Virtual Memory Design Issues: Address Translation

Jaswinder Pal Singh
Computer Science Department
Princeton University
(http://www.cs.princeton.edu/courses/cos318/)

Design Issues

- Discussed Address Translation (will refresh in context)
- Thrashing and working sets
- Backing store
- Multiple page sizes and PT entries
- Pinning/locking pages
- Zero pages
- Shared pages
- Copy-on-write
- Distributed shared memory
- Virtual memory in Unix and Linux
- Virtual memory in Windows 2000

Virtual Memory Design Goals

- Protection
- Virtualization
  - Use disk to extend physical memory
  - Make virtualized memory user friendly (0 to high address)
- Enabling memory sharing (libraries, communication)
- Efficiency
  - Translation efficiency (TLB as cache)
  - Access efficiency
    - Access time = h \cdot \text{memory access time} + (1 - h) \cdot \text{disk access time}
    - E.g. Suppose memory access time = 100ns, disk access time = 10ms
    - If h = 90%, VM access time is 1ms!
- Portability

Recall Address translation: Base and Bound

- Pros: Simple, fast, cheap, safe, can relocate
- 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 (stop program, change reg, ...)

Pros: Simple, fast, cheap, safe, can relocate
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 (stop program, change reg, ...)

**Pros:**
- Simple
- Fast
- Cheap
- Safe
- Can relocate

**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 (stop program, change reg, ...)

**Key Points:**
- Use disk to extend physical memory
- Make virtual memory user-friendly (0 to high address)
- Enabling memory sharing (libraries, communication)
- Translation efficiency (TLB as cache)
- Access efficiency
- Portability

**Diagram:**
- Processor
- Virtual Memory
- Base
- Bound
- Exception
- Implementation
- Physical Memory

- **Base + Bound:** Base and Bound address translation
- **Pros:** Simple, fast, cheap, safe, can relocate
- **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 (stop program, change reg, ...)

**Summary:**
- Address translation is a fundamental aspect of virtual memory design.
- Various issues and goals are discussed to ensure efficient and safe memory management.
- Understanding basics of memory management is crucial for operating systems design.

---

COS 318: Operating Systems
Virtual Memory Design Issues: Address Translation

Jaswinder Pal Singh
Computer Science Department
Princeton University
(http://www.cs.princeton.edu/courses/cos318/)
Recall Address Translation: Segmentation

- A segment is a contiguous region of virtual memory
- Every process has a segment table (in hardware)
  - Entry in table oer 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

Segments Enable Copy-on-Write

- Idea of Copy-on-Write
  - Child process inherits copy of parent’s address space on fork
  - But don’t really want to make a copy of all data upon fork
  - Would like to share as far as possible and make own copy only “on-demand”, i.e. upon a write
- Segments allow this to an extent
  - Copy segment table into child, not entire address space
  - Mark all parent and child segments read-only
  - Start child process; return to parent
  - If child or parent writes to a segment (e.g. stack, heap)
    - Trap into kernel
    - At this point, make a copy of the segment, and resume

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 mgmt: need to find chunk of 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
Recall Address Translation: Paging

- Manage memory in fixed size units, or pages
- Finding a free page is easy
  - Effectively bitmap allocation: 00111110000001100
  - Each bit represents one physical page frame
- Every process has its own page table
  - Stored in physical memory
  - Hardware registers
    - Pointer to page table start
    - Page table length
- Recall fancier structures: segmentation+paging, multi-level PT
  - Better for sparse virtual address spaces
  - E.g. per-processor heaps, per-thread stacks, memory mapped files, dynamically linked libraries, ...
  - Don’t have fine-grain page table entries for “holes”

Sharing and Copy on Write with Paging

- Can we share memory between processes?
  - Entries in both page tables to point to same page frames
  - Need core map of page frames to track which / how many processes are pointing to which page frames (e.g., reference count), so know when a a page is still “live”
- 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

Pinning (or Locking) Page Frames

- When do you need it?
  - When DMA is in progress, you don’t want to page the pages out to avoid CPU from overwriting the pages
- Mechanism?
  - A data structure to remember all pinned pages
  - Paging algorithm checks the data structure to decide on page replacement
  - Special calls to pin and unpin certain pages
Zeroing Pages

- Initialize pages to all zero values
  - Heap and static data are initialized
- How to implement?
  - On the first page fault on a data page or stack page, zero it
  - Or, have a special thread zeroing pages in the background

Shared Pages

- PTEs from two processes share the same physical pages
  - What use cases?
- Implementation issues
  - What if you terminate a process with shared pages
  - Paging in/out shared pages
  - Pinning, unpinning shared pages
  - Deriving the working set for a process with shared pages

Efficient address translation

- Recall translation lookaside buffer (TLB)
  - Cache of recent virtual page -> physical page translations
  - If cache hit, use translation
  - If cache miss, walk (perhaps 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 Performance

- What is the cost of a TLB miss on a modern processor?
  - Cost of multi-level page table walk
  - Software-controlled: plus cost of trap handler entry/exit
  - Use additional caching principles: multi-level caching, etc

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 μs</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>

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

Problem with Translation Slowdown

- 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

- Problem: Do we need to wait for the address translation in order to look up the caches (for code and data)?

Virtually vs. Physically Addressed Caches

- It can be 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 with cache lookup using virtual address, access TLB to generate physical address in case of a cache miss
Virtually addressed caches

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 a set of pages in one page table
  - x86 TLB entries
    - 4KB
    - 2MB
    - 1GB
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?
  - Keep using TLB?
  - Flush 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
TLB Consistency Issues

- "Snoopy" cache protocols (hardware)
  - Maintain consistency with DRAM, even when DMA happens
- Consistency between DRAM and TLBs (software)
  - You need to flush related TLBs whenever changing a page table entry in memory
- TLB "shoot-down"
  - On multiprocessors/multicore, when you modify a page table entry, need to flush all related TLB entries on all processors/cores

Summary

- Must consider many issues
  - Global and local replacement strategies
  - Management of backing store
  - Primitive operations
    - Pin/lock pages
    - Zero pages
    - Shared pages
    - Copy-on-write
- Real system designs are complex

<table>
<thead>
<tr>
<th>Processor 1 TLB</th>
<th>Process ID</th>
<th>VirtualPage</th>
<th>PageFrame</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0x0053</td>
<td>0x0003</td>
<td>R/W</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0x40FF</td>
<td>0x0012</td>
<td>R/W</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor 2 TLB</th>
<th>Process ID</th>
<th>VirtualPage</th>
<th>PageFrame</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0x0053</td>
<td>0x0003</td>
<td>R/W</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0x0001</td>
<td>0x0005</td>
<td>Read</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor 3 TLB</th>
<th>Process ID</th>
<th>VirtualPage</th>
<th>PageFrame</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0x40FF</td>
<td>0x0012</td>
<td>R/W</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0x0001</td>
<td>0x0005</td>
<td>Read</td>
</tr>
</tbody>
</table>