Address Translation - Continued

David E. Culler
http://cs162.eecs.berkeley.edu/
Lecture #15
Oct 1, 2014
Summary: Address Segmentation

Virtual memory view

<table>
<thead>
<tr>
<th>Seg #</th>
<th>base</th>
<th>limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1011 0000</td>
<td>1 0000</td>
</tr>
<tr>
<td>10</td>
<td>0111 0000</td>
<td>1 1000</td>
</tr>
<tr>
<td>01</td>
<td>0101 0000</td>
<td>10 0000</td>
</tr>
<tr>
<td>00</td>
<td>0001 0000</td>
<td>10 0000</td>
</tr>
</tbody>
</table>

Physical memory view

Heap

Virtual memory view: stack at 1111 0000 (0xFO), heap at 1100 0000 (0xC0), data at 1000 0000 (0x80), code at 0100 0000 (0x40), and stack at 0000 0000 (0x00).

Physical memory view: stack at 1110 0000 (0xE0), heap at 1110 0000 (0xE0), data at 0111 0000 (0x70), and code at 0101 0000 (0x50).
Summary: Address Segmentation

Virtual memory view

Stack

Heap

Code

Data

Physical memory view

Seg # | base | limit
--- | --- | ---
11 | 1011 0000 | 1 0000
10 | 0111 0000 | 1 1000
01 | 0101 0000 | 10 0000
00 | 0001 0000 | 10 0000

What happens if stack grows to 1110 0000?
Recap: Address Segmentation

Virtual memory view

1111 1111

stack

1110 0000

heap

1100 0000

data

1000 0000

code

Physical memory view

0000 0000

stack

110 0000

data

01 0000

heap

00 0000

code

Seg #  base  limit
11  1011 0000  1 0000
10  0111 0000  1 1000
01  0101 0000  10 0000
00  0001 0000  10 0000

No room to grow!! Buffer overflow error or resize segment and move segments around to make room
How do we run more programs than fit in memory?
Q: What if not all processes fit in memory?
A: Swapping: Extreme form of Context Switch
   – In order to make room for next process, some or all of the previous process is moved to disk
   – This greatly increases the cost of context-switching

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
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
What about Sharing?

Virtual Address (Process A):

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

Virtual Address (Process B):

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

PageTablePtrA

PageTablePtrB

Shared Page

This physical page appears in address space of both processes.
Simple Page Table Example

Example (4 byte pages)

Virtual Memory

<table>
<thead>
<tr>
<th>0x00</th>
<th>0x04</th>
<th>0x06?</th>
<th>0x08</th>
<th>0x09?</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 0000</td>
<td>0000 0100</td>
<td>0000 1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Page Table

<table>
<thead>
<tr>
<th>0x00</th>
<th>0x04</th>
<th>0x08</th>
<th>0x0C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>0000 0100</td>
<td>0000 1000</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Physical Memory

<table>
<thead>
<tr>
<th>0x00</th>
<th>0x04</th>
<th>0x08</th>
<th>0x0C</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0x05!

0x0E!
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 $\Rightarrow$ would be nice to have working set of page table in memory

- How about combining paging and segmentation?
E.g., Linux 32-bit

Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault.

Stack (grows down)

Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

Heap

BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;

Data segment
Static variables initialized by the programmer. Example: static char *gonzo = “God’s own prototype”;

Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)

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

Virtual memory view

Page Table

Physical memory view

Stack

Heap

Data

Code

Page Table

Virtual memory view

Physical memory view
Summary: Paging

Virtual memory view

Physical memory view

Page Table

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

Virtual memory view:
- Stack: 1111 1111
- Heap: 1110 0000
- Data: 1100 0000
- Code: 0100 0000

Physical memory view:
- Stack: 1110 0000
- Data: 0101 0000
- Code: 0001 0000
- Null pages: 0000 0000

Page Table:
- Allocate new pages where room!

Page dimensions: 720.0x540.0
Multi-level Translation

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

<table>
<thead>
<tr>
<th>Virtual Seq #</th>
<th>Virtual Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base0</td>
<td>Limit0</td>
<td>V</td>
</tr>
<tr>
<td>Base1</td>
<td>Limit1</td>
<td>V</td>
</tr>
<tr>
<td>Base2</td>
<td>Limit2</td>
<td>V</td>
</tr>
<tr>
<td>Base3</td>
<td>Limit3</td>
<td>N</td>
</tr>
<tr>
<td>Base4</td>
<td>Limit4</td>
<td>V</td>
</tr>
<tr>
<td>Base5</td>
<td>Limit5</td>
<td>N</td>
</tr>
<tr>
<td>Base6</td>
<td>Limit6</td>
<td>N</td>
</tr>
<tr>
<td>Base7</td>
<td>Limit7</td>
<td>V</td>
</tr>
</tbody>
</table>

- 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)?

<table>
<thead>
<tr>
<th>Process A</th>
<th>Virtual Seq #</th>
<th>Virtual Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual Page #</th>
<th>Offset</th>
<th>Base0</th>
<th>Limit0</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base1</td>
<td>Limit1</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base2</td>
<td>Limit2</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base3</td>
<td>Limit3</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base4</td>
<td>Limit4</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base5</td>
<td>Limit5</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base6</td>
<td>Limit6</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base7</td>
<td>Limit7</td>
<td>V</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process B</th>
<th>Virtual Seq #</th>
<th>Virtual Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual Page #</th>
<th>Offset</th>
<th>Base0</th>
<th>Limit0</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base1</td>
<td>Limit1</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base2</td>
<td>Limit2</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base3</td>
<td>Limit3</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base4</td>
<td>Limit4</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base5</td>
<td>Limit5</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base6</td>
<td>Limit6</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base7</td>
<td>Limit7</td>
<td>V</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shared Segment</th>
<th>Virtual Seq #</th>
<th>Virtual Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual Page #</th>
<th>Offset</th>
<th>Base0</th>
<th>Limit0</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base1</td>
<td>Limit1</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base2</td>
<td>Limit2</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base3</td>
<td>Limit3</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base4</td>
<td>Limit4</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base5</td>
<td>Limit5</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base6</td>
<td>Limit6</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base7</td>
<td>Limit7</td>
<td>V</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Another common example: two-level page

Virtual Address:

- 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 2\textsuperscript{nd}-level table
  - Even when exist, 2\textsuperscript{nd}-level tables can reside on disk if not in use
Multi-level Translation Analysis

• Pros:
  – Only need to allocate as many page table entries as we need for application – size is proportional to usage
    » 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!
Summary: Two-Level Paging

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

Page Tables (level 1):
- Page Table
  - Stack
  - Heap
  - Data
  - Code

Page Tables (level 2):
- Stack
- Heap
- Data
- Code

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

Virtual memory view

- stack
- heap
- data
- code

Page Tables

(level 1)

- Page Table

(level 2)

Physical memory view

- stack
- heap
- data
- code

In best case, total size of page tables \( \approx \) number of pages used by program.

Requires two additional memory access!
Inverted Page Table

- With all previous examples (“Forward Page Tables”)
  - Size of page tables is at least as large as amount of virtual memory allocated to ALL processes
  - Physical memory may be much, 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 phy mem (1 entry per phy page)
  - Very attractive option for 64-bit address spaces (IA64, PowerPC, UltraSPARC)

- Cons: Complexity of managing hash chains in hardware

```
Virtual Page # | Offset
---------------|--------

Hash Table

Process ID

Physical Page # | Offset
```

10/3/14
Summary: Inverted Table

Virtual memory view

1111 1111
stack

1110 0000

1100 0000
heap

1000 0000
data

0100 0000
code

Inverted Table
Hash(procID & virt. page #) = phys. page #

physical memory view

stack

1110 0000

1011 0000
heap

0111 0000
data

0101 0000
code

Total size of page table ≈ number of pages used by program in physical memory.

Hash more complex!
### Address Translation Comparison

<table>
<thead>
<tr>
<th>Method</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>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></td>
<td></td>
<td>Internal fragmentation</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></td>
<td></td>
</tr>
<tr>
<td>Inverted Table</td>
<td>Table size ~ # of pages in physical memory</td>
<td>Hash function more complex</td>
</tr>
</tbody>
</table>
What happens when …

Processing a virtual address:
- Instruction
- Virtual address
- MMU
- MMU accesses page table
- Page Table
  - Frame#
  - Offset
- Physical address

Page fault:
- Exception
- Retry

Operating System:
- Page Fault Handler
- Load page from disk
- Update PT entry

Scheduler:
- Process
- Retry
How has OS design choices been influenced by technological change?
Figure 7  2011 ITRS Product Technology Trends: Memory Product Functions/Chip and Industry Average “Moore’s Law” and Chip Size Trends [unchanged for the 2012 Update]

Source: 2011 ITRS - Executive Summary Fig 7
Summary

• Memory is a resource that must be multiplexed
  – Controlled Overlap: only shared when appropriate
  – Translation: Change virtual addresses into physical addresses
  – Protection: Prevent unauthorized sharing of resources

• Simple Protection through segmentation
  – Base + Limit registers restrict memory accessible to user
  – Can be used to translate as well

• Page Tables
  – Memory divided into fixed-sized chunks of memory
  – Offset of virtual address same as physical address

• 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