CS 422/522  Design & Implementation  
of Operating Systems  

Lecture 13: Address Translation  

Zhong Shao  
Dept. of Computer Science  
Yale University  

Acknowledgement: some slides are taken from previous versions of the CS422/522 lectures taught by Prof. Bryan Ford and Dr. David Wolinsky, and also from the official set of slides accompanying the OSPP textbook by Anderson and Dahlin.  

Main points  

◆ Address translation concept  
  - How do we convert a virtual address to a physical address?  

◆ Flexible address translation  
  - Base and bound  
  - Segmentation  
  - Paging  
  - Multilevel translation  

◆ Efficient address translation  
  - Translation Lookaside Buffers  
  - Virtually and physically addressed caches
**Address translation concept**

- Processor
  - Virtual Address
  - Translation
  - Physical Address
  - Physical Memory
  - Data

**Address translation goals**

- **Memory protection**
- **Memory sharing**
  - Shared libraries, interprocess communication
- **Sparse addresses**
  - Multiple regions of dynamic allocation (heaps/stacks)
- **Efficiency**
  - Memory placement
  - Runtime lookup
  - Compact translation tables
- **Portability**
Bonus feature

◆ What can you do if you can (selectively) gain control whenever a program reads or writes a particular virtual memory location?

◆ Examples:
  - Copy on write
  - Zero on reference
  - Fill on demand
  - Demand paging
  - Memory mapped files
  - ...

A Preview: MIPS address translation

◆ Software-Loaded Translation lookaside buffer (TLB)
  - Cache of virtual page -> physical page translations
  - If TLB hit, physical address
  - If TLB miss, trap to kernel
  - Kernel fills TLB with translation and resumes execution

◆ Kernel can implement any page translation
  - Page tables
  - Multi-level page tables
  - Inverted page tables
  - ...
A Preview: MIPS lookup

Virtually addressed base and bounds
Virtually addressed base and bounds

◆ Pros?
  - Simple
  - Fast (2 registers, adder, comparator)
  - Safe
  - Can relocate in physical memory without changing process
◆ Cons?
  - Can't keep program from accidentally overwriting its own code
  - Can't share code/data with other processes
  - Can't grow stack/heap as needed

Segmentation

◆ Segment is a contiguous region of virtual memory
◆ Each process has a segment table (in hardware)
  - Entry in table = segment
◆ Segment can be located anywhere in physical memory
  - Each segment has: start, length, access permission
◆ Processes can share segments
  - Same start, length, same/different access permissions
Segmentation

UNIX fork and copy on write

- UNIX fork
  - Makes a complete copy of a process

- Segments allow a more efficient implementation
  - Copy segment table into child
  - Mark parent and child segments read-only
  - Start child process; return to parent
  - If child or parent writes to a segment (ex: stack, heap)
    * trap into kernel
    * make a copy of the segment and resume
Zero-on-reference

- How much physical memory is needed for the stack or heap?
  - Only what is currently in use

- When program uses memory beyond end of stack
  - Segmentation fault into OS kernel
  - Kernel allocates some memory
    - How much?
  - Zeros the memory
    - Avoid accidentally leaking information!
  - Modify segment table
  - Resume process
Segmentation

◆ Pros?
- Can share code/data segments between processes
- Can protect code segment from being overwritten
- Can transparently grow stack/heap as needed
- Can detect if need to copy-on-write

◆ Cons?
- Complex memory management
  * Need to find chunk of a particular size
- May need to rearrange memory from time to time to make room for new segment or growing segment
  * External fragmentation: wasted space between chunks

Paged translation

◆ Manage memory in fixed size units, or pages
◆ Finding a free page is easy
  - Bitmap allocation: 0011111100000001100
  - Each bit represents one physical page frame
◆ Each process has its own page table
  - Stored in physical memory
  - Hardware registers
    * pointer to page table start
    * page table length
Paging and copy on write

- Can we share memory between processes?
  - Set entries in both page tables to point to same page frames
  - Need core map of page frames to track which processes are pointing to which page frames (e.g., reference count)
- UNIX fork with copy on write
  - Copy page table of parent into child process
  - Mark all pages (in new and old page tables) as read-only
  - Trap into kernel on write (in child or parent)
  - Copy page
  - Mark both as writeable
  - Resume execution

Fill on demand

- Can I start running a program before its code is in physical memory?
  - Set all page table entries to invalid
  - When a page is referenced for first time, kernel trap
  - Kernel brings page in from disk
  - Resume execution
  - Remaining pages can be transferred in the background while program is running
Sparse address spaces

- Might want many separate dynamic segments
  - Per-processor heaps
  - Per-thread stacks
  - Memory-mapped files
  - Dynamically linked libraries

- What if virtual address space is large?
  - 32-bits, 4KB pages => 500K page table entries
  - 64-bits => 4 quadrillion page table entries

Multi-level translation

- Tree of translation tables
  - Paged segmentation
  - Multi-level page tables
  - Multi-level paged segmentation

- Fixed-size page as lowest level unit of allocation
  - Efficient memory allocation (compared to segments)
  - Efficient for sparse addresses (compared to paging)
  - Efficient disk transfers (fixed size units)
  - Easier to build translation lookaside buffers
  - Efficient reverse lookup (from physical -> virtual)
  - Variable granularity for protection/sharing
Paged segmentation

- Process memory is segmented
- Segment table entry:
  - Pointer to page table
  - Page table length (# of pages in segment)
  - Access permissions
- Page table entry:
  - Page frame
  - Access permissions
- Share/protection at either page or segment-level

Paged segmentation (implementation)
Multilevel paging

X86 multilevel paged segmentation

- **Global Descriptor Table (segment table)**
  - Pointer to page table for each segment
  - Segment length
  - Segment access permissions
  - Context switch: change global descriptor table register (GDTR, pointer to global descriptor table)

- **Multilevel page table**
  - 4KB pages; each level of page table fits in one page
  - 32-bit: two level page table (per segment)
  - 64-bit: four level page table (per segment)
  - Omit sub-tree if no valid addresses
Multilevel translation

◆ Pros:
  - Allocate/fill only page table entries that are in use
  - Simple memory allocation
  - Share at segment or page level
◆ Cons:
  - Space overhead: one pointer per virtual page
  - Two (or more) lookups per memory reference

Portability

◆ Many operating systems keep their own memory translation data structures
  - List of memory objects (segments)
  - Virtual page \(\rightarrow\) physical page frame
  - Physical page frame \(\rightarrow\) set of virtual pages
◆ One approach: Inverted page table
  - Hash from virtual page \(\rightarrow\) physical page
  - Space proportional to \# of physical pages
Efficient address translation

◆ Translation lookaside buffer (TLB)
  - Cache of recent virtual page -> physical page translations
  - If cache hit, use translation
  - If cache miss, walk multi-level page table
◆ Cost of translation =
  Cost of TLB lookup +
  Prob(TLB miss) * cost of page table lookup

TLB and page table translation
**TLB lookup**

**MIPS software loaded TLB**

- **Software defined translation tables**
  - If translation is in TLB, ok
  - If translation is not in TLB, trap to kernel
  - Kernel computes translation and loads TLB
  - Kernel can use whatever data structures it wants

- **Pros/cons?**
Question

- What is the cost of a TLB miss on a modern processor?
  - Cost of multi-level page table walk
  - MIPS: plus cost of trap handler entry/exit

Hardware design principle

The bigger the memory, the slower the memory
Intel i7

Integrated Memory Controller - 3 Ch DDR3

Core 0 Core 1 Core 2 Core 3

Shared L3 Cache

Memory hierarchy

<table>
<thead>
<tr>
<th>Cache</th>
<th>Hit Cost</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st level cache/first level TLB</td>
<td>1 ns</td>
<td>64 KB</td>
</tr>
<tr>
<td>2nd level cache/second level TLB</td>
<td>4 ns</td>
<td>256 KB</td>
</tr>
<tr>
<td>3rd level cache</td>
<td>12 ns</td>
<td>2 MB</td>
</tr>
<tr>
<td>Memory (DRAM)</td>
<td>100 ns</td>
<td>10 GB</td>
</tr>
<tr>
<td>Data center memory (DRAM)</td>
<td>100 μs</td>
<td>100 TB</td>
</tr>
<tr>
<td>Local non-volatile memory</td>
<td>100 μs</td>
<td>100 GB</td>
</tr>
<tr>
<td>Local disk</td>
<td>10 ms</td>
<td>1 TB</td>
</tr>
<tr>
<td>Data center disk</td>
<td>10 ms</td>
<td>100 PB</td>
</tr>
<tr>
<td>Remote data center disk</td>
<td>200 ms</td>
<td>1 XB</td>
</tr>
</tbody>
</table>

i7 has 8MB as shared 3rd level cache; 2nd level cache is per-core
Question

- What is the cost of a first level TLB miss?
  - Second level TLB lookup
- What is the cost of a second level TLB miss?
  - x86: 2-4 level page table walk
- How expensive is a 4-level page table walk on a modern processor?

Virtually addressed vs. physically addressed caches

- Too slow to first access TLB to find physical address, then look up address in the cache
- Instead, first level cache is virtually addressed
- In parallel, access TLB to generate physical address in case of a cache miss
When do TLBs work/not work?

- **Video Frame Buffer**: 32 bits x 1K x 1K = 4MB

<table>
<thead>
<tr>
<th>Page#</th>
<th>Video Frame Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1021</td>
<td></td>
</tr>
<tr>
<td>1022</td>
<td></td>
</tr>
<tr>
<td>1023</td>
<td></td>
</tr>
</tbody>
</table>

Superpages

- On many systems, TLB entry can be
  - A page
  - A superpage: a set of contiguous pages

- x86: superpage is set of pages in one page table
  - x86 TLB entries
    * 4KB
    * 2MB
    * 1GB
Superpages

When do TLBs work/not work, part 2

- What happens when the OS changes the permissions on a page?
  - For demand paging, copy on write, zero on reference, ...

- TLB may contain old translation
  - OS must ask hardware to purge TLB entry

- On a multicore: TLB shootdown
  - OS must ask each CPU to purge TLB entry
When do TLBs work/not work, part 3

- What happens on a context switch?
  - Reuse TLB?
  - Discard TLB?

- Solution: Tagged TLB
  - Each TLB entry has process ID
  - TLB hit only if process ID matches current process
Aliasing

- **Alias:** two (or more) virtual cache entries that refer to the same physical memory
  - A consequence of a tagged virtually addressed cache!
  - A write to one copy needs to update all copies

- **Typical solution**
  - Keep both virtual and physical address for each entry in virtually addressed cache
  - Lookup virtually addressed cache and TLB in parallel
  - Check if physical address from TLB matches multiple entries, and update/invalidate other copies
Multicore and hyperthreading

- Modern CPU has several functional units
  - Instruction decode
  - Arithmetic/branch
  - Floating point
  - Instruction/data cache
  - TLB
- Multicore: replicate functional units (i7: 4)
  - Share second/third level cache, second level TLB
- Hyperthreading: logical processors that share functional units (i7: 2)
  - Better functional unit utilization during memory stalls
- No difference from the OS/programmer perspective
  - Except for performance, affinity, ...

Address translation uses

- Process isolation
  - Keep a process from touching anyone else’s memory, or the kernel’s
- Efficient inter-process communication
  - Shared regions of memory between processes
- Shared code segments
  - E.g., common libraries used by many different programs
- Program initialization
  - Start running a program before it is entirely in memory
- Dynamic memory allocation
  - Allocate and initialize stack/heap pages on demand
Address translation (more)

- Cache management
  - Page coloring
- Program debugging
  - Data breakpoints when address is accessed
- Zero-copy I/O
  - Directly from I/O device into/out of user memory
- Memory mapped files
  - Access file data using load/store instructions
- Demand-paged virtual memory
  - Illusion of near-infinite memory, backed by disk or memory on other machines

Address translation (even more)

- Checkpointing/restart
  - Transparently save a copy of a process, without stopping the program while the save happens
- Persistent data structures
  - Implement data structures that can survive system reboots
- Process migration
  - Transparently move processes between machines
- Information flow control
  - Track what data is being shared externally
- Distributed shared memory
  - Illusion of memory that is shared between machines