Address Translation

March 2nd, 2016
Prof. Anthony D. Joseph
http://cs162.eecs.Berkeley.edu

Review: Simple Example: Base and Bounds (CRAY-1)

- Could use base/bounds for dynamic address translation – translation happens at execution:
  - Alter address of every load/store by adding “base”
  - Generate error if address bigger than limit
- This gives program the illusion that it is running on its own dedicated machine, with memory starting at 0
  - Program gets continuous region of memory
  - Addresses within program do not have to be relocated when program placed in different region of DRAM

Review: Issues with Simple B&B Method

- Fragmentation problem over time
  - Not every process is same size → memory becomes fragmented
- Missing support for sparse address space
  - Would like to have multiple chunks/program (Code, Data, Stack)
- Hard to do inter-process sharing
  - Want to share code segments when possible
  - Want to share memory between processes
  - Helped by providing multiple segments per process

More Flexible Segmentation

- Logical View: multiple separate segments
  - Typical: Code, Data, Stack
  - Others: memory sharing, etc
- Each segment is given region of contiguous memory
  - Has a base and limit
  - Can reside anywhere in physical memory
Implementation of 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

Example: Four Segments (16 bit addresses)

Example of Segment Translation

Let’s simulate a bit of this code to see what happens (PC=0x240):
1. Fetch 0x240. Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get “la $a0, varx”
   Move 0x4050 → $a0, Move PC+4 → PC
2. Fetch 0x244. Translated to Physical=0x4244. Get “jal strlen”
   Move 0x0248 → $ra (return address), Move 0x0360 → PC
3. Fetch 0x360. Translated to Physical=0x4360. Get “li $v0, 0”
   Move 0x0000 → $v0, Move PC+4 → PC
4. Fetch 0x364. Translated to Physical=0x4364. Get “lb $t0,($a0)’
   Since $a0 is 0x4050, try to load byte from 0x4050
   Translate 0x4050. Virtual segment #? 1; Offset? 0x50
   Physical address? Base=0x4800, Physical addr = 0x4850,
   Load Byte from 0x4850 → $t0, Move PC+4 → PC
Observations about Segmentation

- Virtual address space has holes
  - Segmentation efficient for sparse address spaces
  - A correct program should never address gaps (except as mentioned in moment)
    » If it does, trap to kernel and dump core
- When it is OK to address outside valid range:
  - This is how the stack and heap are allowed to grow
  - For instance, stack takes fault, system automatically increases size of stack
- Need protection mode in segment table
  - For example, code segment would be read-only
  - Data and stack would be read-write (stores allowed)
  - Shared segment could be read-only or read-write
- What must be saved/restored on context switch?
  - Segment table stored in CPU, not in memory (small)
  - Might store all of processes memory onto disk when switched (called “swapping”)

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

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:
      00110001110001101 … 110010
    » Each bit represents page of physical memory
      1⇒allocated, 0⇒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 Paging?

- **Page Table** (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page
    - Permissions include: Valid bits, Read, Write, etc.

- **Virtual address mapping**
  - Offset from Virtual address copied to Physical Address
    - Example: 10 bit offset ⇒ 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: 0000 0100
- Physical Memory: 0000 0000
- Page Table:
  - Page #0: VR
  - Page #1: VR
  - Page #2: VR,W
  - Page #3: VR,W
  - Page #4: N
  - Page #5: VR,W

**Virtual Address**: 0x10

- 0x00: page #0
- 0x04: page #1
- 0x08: page #2
- 0x0C: page #3
- 0x10: page #4
- 0x14: page #5

### What about Sharing?

- **Process A**:
  - Page Table:
    - Page #0: VR
    - Page #1: VR
    - Page #2: VR,W
    - Page #3: VR,W
    - Page #4: N
    - Page #5: VR,W

- **Process B**:
  - Page Table:
    - Page #0: VR
    - Page #1: N
    - Page #2: N
    - Page #3: N
    - Page #4: VR
    - Page #5: VR,W

- **Shared Page**
  - This physical page appears in address space of both processes

### Administrivia

- **Upcoming deadlines**:
  - Project 1 final code due Fri 3/4, final report due Mon 3/7

- **Midterm next week Wed 3/9 6-7:30 10 Evans and 155 Dwinelle**
  - Midterm review session: Sun 3/6 2-5PM at 2060 VLSB
  - Rooms assignment: aa-eh 10 Evans, ej-oa 155 Dwinelle
  - Lectures (including #12), project, homeworks readings, textbook
  - No books, no calculators, one double-side page handwritten notes
  - No class that day, extra office hours
Memory Layout for Linux 32-bit

http://static.duartes.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png

Summary: Paging

Stack grows to 1110 0000?

What happens if stack grows to 1110 0000?
Summary: Paging

- Virtual memory view
  - Stack
  - Heap
  - Data

- Physical memory view
  - Data
  - Code
  - Stack

Page Table Discussion

- What needs to be switched on a context switch?
  - Page table pointer and limit

- 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 ⇒ would be nice to have working set of page table in memory

- How about combining paging and segmentation?

---

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

- Virtual Address: 10 bits 10 bits 12 bits
- Offset
- Physical Address: Page # Offset

- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Valid bits on Page Table Entries
  - Don't need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

Summary: Two-Level Paging

- Virtual memory view
  - Stack
  - Heap
  - Data
  - Code

- Physical memory view
  - Data
  - Code
  - Stack
Summary: Two-Level Paging

Virtual memory view

- Stack
- Heap
- Data
- Code

Page Tables
(level 2)

- Stack
- Stack
- Stack
- Stack
- Stack
- Stack
- Stack
- Stack

Physical memory view

- Stack
- Stack
- Stack
- Stack
- Stack
- Stack
- Stack
- Stack

Virtual memory view

- Heap
- Data
- Code
- Code
- Data
- Data
- Data
- Data

Physical memory view

- Heap
- Heap
- Heap
- Heap
- Heap
- Heap
- Heap
- Heap

Virtual memory view

- Code
- Code
- Code
- Code
- Code
- Code
- Code
- Code

Physical memory view

- Code
- Code
- Code
- Code
- Code
- Code
- Code
- Code

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:

- Virtual Seg #
- Virtual Page #
- Offset

Physical Page #

- Physical Address

- Check Perm

- Access Error

- Access Error

- Access Error

- Access Error

- Access Error

- Access Error

What about Sharing (Complete Segment)?

Process A

- Virtual Seg #
- Virtual Page #
- Offset

- Page #0 VR
- Page #1 VR
- Page #2 VRW
- Page #3 VRW
- Page #4 N
- Page #5 VRW

Shared Segment

Process B

- Virtual Seg #
- Virtual Page #
- Offset

- Page #0 VR
- Page #1 VR
- Page #2 VRW
- Page #3 VRW
- Page #4 N
- Page #5 VRW

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, previous example keeps tables to exactly one page in size
  - Two (or more, if >2 levels) lookups per reference
    » Seems very expensive
What is in a Page Table Entry?

- **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”

<table>
<thead>
<tr>
<th>Page Frame Number (Physical Page Number)</th>
<th>Free (OS)</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>G/L</th>
<th>P</th>
<th>DPL</th>
<th>S</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td></td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
  - P: Present (same as “valid” bit in other architectures)
  - W: Writeable
  - U: User accessible
  - PWT: Page write transparent: external cache write-through
  - PCD: Page cache disabled (page cannot be cached)
  - A: Accessed: page has been accessed recently
  - D: Dirty (PTE only): page has been modified recently
  - L: L=1⇒4MB page (directory only).
    Bottom 22 bits of virtual address serve as offset

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

**X86 Segment Descriptors (32-bit Protected Mode)**

- Segments are either implicit in the instruction (say for code segments) or actually part of the instruction
  - There are 6 registers: SS, CS, DS, ES, FS, GS

- What is in a segment register?
  - A pointer to the actual segment description:

    | Segment selector [13 bits] |
    |---------------------------|
    | 3                        |

    - G/L selects between GDT and LDT tables (global vs local descriptor tables)
    - Two registers: GDTR and LDTR hold pointers to the global and local descriptor tables in memory
      » Includes length of table (for < 2^16 entries)

- Descriptor format (64 bits):

<table>
<thead>
<tr>
<th>Segment Base Address</th>
<th>Linear Address Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment Description</td>
<td>Page Table</td>
</tr>
<tr>
<td></td>
<td>Page Directory</td>
</tr>
<tr>
<td>Global Descriptor Table (GDT)</td>
<td></td>
</tr>
<tr>
<td>Linear Address</td>
<td>Physical Address</td>
</tr>
<tr>
<td>Offset</td>
<td>Entry</td>
</tr>
<tr>
<td>Segment Selector</td>
<td>Offset</td>
</tr>
<tr>
<td>Offset</td>
<td>Linear Address</td>
</tr>
<tr>
<td>Global Descriptor</td>
<td>Page Table</td>
</tr>
<tr>
<td>Segment Description</td>
<td>Page Directory</td>
</tr>
</tbody>
</table>

  - G: Granularity of segment (0: 16bit, 1: 4KB unit)
  - DB: Default operand size (0: 16bit, 1: 32bit)
  - A: Freely available for use by software
  - P: Segment present
  - DPL: Descriptor Privilege Level
  - S: System Segment (0: System, 1: code or data)
  - Type: Code, Data, Segment
Recall: 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
  - Limited to 64K segments
- Modern use in 32-bit Mode:
  - Segments “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:
      \[
      \text{mov eax, gs(0x0)}
      \]
- 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 (for use in TLS)

**BREAK**

X86_64: Four-level page table!

IA64: 64bit addresses: Six-level page table?!

No!

Too slow
Too many almost-empty tables
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
- Cons: Complexity of managing hash changes
  - Often in hardware!

IPT address translation

- Need an associative map from VM page to IPT address:
  - Use a hash map

IA64: Inverse Page Table (IPT)

Idea: index page table by physical pages instead of VM

Summary: Inverted Table

Virtual memory view

Inverted Table

Physical memory view

Hash(procID & virt. page #) = phys. page #
## 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 mapping maintained by CPU</td>
<td>External fragmentation</td>
</tr>
<tr>
<td>Paging (single-level page)</td>
<td>No external fragmentation, fast easy allocation</td>
<td>Large table size ~ virtual memory</td>
</tr>
<tr>
<td>Paged segmentation</td>
<td>Table size ~ # of pages in virtual memory, fast easy allocation</td>
<td>Multiple memory references per page access</td>
</tr>
<tr>
<td>Two-level pages</td>
<td>Table size ~ # of pages in physical memory</td>
<td>Hash function more complex</td>
</tr>
</tbody>
</table>

## Summary

### Segment Mapping
- Segment registers within processor
- Segment ID associated with each access
  - Often comes from portion of virtual address
  - Can come from bits in instruction instead (x86)
- Each segment contains base and limit information
  - Offset (rest of address) adjusted by adding base

### Page Tables
- Memory divided into fixed-sized chunks of memory
- Virtual page number from virtual address mapped through page table to physical page number
- Offset of virtual address same as physical address
- Large page tables can be placed into virtual memory

### Multi-Level Tables
- Virtual address mapped to series of tables
- Permit sparse population of address space

### Inverted page table
- Size of page table related to physical memory size