPenalty_{L1} ≡ HitTime_{L1} + AMAT_{L2}
Recall: Why Does Caching Help? Locality!

• **Temporal Locality** (Locality in Time)
  • Keep recently accessed data items closer to processor

• **Spatial Locality** (Locality in Space)
  • Move contiguous blocks to the upper levels
Recall: Working Set Model

- As a program executes it transitions through a sequence of “working sets” consisting of varying sized subsets of the address space.
Recall: Memory Hierarchy

- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology

---

<table>
<thead>
<tr>
<th>Speed (ns)</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3</td>
<td>100Bs</td>
</tr>
<tr>
<td>1</td>
<td>10kBs</td>
</tr>
<tr>
<td>3</td>
<td>100kB</td>
</tr>
<tr>
<td>10-30</td>
<td>MBs</td>
</tr>
</tbody>
</table>

- Processor
  - Core
    - Registers
    - L1 Cache
    - L2 Cache
  - L3 Cache (shared)
- Main Memory (DRAM): 100 (0.1 ms)
- Secondary Storage (SSD): 100,000 (TBs)
- Secondary Storage (Disk): 10,000,000 (TBs)
Making Address Translation Fast

- Cache results of recent translations
  - Separate from memory cache
  - Cache PTEs using Virtual Page Number as the key
Translation Lookaside Buffer (TLB)

• Record recent Virtual Page # to Physical Frame # translations
• If present in the TLB, can translate address without reading page table
  • Caches the end-to-end result, even if the translation involved multiple levels
• Was invented by Sir Maurice Wilkes – prior to caches
• People realized “if it’s good for page tables, why not the rest of the data in memory?”
• On a TLB miss, the page tables may be cached, so only go to memory when both miss
Caching Applied to Address Translation

- This relies on locality of PTE accesses
  - Instruction accesses spend a lot of time on the same page
  - Stack accesses have locality
  - Data accesses???
The Big Picture

CPU → TLB → Cache → Memory

- MMU traverses page table on miss
- Page Fault trap on translation failure
What Kind of Cache for TLB?

- Remember all those cache design parameters and trade-offs?
- Amount of Data = N * L * K
- Write Policy (write-thru, write-back), Eviction Policy (LRU, ...)

![Diagram of cache structure]

Set Size (k) - Associativity

# of Sets (N)

... ...

tag data ...

line size (L)
How might organization of a TLB differ from that of a conventional instruction or data cache?

Let’s do some review...
Recall: Sources of Cache Misses

- **Compulsory** (cold start or first reference): first access to a block
  - “Cold” fact of life: not a whole lot you can do about it
  - Note: If you are going to run “billions” of instruction, Compulsory Misses are insignificant

- **Capacity**:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size

- **Conflict** (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity

- **Coherence** (Invalidation): other process (e.g., I/O) updates memory
Recall: Finding a Block in a Cache?

- **Block** is minimum quantum of caching
  - Data select field used to select data within block
  - Many caching applications don’t have data select field
- **Tag** used to identify actual the block (what address?)
  - If no candidates match, then declare cache miss
- **Index** Used to Lookup Candidates in Cache
  - Index identifies the set of possibilities (check tag)
Recall: Direct-Mapped Cache

- **Direct Mapped** $2^N$ byte cache:
  - The uppermost ($32 - N$) bits are always the Cache Tag
  - The lowest $M$ bits are the Byte Select (Block Size = $2^M$)

Example: 1 KB Direct Mapped Cache with 32 B Blocks
- Index chooses potential block
- Tag checked to verify block
- Byte select chooses byte within block
Recall: Fully Associative Cache

- **Fully Associative**: Every block can hold any line
  - Address does not include a cache index
  - Compare tags of all cache entries in parallel

Example: Block Size=32B blocks
- We need N 27-bit comparators
- Still have byte select to choose from within block
Recall: Set-Associative Cache

- **N-way set associative**: N entries per Cache Index
  - N direct mapped caches operate in parallel

Example: Two-way set associative cache
- Cache Index selects a “set” from the cache
- Two tags in the set are compared to input in parallel
- Data is selected based on the tag result
Where Does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8-block cache
  - Address space has 32 blocks

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)

Fully associative:
- block 12 can go anywhere
Recall: Which Block to Replace on a Miss?

• Easy for Direct Mapped: Only one possibility
• Set Associative or Fully Associative:
  • Random
  • LRU (Least Recently Used)

• Miss rates for a workload:

<table>
<thead>
<tr>
<th>Size</th>
<th>2-way LRU</th>
<th>Random</th>
<th>4-way LRU</th>
<th>Random</th>
<th>8-way LRU</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
<td>4.7%</td>
<td>5.3%</td>
<td>4.4%</td>
<td>5.0%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
<td>1.5%</td>
<td>1.7%</td>
<td>1.4%</td>
<td>1.5%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
<td>1.13%</td>
<td>1.12%</td>
<td>1.12%</td>
</tr>
</tbody>
</table>
Recall: What Happens on a Write?

- **Write through**: The information is written to both the block in the cache and to the block in the lower-level memory
- **Write back**: The information is written only to the block in the cache
  - Modified cache block is written to main memory only when it is replaced
  - Question is block clean or dirty?
- **Pros and Cons of each?**
  - **WT:**
    - **PRO**: read misses cannot result in writes
    - **CON**: Processor held up on writes unless writes buffered
  - **WB:**
    - **PRO**: repeated writes not sent to DRAM
       processor not held up on writes
    - **CON**: More complex
       Read miss may require writeback of dirty data
What does our understanding of caches tell us about TLB design?
What TLB Organization Makes Sense?

• Needs to be really fast
  • Seems to argue for Direct Mapped or Low Associativity
• However, needs to have very few conflicts!
  • With TLB, the Miss Time extremely high! (PT traversal)
  • Cost of Conflict (Miss Time) is high
  • Hit Time – dictated by clock cycle
• Thrashing: continuous conflicts between accesses
  • What if use low order bits of page as index into TLB?
    • First page of code, data, stack may map to same entry
    • Need 3-way associativity at least?
  • What if use high order bits as index?
    • TLB mostly unused for small programs
TLB Organization: Include Protection

• How big does TLB actually have to be?
  • Usually fewer entries than the cache (why?)
  • Not very big, can support higher associativity
• Small TLBs usually organized as fully-associative cache
  • Lookup is by Virtual Address
  • Returns Physical Address + other info
• What happens when fully-associative is too slow?
  • Put a small (4-16 entry) direct-mapped cache in front
  • Called a “TLB Slice”
• Example for MIPS R3000:

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>Dirty</th>
<th>Ref</th>
<th>Valid</th>
<th>Access</th>
<th>ASID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFA00</td>
<td>0x0003</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>R/W</td>
<td>34</td>
</tr>
<tr>
<td>0x0040</td>
<td>0x0010</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>0x0041</td>
<td>0x0011</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>R</td>
<td>0</td>
</tr>
</tbody>
</table>
Example: R3000 Pipeline Includes TLB Stages

MIPS R3000 Pipeline

<table>
<thead>
<tr>
<th>Inst Fetch</th>
<th>Dcd/ Reg</th>
<th>ALU / E.A</th>
<th>Memory</th>
<th>Write Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>I-Cache</td>
<td>RF</td>
<td>Operation</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E.A.</td>
<td>TLB</td>
<td>D-Cache</td>
</tr>
</tbody>
</table>

TLB

64 entry, on-chip, fully associative, software TLB fault handler

Virtual Address Space

<table>
<thead>
<tr>
<th>ASID</th>
<th>V. Page Number</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>20</td>
<td>12</td>
</tr>
</tbody>
</table>

0xx User segment (caching based on PT/TLB entry)
100 Kernel physical space, cached
101 Kernel physical space, uncached
11x Kernel virtual space

Allows context switching among
64 user processes without TLB flush

• Four different TLBs
  • Instruction TLB for 4K pages
    • 128 entries, 4-way set associative
  • Instruction TLB for large pages
    • 2 entries, fully associative
  • Data TLB for 4K pages
    • 128 entries, 4-way set associative
  • Data TLB for large pages
    • 8 entries, 4-way set associative

• All TLBs use LRU replacement policy
• Why different TLBs for instruction, data, and page sizes?
Example: Intel Nehalem (2008)

- L1 DTLB
  - 64 entries for 4 K pages and
  - 32 entries for 2/4 M pages,

- L1 ITLB
  - 128 entries for 4 K pages using 4-way associativity and
  - 14 fully associative entries for 2/4 MiB pages

- unified 512-entry L2 TLB for 4 KiB pages, 4-way associative
Example: Skylake, Cascade Lake
Current Example: Memory Hierarchy

- Caches (all 64 B line size)
  - L1 I-Cache: 32 KiB/core, 8-way set assoc.
  - L1 D Cache: 32 KiB/core, 8-way set assoc., 4-5 cycles load-to-use, Write-back policy
  - L2 Cache: 1 MiB/core, 16-way set assoc., Inclusive, Write-back policy, 14 cycles latency
  - L3 Cache: 1.375 MiB/core, 11-way set assoc., shared across cores, Non-inclusive victim cache, Write-back policy, 50-70 cycles latency

- TLB
  - L1 ITLB, 128 entries; 8-way set assoc. for 4 KB pages
    - 8 entries per thread; fully associative, for 2 MiB / 4 MiB page
  - L1 DTLB 64 entries; 4-way set associative for 4 KB pages
    - 32 entries; 4-way set associative, 2 MiB / 4 MiB page translations
    - 4 entries; 4-way associative, 1 GiB page translations
  - L2 STLB: 1536 entries; 12-way set assoc. 4 KiB + 2 MiB pages
    - 16 entries; 4-way set associative, 1 GiB page translations
What Happens on a Context Switch?

• Address Space just changed, so TLB entries no longer valid!
• Options?
  • Invalidate TLB (simple but expensive)
  • Include ASID (address space identifier) in TLB
• What if the OS changes the page table?
  • Must invalidate TLB entry!
  • Called “TLB Consistency”
Putting Everything Together

Address Translation
Putting Everything Together

Physical Address:

Virtual Address:

Page Table (1st level)

Page Table (2nd level)

TLB:

Virtual P1 index Virtual P2 index Offset

Physical Page # Offset

Physical Memory:

PageTablePtr

7/21/2020
Putting Everything Together

Virtual Address:
- Virtual P1 index
- Virtual P2 index
- Offset

Physical Address:
- Physical Page #
- Offset

Page Table (1st level)
- Page Table (2nd level)

TLB:
- Tag
- Index
- Byte

Physical Memory:
- Tag
- Block

Cache
Putting Everything Together

Physical Address:

Offset

Virtual Address:

Virtual P1 index

Virtual P2 index

Offset

PageTablePtr

Page Table (1st level)

Page Table (2nd level)

Physical Page #

Offset

Cache:

tag:

index

byte

Physical Memory:

Physical Address:

tag:

block:

...
Summary: Page Tables

• Page Tables
  • Memory divided into fixed-sized chunks of memory
  • Virtual page number from virtual address mapped through page table to physical page number
  • Offset of virtual address same as physical address
  • Large page tables can be placed into virtual memory

• Multi-Level Tables
  • Virtual address mapped to series of tables
  • Permit sparse population of address space

• Inverted Page Table
  • Use of hash-table to hold translation entries
  • Size of page table ~ size of physical memory rather than size of virtual memory
Summary: Caching

• The Principle of Locality:
  • Program likely to access a relatively small portion of the address space at any instant of time.
    • Temporal Locality: Locality in Time
    • Spatial Locality: Locality in Space

• Three (+1) Major Categories of Cache Misses:
  • Compulsory Misses: sad facts of life. Example: cold start misses.
  • Conflict Misses: increase cache size and/or associativity
  • Capacity Misses: increase cache size
  • Coherence Misses: Caused by external processors or I/O devices

• Cache Organizations:
  • Direct Mapped: single block per set
  • Set associative: more than one block per set
  • Fully associative: all entries equivalent
Summary: TLBs

• “Translation Lookaside Buffer” (TLB)
  • Small number of PTEs and optional process IDs (< 512)
  • Fully Associative (Since conflict misses expensive)
  • On TLB miss, page table must be traversed and if located PTE is invalid, cause Page Fault
  • On change in page table, TLB entries must be invalidated
  • TLB is logically in front of cache (need to overlap with cache access)

• On Page Fault, OS can take actions to resolve the situation
  • Demand paging, automatic memory management
  • Make copy of existing page for process
  • On process start, don’t have to load much of executable into memory
  • Rarely used code and data may never get paged in

• Need to handle the exception carefully