Memory 2: Virtual Memory (Con’t), Caching and TLBs

March 8th, 2022
Prof. Anthony Joseph and John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Recall: General Address translation

• Consequently, two views of memory:
  – View from the CPU (what program sees, virtual memory)
  – View from memory (physical memory)
  – Translation box (Memory Management Unit or MMU) converts between the two views

• Translation ⇒ much easier to implement protection!
  – If task A cannot even gain access to task B’s data, no way for A to adversely affect B
  – Extra benefit: every program can be linked/loaded into same region of user address space

Recall: Multi-Segment Model

• Segment map resides in processor
  – Segment number mapped into base/limit pair
  – Base added to offset to generate physical address
  – Error check catches offset out of range
• As many chunks of physical memory as entries
  – Segment addressed by portion of virtual address
  – However, could be included in instruction instead:
    » x86 Example: mov [es:bx],ax.
• What is “V/N” (valid / not valid)?
  – Can mark segments as invalid; requires check as well

What if not all segments fit in memory?

• Extreme form of Context Switch: Swapping
  – To make room for next process, some or all of the previous process is moved to disk
    » Likely need to send out complete segments
  – This greatly increases the cost of context-switching
• What might be a desirable alternative?
  – Some way to keep only active portions of a process in memory at any one time
  – Need finer granularity control over physical memory
Problems with Segmentation

• Must fit variable-sized chunks into physical memory
• May move processes multiple times to fit everything
• Limited options for swapping to disk
• Fragmentation: wasted space
  – External: free gaps between allocated chunks
  – Internal: don’t need all memory within allocated chunks

Recall: General Address Translation

Paging: Physical Memory in Fixed Size Chunks

• Solution to fragmentation from segments?
  – Allocate physical memory in fixed size chunks ("pages")
  – Every chunk of physical memory is equivalent
    » Can use simple vector of bits to handle allocation:
      00110011001101100101
    » Each bit represents page of physical memory
      1 \Rightarrow allocated, 0 \Rightarrow free

• Should pages be as big as our previous segments?
  – No: Can lead to lots of internal fragmentation
    » Typically have small pages (1K-16K)
  – Consequently: need multiple pages/segment

How to Implement Simple Paging?

• Page Table (One per process)
  – Resides in physical memory
  – Contains physical page and permission for each virtual page (e.g. Valid bits, Read, Write, etc)
• Virtual address mapping
  – Offset from Virtual address copied to Physical Address
    » Example: 10 bit offset \Rightarrow 1024-byte pages
  – Virtual page # is all remaining bits
    » Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    » Physical page # copied from table into physical address
  – Check Page Table bounds and permissions
Simple Page Table Example

Example (4 byte pages)

Virtual Memory

Physical Memory

What about Sharing?

Offset

Virtual Address

(Process A):

PageTablePtrA

PageTablePtrB

Virtual Address

(Process B):

Shared Page

This physical page appears in address space of both processes

Where is page sharing used?

- The “kernel region” of every process has the same page table entries
  - The process cannot access it at user level
  - But on U->K switch, kernel code can access it AS WELL AS the region for THIS user
    » What does the kernel need to do to access other user processes?
- Different processes running same binary!
  - Execute-only, but do not need to duplicate code segments
- User-level system libraries (execute only)
- Shared-memory segments between different processes
  - Can actually share objects directly between processes
    » Must map page into same place in address space!
  - This is a limited form of the sharing that threads have within a single process

Recall: Memory Layout for Linux 32-bit (Pre-Meltdown patch)
Some simple security measures

- Address Space Randomization
  - Position-Independent Code ⇒ can place user code anywhere in address space
    » Random start address makes much harder for attacker to cause jump to code that it seeks to take over
  - Stack & Heap can start anywhere, so randomize placement
- Kernel address space isolation
  - Don’t map whole kernel space into each process, switch to kernel page table
  - Meltdown ⇒ map none of kernel into user mode!

![Kernel page-table isolation diagram]

Summary: Paging

What happens if stack grows to 1110 0000?

Allocate new pages where room!
How big do things get?

- 32-bit address space => 2^{32} bytes (4 GB)
  - Note: "b" = bit, and "B" = byte
  - And for memory:
    - "K" (kilo) = 2^{10} = 1024 = 10^3 (But not quite!): Sometimes called "Ki" (Kibi)
    - "M" (mega) = 2^{20} = (1024)^2 = 1,048,576 = 10^6 (But not quite!): Sometimes called "Mi" (Mibi)
    - "G" (giga) = 2^{30} = (1024)^3 = 1,073,741,824 = 10^9 (But not quite!): Sometimes called "Gi" (Gibi)

- Typical page size: 4 KB
  - How many bits of the address is that? (remember 2^{10} = 1024)
    - Ans: 4KB = 4 x 2^{10} = 2^{12} \Rightarrow 12 bits of the address

- So how big is the simple page table for each process?
  - 2^{32}/2^{12} = 2^{20} (that’s about a million entries) x 4 bytes each \Rightarrow 4 MB
  - When 32-bit machines got started (vax 11/780, intel 80386), 16 MB was a LOT of memory

- How big is a simple page table on a 64-bit processor (x86_64)?
  - 2^{64}/2^{12} = 2^{52}(that’s 4.5 \times 10^{16} or 4.5 exa-entries)x8 bytes each = 36 \times 10^{15} bytes or 36 exa-bytes!!!! This is a ridiculous amount of memory!
  - This is really a lot of space – for only the page table!!!

- The address space is sparse, i.e. has holes that are not mapped to physical memory
  - So, most of this space is taken up by page tables mapped to nothing

---

Page Table Discussion

- What needs to be switched on a context switch?
  - Page table pointer and limit
- What provides protection here?
  - Translation (per process) and dual-mode!
  - Can’t let process alter its own page table!
- Analysis
  - Pros
    - Simple memory allocation
    - Easy to share
  - Con: What if address space is sparse?
    - E.g., on UNIX, code starts at 0, stack starts at (2^{31} - 1)
    - With 1K pages, need 2 million page table entries!
  - Con: What if table really big?
    - Not all pages used all the time \Rightarrow would be nice to have working set of page table in memory

- Simple Page table is way too big!
  - Does it all need to be in memory?
  - How about multi-level paging?
  - or combining paging and segmentation

---

How to Structure a Page Table

- Page Table is a map (function) from VPN to PPN
  - Virtual Address \rightarrow Page Table \rightarrow Physical Address
  - Simple page table corresponds to a very large lookup table
    - VPN is index into table, each entry contains PPN
  - What other map structures can you think of?
    - Trees?
    - Hash Tables?

---

Fix for sparse address space: The two-level page table

- Tree of Page Tables
  - "Magic" 10b-10b-12b pattern!
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register (i.e. CR3)
- Valid bits on Page Table Entries
  - Don’t need every 2^{12}-level table
  - Even when exist, 2^{12}-level tables can reside on disk if not in use
Example: x86 classic 32-bit address translation

- Intel terminology: Top-level page-table called a "Page Directory"
  - With "Page Directory Entries"
- CR3 provides physical address of the page directory
  - This is what we have called the "PageTablePtr" in previous slides
  - Change in CR3 changes the whole translation table!

What is in a Page Table Entry (PTE)?

- What is in a Page Table Entry (or PTE)?
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called "Directories"

What is in a Page Table Entry (PTE)?

- What is in a Page Table Entry (or PTE)?
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called "Directories"

Examples of how to use a PTE

- How do we use the PTE?
  - Invalid PTE can imply different things:
    » Region of address space is actually invalid or
    » Page/directory is just somewhere else than memory
  - Validity checked first
    » OS can use other (say) 31 bits for location info
- Usage Example: Demand Paging
  - Keep only active pages in memory
  - Place others on disk and mark their PTEs invalid
- Usage Example: Copy on Write
  - UNIX fork gives copy of parent address space to child
    » Address spaces disconnected after child created
    » How to do this cheaply?
      » Make copy of parent’s page tables (point at same memory)
      » Mark entries in both sets of page tables as read-only
      » Page fault on write creates two copies
- Usage Example: Zero Fill On Demand
  - New data pages must carry no information (say be zeroed)
  - Mark PTEs as invalid; page fault on use gets zeroed page
  - Often, OS creates zeroed pages in background

Sharing with multilevel page tables

- Entire regions of the address space can be efficiently shared
Summary: Two-Level Paging

Virtual memory view

<table>
<thead>
<tr>
<th>Offset</th>
<th>Virtual Address</th>
<th>Page Table (level 1)</th>
<th>Page Table (level 2)</th>
<th>Physical memory view</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>page1 #</td>
<td>stack</td>
<td>stack</td>
<td>1110 0000</td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td>heap</td>
<td>0111 0000</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td>data</td>
<td>0011 0000</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td>code</td>
<td>0001 0000</td>
<td></td>
</tr>
</tbody>
</table>

Physical memory view

<table>
<thead>
<tr>
<th>Offset</th>
<th>Physical Address</th>
<th>Check Permissions</th>
<th>Access Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>page0 #</td>
<td></td>
<td>Access Error</td>
</tr>
<tr>
<td>0100</td>
<td>page1 #</td>
<td></td>
<td>Access Error</td>
</tr>
<tr>
<td>1000</td>
<td>page2 #</td>
<td></td>
<td>Access Error</td>
</tr>
</tbody>
</table>

Multi-level Translation: Segments + Pages

- What about a tree of tables?
  - Lowest level page table ⇒ memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):

  Virtual Address: Base0 Limit0 V
  page #0 V.R
  page #1 V.R
  page #2 V.R
  page #3 V.R
  page #4 N
  page #5 V.R

  Physical Address: Base1 Limit1 V
  page #1 V.R
  page #2 V.R
  page #3 V.R
  page #4 N
  page #5 V.R

- What must be saved/restored on context switch?
  - Contents of top-level segment registers (for this example)
  - Pointer to top-level table (page table)

What about Sharing (Complete Segment)?

Process A:

- Virtual Base Page Offset
- page #0 V.R
- page #1 V.R
- page #2 V.R
- page #3 V.R.W
- page #4 N
- page #5 V.R.W

Shared Segment

Process B:

- Virtual Base Page Offset
- page #0 V.R
- page #1 V.R
- page #2 V.R
- page #3 V.R.W
- page #4 N
- page #5 V.R.W
Multi-level Translation Analysis

- **Pros:**
  - Only need to allocate as many page table entries as we need for application
  - In other words, sparse address spaces are easy
  - Easy memory allocation
  - Easy Sharing
  - Share at segment or page level (need additional reference counting)

- **Cons:**
  - One pointer per page (typically 4K – 16K pages today)
  - Page tables need to be contiguous
    - However, the 10b-10b-12b configuration keeps tables to exactly one page in size
  - Two (or more, if >2 levels) lookups per reference
    - Seems very expensive!

Recall: Dual-Mode Operation

- Can a process modify its own translation tables? **NO!**
  - If it could, could get access to all of physical memory (no protection!)
- To Assist with Protection, **Hardware** provides at least two modes (Dual-Mode Operation):
  - “Kernel” mode (or “supervisor” or “protected”)
  - “User” mode (Normal program mode)
  - Mode set with bit(s) in control register only accessible in Kernel mode
  - Kernel can easily switch to user mode; User program must invoke an exception of some sort to get back to kernel mode (more in moment)
- Note that x86 model actually has more modes:
  - Traditionally, four “rings” representing priority; most OSes use only two:
    - Ring 0 ⇒ Kernel mode, Ring 3 ⇒ User mode
    - Called “Current Privilege Level” or CPL
  - Newer processors have additional mode for hypervisor (“Ring -1”)
- Certain operations restricted to Kernel mode:
  - Modifying page table base (CR3 in x86), and segment descriptor tables
    - Have to transition into Kernel mode before you can change them!
  - Also, all page-table pages must be mapped only in kernel mode

Making it real: X86 Memory model with segmentation (16/32-bit)

- Two registers: GDTR/LDTR hold pointers to global/local descriptor tables in memory
  - Descriptor format (64 bits):
    - G: Granularity of segment [ Limit Size ] (0: 16bit, 1: 4KiB unit)
    - DB: Default operand size (0: 16bit, 1: 32bit)
    - A: Freely available for use by software
    - P: Segment present
    - DPL: Descriptor Privilege Level: Access requires Max(CPL,RPL)≤DPL
    - S: System Segment (0: System, 1: code or data)
How are segments used?

- One set of global segments (GDT) for everyone, different set of local segments (LDT) for every process.
- In legacy applications (16-bit mode):
  - Segments provide protection for different components of user programs
  - Separate segments for chunks of code, data, stacks
    » RPL of Code Segment ⇒ CPL (Current Privilege Level)
  - Limited to 64K segments
- Modern use in 32-bit Mode:
  - Even though there is full segment functionality, segments are set up as "flattened", i.e. every segment is 4GB in size
  - One exception: Use of GS (or FS) as a pointer to “Thread Local Storage” (TLS)
    » A thread can make accesses to TLS like this:
      ```c
      mov eax, gs(0x0)
      ```
- Modern use in 64-bit ("long") mode
  - Most segments (SS, CS, DS, ES) have zero base and no length limits
  - Only FS and GS retain their functionality TLS

From x86_64 architecture specification

- All current x86 processor support a 64 bit operation
- 64-bit words (so ints are 8 bytes) but 48-bit addresses

X86_64: Four-level page table!

48-bit Virtual Address: 
- 9 bits 
- 9 bits 
- 9 bits 
- 9 bits 
- 12 bits

PageTablePtr

4096-byte pages (12 bit offset)
Page tables also 4K bytes (pageable)

Physical Address: 
- 12bit Offset
- (40-50 bits)

Larger page sizes supported as well

- Larger page sizes (2MB, 1GB) make sense since memory is now cheap
  - Great for kernel, large libraries, etc
  - Use limited primarily by internal fragmentation...
**IA64: 64bit addresses: Six-level page table?!?**

64bit Virtual Address:

<table>
<thead>
<tr>
<th>7 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>12 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual</td>
<td>Virtual</td>
<td>Virtual</td>
<td>Virtual</td>
<td>Virtual</td>
<td>Offset</td>
</tr>
</tbody>
</table>

No!

Too slow
Too many almost-empty tables

**Address Translation Comparison**

<table>
<thead>
<tr>
<th></th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Segmentation</td>
<td>Fast context switching (segment map maintained by CPU)</td>
<td>External fragmentation</td>
</tr>
<tr>
<td>Paging (Single-Level)</td>
<td>No external fragmentation Fast and easy allocation</td>
<td>Large table size (~ virtual memory) Internal fragmentation</td>
</tr>
<tr>
<td>Paged Segmentation</td>
<td>Table size ~ # of pages in virtual memory Fast and easy allocation</td>
<td>Multiple memory references per page access</td>
</tr>
<tr>
<td>Multi-Level Paging</td>
<td>Table size ~ # of pages in virtual memory Fast and easy allocation</td>
<td>Hash function more complex No cache locality of page table</td>
</tr>
<tr>
<td>Inverted Page Table</td>
<td>Table size ~ # of pages in physical memory</td>
<td></td>
</tr>
</tbody>
</table>

**Alternative: Inverted Page Table**

- With all previous examples (“Forward Page Tables”)
  - Size of page table is at least as large as amount of virtual memory allocated to processes
  - Physical memory may be much less
    - Much of process space may be out on disk or not in use

- Answer: use a hash table
  - Called an “Inverted Page Table”
  - Size is independent of virtual address space
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces
    - PowerPC, UltraSPARC, IA64

- Cons:
  - Complexity of managing hash chains: Often in hardware!
  - Poor cache locality of page table

**Administrivia**

- Prof Joseph’s office hours: Tuesdays 1-2pm and Thursdays 12-1 (Soda 447A)

- Project 2 design docs are due Friday 3/11

- Midterm 2: Coming up on Thursday 3/17 7-9pm
  - Topics: up until Lecture 16: Scheduling, Deadlock, Address Translation, Virtual Memory, Caching, TLBs, Demand Paging

- Review Session: Wednesday 3/16 (Details TBA)