Address Translation
Main Points

• Address Translation Concept
  – How do we convert a virtual address to a physical address?

• Flexible Address Translation
  – Segmentation
  – Paging
  – Multilevel translation

• Efficient Address Translation
  – Translation Lookaside Buffers
  – Virtually and physically addressed caches
Address Translation Concept

Diagram:
- Processor
- Translation
- Physical Memory
- Data
- Virtual Address
- Physical Address
- Valid
- Invalid
- Raise Exception
Address Translation Goals

• Memory protection
  – Isolate process to its only memory
  – Prevent virus from re-writing machine instructions
• Memory sharing
  – Shared libraries, interprocess communication
• Sparse addresses
  – Dynamically allocated regions: heaps, stacks, mmap
• Efficiency
  – Reduce fragmentation and copying
  – Runtime lookup cost and TLB hit rate
  – Translation table size
• Portability
Bonus Feature

• What if the kernel can regain 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
  – ...

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
Process Regions or Segments

• Every process has logical regions or segments
  – Contiguous region of process memory
• Code, data, heap, stack, dynamic library (code, data), memory mapped files, ...
• Each with its own
  – protection: read-only, read-write, execute-only
  – sharing: code vs. data
  – access pattern: code vs. mmap file
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

Processor’s View

Virtual Memory

- Processor
- Virtual Address
- Code
- Data
- Heap
- Stack

Implementation

- Processor
- Virtual Address
- Segment
- Offset

Segment Table

- Base
- Bound
- Access
- Read
  - R/W
  - R/W
  - R/W

Physical Memory

- Stack
- Base+ Bound
- Code
- Base+ Bound
- Base
- Data
- Base+ Bound
- Heap
- Base+ Bound

Physical Address

- Raise Exception
### Segment start & length

<table>
<thead>
<tr>
<th>Segment</th>
<th>Start (hex)</th>
<th>Length (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>0x4000</td>
<td>0x700</td>
</tr>
<tr>
<td>data</td>
<td>0</td>
<td>0x500</td>
</tr>
<tr>
<td>heap</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>stack</td>
<td>0x2000</td>
<td>0x1000</td>
</tr>
</tbody>
</table>

### Virtual Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Store/Load</th>
<th>Data</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>main: 0:240</td>
<td>#108, r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:244</td>
<td>pc+8, r31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:248</td>
<td>jump 360</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:24c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strlen: 0:360</td>
<td>loadbyte (r2), r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:420</td>
<td>jump (r31)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x: 1:108</td>
<td>a b c \0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Physical Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Store/Load</th>
<th>Data</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 108</td>
<td>a b c \0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main: 4240</td>
<td>#1108, r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4244</td>
<td>pc+8, r31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4248</td>
<td>jump 360</td>
<td></td>
<td></td>
</tr>
<tr>
<td>424c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strlen: 4360</td>
<td>loadbyte (r2), r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4420</td>
<td>jump (r31)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Question

• With segmentation, what is saved/restored on a process context switch?
Segmentation

• Pros?
  – Can share code/data segments between processes
  – Can protect code segment from being overwritten

• Cons? Complex memory management
  – Need to find chunk of a particular size
  – May need to rearrange memory to make room for
    new segment or growing segment (e.g., sbrk)
  – 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 Questions

• With paging, what is 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 Sharing

• Can we use page tables to share memory between processes?

• Set page tables to point to same page frame

• Need core map
  – Array of information about each physical page frame
  – Set of processes pointing to that page frame
  – When reference count goes to zero, can reclaim!
Question

• How big a user stack should I allocate?

• What if some programs need a large stack and others need a small one?
Expand Stack on Reference

• When program references memory beyond end of stack
  – Page fault into OS kernel
  – Kernel allocates some additional memory
    • How much?
  – Remember to zero the memory to avoid accidentally leaking information!
  – Modify page table
  – Resume process
UNIX fork seems inefficient

- Makes a complete copy of process
- Throw copy away on exec
- Do we need to make the copy?
  - One solution: change the syscall interface!
Copy on Write

- Paging allows an efficient fork
  - Copy page table of parent into child
  - Mark all pages (in new/old page tables) as read-only
  - Start child process; restart parent
  - Trap into kernel on write (in child or parent)
  - Copy page
  - Mark both as writeable
  - Resume execution
Question

• Can I start running a program before all of its code is in memory?
Fill On Demand

- 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
Beyond Paging: 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 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

• All have pages as lowest level; why?
Multilevel Translation with Pages at Lowest Level

• Efficient memory allocation (vs. segments)
• Efficient for sparse addresses (vs. 1 level paging)
• Efficient disk transfers (fixed size units)
• Easier to build translation lookaside buffers
• Efficient reverse lookup (from physical -> virtual)
• Variable granularity for protection/sharing

• Except: see discussion of superpages
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 memory or set access permissions at either page or segment-level
Question

• With paged segmentation, what must be saved/restored across a process context switch?
Multi-level or hierarchical page tables

• Ex: 2-level page table
  – Level 1 table: each PTE points to a page table
  – Level 2 table: each PTE points to a page

• Can share/protect/page in/out at either level 1 or level 2
Implementation

Processor

Virtual Address

Index 1  Index 2  Index 3  Offset

Level 1

Level 2

Level 3

Physical Memory

Physical Address

Frame  Offset
Question

• Write pseudo-code for translating a virtual address to a physical address for a system using 3-level paging, with 8 bits of address per level
x86 Multilevel Paged Segmentation

• Global Descriptor Table (segment table)
• Each segment descriptor
  – Pointer to (multilevel) page table
  – Segment length
  – Segment access permissions
• Context switch
  – change global descriptor table register (GDTR), pointer to global descriptor table
  – Side effect: invalidates TLB
x86 Multilevel Paging

- Omit sub-tree if no valid addresses
  - Good for sparse address space
- 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)
x86-32 Paging
x86-64 Paging

Virtual address

16
Sign
9
VPN1
9
VPN2
9
VPN3
9
VPN4
12
VPO

Page Map
Level 4

Page Directory Pointer Table

Page Directory

Page Table

CR3

PM4LE

PDPE

PDE

PTE

36
PPN

12
PPO

Physical address
# Page directory entries (x86 32 bit)

<table>
<thead>
<tr>
<th>Empty</th>
<th>Ignored</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>4MB page</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Bits 31:22 of address of 4MB page frame</td>
<td>Ign</td>
<td>G 1 D A P C D P W / U R / W 1</td>
</tr>
<tr>
<td>Page table</td>
<td>Ign</td>
<td>0</td>
</tr>
<tr>
<td>Bits 31:12 of address of page table</td>
<td>Ign</td>
<td>G 1 D A P C D P W / U R / W 1</td>
</tr>
</tbody>
</table>
# Page table entries (x86 32 bit)

<table>
<thead>
<tr>
<th>Empty</th>
<th>Ignored</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB page</td>
<td>Bits 31:12 of address of page frame</td>
<td>Ign</td>
</tr>
</tbody>
</table>
Small page translation

CR3 register

Virtual address

Address of PDE

PDE

Address of PTE

PTE

Physical address

Address of page directory  SBZ

PDE index  PTE index  Offset

Page table base address  Access control 1

Page table base address  PTE index 0 0

Small page base address  Access control 1

Small page base address  Offset

Note: addresses in physical memory!
Page table for small pages

Base address

1024 entries 4kB

Page directory

Translates VA[31:22]

Translates VA[21:12]

Page table

1024 entries 4kB

Page table

4kB page

Translates VA[11:0] = offset in page

4kB page
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
  – Multiple lookups per memory reference
Page Translation in the OS

• OS’s need to keep their own data structures
  – List of memory objects (segments)
  – Virtual page -> physical page frame
  – Physical page frame -> set of virtual pages
  – Keep track of copy on write, load on demand, ...

• Why not just use the hardware page tables?
Kernel Page Translation

• Kernel maintains its own page translation data structures
  – Portable, flexible
  – Copy changes down into hardware page tables

• Example: Inverted page table
  – Hash from virtual page -> physical page
  – Space proportional to # of physical pages

• Example: virtual/shadow page table
  – Linux kernel tables mirror x86 structure, even on ARM
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
Translation Lookaside Buffer (TLB)

Matching Entry

Page Table
Lookup

Virtual Address

Page#  Offset

Physical Memory

Frame  Offset

Physical Address

TLB Lookup
Question

• What happens on a context switch?
  – Reuse TLB?
  – Discard TLB? (xk resets TLB)

• Solution: Tagged TLB
  – Each TLB entry has process ID
  – TLB hit only if process ID matches current process
Translation Lookaside Buffer (TLB)
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
  - ...
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
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?
  – 64 bit x86: 4-level page table walk

• How expensive is a 4-level page table walk?
Virtually Addressed vs. Physically Addressed Caches

• First level cache has at most a few cycles
  – Delays every instruction fetch and data reference
• Lookup TLB to get physical address, then lookup physical address in the cache?
  – Too slow!
• Instead, lookup virtual address in cache
• In parallel, lookup TLB in case of a cache miss
Virtually Addressed Caches

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

• With a virtual cache, what do we need to do on a context switch?
Physically Addressed Cache

Processor - Virtual Address

Virtual Cache - Virtual Address Miss

TLB - Virtual Address Miss

Page Table - Invalid

Raise Exception

Data - Hit

Frame - Hit

Offset

Physical Cache - Miss

Physical Memory - Data
TLB Size (Intel Kaby Lake, 2017)

First level TLB
- Instruction: 128 entries
- Data: 64 entries

Second level TLB
- 1536 entries

Modern server can have 10 TB (!) of DRAM
- 10-20% of server CPU time spent in TLB misses
When Do TLBs Work/Not Work?

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

2017 laptop: 3K x 2K = 24MB

4K display: 4K x 3K = 48MB
Superpages

• On x86 and ARM, TLB entry can be
  – A page
  – A superpage: a set of contiguous, aligned pages

• x86: superpage is set of pages in one page table
  – One page: 4KB
  – One page table: 2MB
  – One page table of page tables: 1GB
  – One page table of page tables of page tables: 0.5TB
Superpages

<table>
<thead>
<tr>
<th>Page#</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>Offset</td>
</tr>
</tbody>
</table>

Translation Lookaside Buffer (TLB)

Matching Entry

Matching Superpage

Page Table Lookup

Physical Address

Frame Offset

SF Offset

Superpage Superframe (SP) or Page# (SF) or Frame Access
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
### TLB Shootdown

<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>1</td>
<td>0x40FF</td>
<td>0x0012</td>
<td>R/W</td>
</tr>
<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>
<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>
Virtual Cache Shootdown

• When permissions change for a page, we must shoot down the TLB entry on every CPU
• What about the contents of the virtual cache?
• Lazy shootdown of the virtual cache:
  – Lookup virtually addressed cache and TLB in \textit{parallel}
  – Use the TLB to verify virtual address is still valid!
  – Evict entry from cache if not
Virtual Cache Aliases

• 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

• Solution:
  – Virtual cache keeps both virtual and physical address for each entry
  – Lookup virtually addressed cache and TLB in parallel
  – Check if physical address from TLB matches any other entries, and update/invalidate those copies
x86 caches

- 64 byte line size
- Physically indexed
- Physically tagged
- Write buffer
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 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