Address Translation
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
Why Address Translation?
Address Translation Concept
Address Translation Goals

• Memory protection
• Memory sharing
• Flexible memory placement
• Sparse addresses
• Runtime lookup efficiency
• Compact translation tables
• Portability
Address Translation

• What can you do if you can (selectively) gain control whenever a program reads or writes a particular memory location?
  – With hardware support
  – With compiler-level support

• Memory management is one of the most complex parts of the OS
  – Serves many different purposes
Address Translation Uses

• Process isolation
  – Keep a process from touching anyone else’s memory, or the kernel’s

• Efficient interprocess 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
Base and Bounds (Abstract)
Base and Bounds (Implementation)
Base and Bounds

• Pros?
  – Simple
  – Fast (2 registers, adder, comparator)
  – 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 memory
  - Virtual or (for now) physical memory
- Each process has a segment table (in hardware)
  - Entry in table = segment
- Segment can be located anywhere in physical memory
  - Start
  - Length
  - Access permission
- Processes can share segments
  - Same start, length, same/different access permissions
Segmentation (Abstract)
Segmentation (Implementation)

Processor’s View

Virtual Memory

Processor

Virtual Address

Segment Offset

Segment Table

Base Bound Access

Read

R/W

R/W

Raise Exception

Physical Memory

Code

Base 0

Base + Bound 0

Base 1

Base + Bound 1

Base 2

Base + Bound 2

Base 3

Base + Bound 3

Data

Heap

Stack

Code

Data

Heap

Stack
<table>
<thead>
<tr>
<th>Segment start</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>0x700</td>
</tr>
<tr>
<td>data</td>
<td>0x500</td>
</tr>
<tr>
<td>heap</td>
<td>-</td>
</tr>
<tr>
<td>stack</td>
<td>0x1000</td>
</tr>
</tbody>
</table>

### Virtual Memory

- **main**: 240
- **store #1108, r2**
- **244**: store pc+8, r31
- **248**: jump 360
- **24c**: ...
- **strlen**: 360
- **loadbyte (r2), r3**
- **420**: jump (r31)
- **...**
- **x**: 1108
- **a b c \0**

### Physical Memory

- **x**: 108
- **a b c \0**
- **main**: 4240
- **store #1108, r2**
- **4244**: store pc+8, r31
- **4248**: jump 360
- **424c**: ...
- **strlen**: 4360
- **loadbyte (r2), r3**
- **...**
- **x**: 1108
- **a b c \0**
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, will trap into kernel
    • make a copy of the segment and resume
Zero-on-Reference

• How much physical memory do we need to allocate for the stack or heap?
  – Zero bytes!

• When program touches the heap
  – Segmentation fault into OS kernel
  – Kernel allocates some memory
    • How much?
  – Zeros the memory
    • avoid accidentally leaking information!
  – Restart 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: 00111111000000001100
  – 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
Paged Translation (Abstract)
Paged Translation (Implementation)
Paging Questions

• What must be saved/restored on a process context switch?
  – Pointer to page table/size of page table
  – Page table itself is in main memory

• What if page size is very small?

• What if page size is very large?
  – Internal fragmentation: if we don’t need all of the space inside a fixed size chunk
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

• UNIX fork with copy on write at page granularity
  – Copy page table entries to new process
  – Mark all pages as read-only
  – Trap into kernel on write (in child or parent)
  – Copy page and resume execution
Paging and Fast Program Start

• 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
    • Trap to OS kernel
    • OS kernel brings in page
    • Resumes execution
  – Remaining pages can be transferred in the background while program is running
Sparse Address Spaces

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

• What if virtual address space is sparse?
  – On 32-bit UNIX, code starts at 0
  – Stack starts at $2^{31}$
  – 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

• All 3: Fixed size page as lowest level unit
  – Efficient memory allocation
  – Efficient disk transfers
  – Easier to build translation lookaside buffers
  – Efficient reverse lookup (from physical -> virtual)
  – Page 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 (Abstract)
Paged Segmentation (Implementation)
Multilevel Paging

Implementation

Processor

Virtual Address

Index 1  Index 2  Index 3  Offset

Level 1

Frame  Offset

Level 2

Level 3

Physical Memory
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
    • Only fill page table if needed
  – 32-bit: two level page table (per segment)
  – 64-bit: four level page table (per segment)
Multilevel Translation

• Pros:
  – Allocate/fill only as many page tables as used
  – Simple memory allocation
  – Share at segment or page level

• Cons:
  – Space overhead: at least 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 -> physical
  – Physical -> virtual
  – Simplifies porting from x86 to ARM, 32 bit to 64 bit

• Inverted page table
  – Hash from virtual page -> physical page
  – Space proportional to # of physical pages
Do we need multi-level page tables?

- Use inverted page table in hardware instead of multilevel tree
  - IBM PowerPC
  - Hash virtual page # to inverted page table bucket
  - Location in IPT => physical page frame

- Pros/cons?
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 (Abstract)

Processor → TLB
  
  Virtual Address → TLB
    
    Hit → Frame
      
      Offset → Physical Memory
        
        Physical Address → Physical Memory
          
          Data → Physical Memory
            
            Physical Memory → TLB
              
              Invalid → Page Table
                
                Valid → Frame
                  
                  Page Table → TLB
                    
                    Miss → TLB
                      
                      Virtual Address → TLB
Software Loaded TLB

• Do we need a page table at all?
  – MIPS processor architecture
  – 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?
When Do TLBs Work/Not Work?

Video Frame Buffer

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

...
When Do TLBs Work/Not Work?

- Video Frame Buffer: 32 bits x 1K x 1K = 4MB
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
When Do TLBs Work/Not Work, part 2

• What happens on a context switch?
  – Reuse TLB?
  – Discard TLB?

• Motivates hardware tagged TLB
  – Each TLB entry has process ID
  – TLB hit only if process ID matches current process
Physical Memory

Processor

Virtual Address

Page Address

Page# Offset

Process ID

Matching Entry

Page Table Lookup

Translation Lookaside Buffer (TLB)

Process ID Page Frame Access

Physical Address

Frame Offset

Page Frame
When Do TLBs Work/Not Work, part 3

• 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
# TLB Shootdown

## Processor 1 TLB

<table>
<thead>
<tr>
<th>Process ID</th>
<th>VirtualPage</th>
<th>PageFrame</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0053</td>
<td>0x003</td>
<td>R/W</td>
</tr>
<tr>
<td>1</td>
<td>0x40FF</td>
<td>0x0012</td>
<td>R/W</td>
</tr>
</tbody>
</table>

## Processor 2 TLB

<table>
<thead>
<tr>
<th>Process ID</th>
<th>VirtualPage</th>
<th>PageFrame</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0053</td>
<td>0x0003</td>
<td>R/W</td>
</tr>
<tr>
<td>0</td>
<td>0x0001</td>
<td>0x0005</td>
<td>Read</td>
</tr>
</tbody>
</table>

## Processor 3 TLB

<table>
<thead>
<tr>
<th>Process ID</th>
<th>VirtualPage</th>
<th>PageFrame</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x40FF</td>
<td>0x0012</td>
<td>R/W</td>
</tr>
<tr>
<td>0</td>
<td>0x0001</td>
<td>0x0005</td>
<td>Read</td>
</tr>
</tbody>
</table>
# 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
Hardware Design Principle

The bigger the memory, the slower the memory
Virtually Addressed Caches

- Processor
  - Virtual Address
    - Virtual Cache
      - Hit
        - Data
      - Miss
        - Offset
          - Physical Address
            - Physical Memory
              - Data
- TLB
  - Hit
    - Frame
  - Miss
- Page Table
  - Invalid
    - Raise Exception
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?
Questions

• With a virtual cache, what do we need to do on a context switch?
• What if the virtual cache > page size?
  – Page size: 4KB (x86)
  – First level cache size: 64KB (i7)
  – Cache block size: 32 bytes
Aliasing

• Alias: two (or more) virtual cache entries that refer to the same physical memory
  – What if we modify one alias and then context switch?

• Typical solution
  – On a write, lookup virtual cache and TLB in parallel
  – Physical address from TLB used to check for aliases
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, ...