Address Translation

Part 2
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 of Address Translation Concept]

- Processor
  - Virtual Address
  - Data

- Translation
  - Invalid
  - Valid
  - Physical Address

- Physical Memory
  - Physical Memory
  - Data

- MMU
  - Raise Exception
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
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

Virtual address

Page Directory

Page Table

Physical address

VPN3

VPN4

VPO

PDE

PTE

PPN

PPO

CR3
x86-64 Paging

Virtual address

<table>
<thead>
<tr>
<th>Sign</th>
<th>9</th>
<th>9</th>
<th>9</th>
<th>9</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPN1</td>
<td>VPN2</td>
<td>VPN3</td>
<td>VPN4</td>
<td>VPO</td>
<td></td>
</tr>
</tbody>
</table>

Page Map

Page Directory Pointer Table

Page Directory

Page Table

CR3

Page Map Level 4

PDPE

PDE

PTE

Virtual address

Physical address

48 bits of Virtual ADDRESS

AWS: 16 TB servers

48 bits of Physical ADDRESS

256 TB
Page table entries (x86 32 bit)

Empty

4KB page

Bits 31:12 of address of page frame

Ignored

Ign

Access (use bit)

MODIFIED

DIRTY

page frame #

(ignored)

invalid

Cacheable (I/O)

Write through (I/O)

Persistent RAM

User / Supervisor

Read-write or read-only
### Page directory entries (x86 32 bit)

<table>
<thead>
<tr>
<th>Empty</th>
<th>Ignored</th>
<th>0</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4MB page</td>
<td></td>
<td></td>
<td>G</td>
<td>D</td>
<td>A</td>
<td>P</td>
<td>P</td>
<td>U</td>
</tr>
<tr>
<td>Page table</td>
<td></td>
<td></td>
<td>G</td>
<td>A</td>
<td>C</td>
<td>W</td>
<td>P</td>
<td>U</td>
</tr>
</tbody>
</table>

- Bits 31:22 of address of 4MB page frame
- Bits 31:12 of address of page table

- **Ignored**: Bits are ignored.
- **VALID**: Mark indicates the page is valid.
- **USE BIT**: Indicates the use bit.
- **PAGE FRAME SUPERPAGE**: Pointer to page frame.
- **PTE to page**: Pointer to page table entry.
- **DIRTY BIT**: Indicates if the page is dirty.
- **USER OR SUPERVISOR ONLY**: Determines access level.
- **READ ONLY**: Indicates read-only access.
- **WRITE THROUGH**: Indicates write-through mode.
- **CACHEABLE**: Indicates cacheability.
- **DISABLE (I/O)**: Indicates disabling I/O.
Small page translation

- CR3 register
- Address of page directory
- Address of page directory
- PDE index
- Page table base address
- Small page base address
- Access control
- Offset

Note: addresses in physical memory!
Base address
1024 entries
4kB

Page table for small pages

Page directory
Translates VA[31:22]

Translates VA[21:12]
Page table

1024 entries
4kB

Page table

4kB page

VA[11:0] = offset in page
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

```
PTD[ADDR >> 24], PT[ADDR >> 16 & 0xFF]
```
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
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