CS162
Operating Systems and
Systems Programming
Lecture 9

Address Translation

February 24, 2014
Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Virtualizing Resources

• Physical Reality: Processes/Threads share the same hardware
  – Need to multiplex CPU (CPU Scheduling)
  – Need to multiplex use of Memory (Today)
• Why worry about memory multiplexing?
  – The complete working state of a process and/or kernel is defined by its data in memory (and registers)
  – Consequently, cannot just let different processes use the same memory
  – Probably don’t want different processes to even have access to each other’s memory (protection)

Goals for Today

• Address Translation Schemes
  – Segmentation
  – Paging
  – Multi-level translation
  – Paged page tables
  – Inverted page tables

Important Aspects of Memory Multiplexing

• Controlled overlap:
  – Processes should not collide in physical memory
  – Conversely, would like the ability to share memory when desired (for communication)

• Protection:
  – Prevent access to private memory of other processes
    » Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc.)
    » Kernel data protected from User programs

• Translation:
  – Ability to translate accesses from one address space (virtual) to a different one (physical)
  – When translation exists, process uses virtual addresses, physical memory uses physical addresses

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Slides courtesy of Anthony D. Joseph, John Kubiatowicz, AJ Shankar, George Necula, Alex Aiken, Eric Brewer, Ras Bodik, Ion Stoica, Doug Tygar, and David Wagner.
9.5 Binding of Instructions and Data to Memory

Process view of memory

Physical addresses

data1: dw 32
start: lw r1,0(data1)
jal checkit
loop: addi r1, r1, -1
bnz r1, loop
checkit: ...

Assume 4byte words
0x300 = 4 * 0x0C0
0x0C0 = 0000 1100 0000
0x300 = 0011 0000 0000

9.6 Binding of Instructions and Data to Memory

Process view of memory

Physical addresses

data1: dw 32
start: lw r1,0(data1)
jal checkit
loop: addi r1, r1, -1
bnz r1, loop
checkit: ...

Assume 4byte words
0x300 = 4 * 0x0C0
0x0C0 = 0000 1100 0000
0x300 = 0011 0000 0000

9.7 Binding of Instructions and Data to Memory

Physical Memory

Process view of memory

Physical addresses

data1: dw 32
start: lw r1,0(data1)
jal checkit
loop: addi r1, r1, -1
bnz r1, loop
checkit: ...

Physical Memory

App X

0x0000
0x0300
0x0900
0x0904
0x0908
0x090C
0x0A00
0x0FFF

9.8 Binding of Instructions and Data to Memory

Processor view of memory

Memory

Process view of memory

Physical addresses

App X

0x0000
0x0300
0x0900
0x0904
0x0908
0x090C
0x0A00
0x0FFF

• One of many possible translations!
• Where does translation take place?
  Compile time, Link/Load time, or Execution time?
Multi-step Processing of a Program for Execution

- Preparation of a program for execution involves components at:
  - Compile time (i.e., "gcc")
  - Link/Load time (UNIX "ld" does link)
  - Execution time (e.g., dynamic libs)

- Addresses can be bound to final values anywhere in this path
  - Depends on hardware support
  - Also depends on operating system

- Dynamic Libraries
  - Linking postponed until execution
  - Small piece of code, stub, used to locate appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes routine

Example of General Address Translation

Translation Map 1

Translation Map 2

Physical Address Space

Uniprogramming (MS-DOS)

- Uniprogramming (no Translation or Protection)
  - Application always runs at same place in physical memory since only one application at a time
  - Application can access any physical address

- With translation, every program can be linked/loaded into same region of user address space

Two Views of Memory

- Address Space:
  - All the addresses and state a process can touch
  - Each process and kernel has different address space

- Consequently, two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)
  - Translation box (MMU) converts between the two views

- Translation helps to implement protection
  - If task A cannot even gain access to task B's data, no way for A to adversely affect B

- Application given illusion of dedicated machine by giving it reality of a dedicated machine
### Multiprogramming (First Version)

- Multiprogramming without Translation or Protection
  - Must somehow prevent address overlap between threads
    - Trick: Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
      - Everything adjusted to memory location of program
      - Translation done by a linker-loader
      - Was pretty common in early days
    - With this solution, no protection: bugs in any program can cause other programs to crash or even the OS

### Multiprogramming (Version with Protection)

- Can we protect programs from each other without translation?
  - Yes: use two special registers `BaseAddr` and `LimitAddr` to prevent user from straying outside designated area
    - If user tries to access an illegal address, cause an error
  - During switch, kernel loads new base/limit from PCB (Process Control Block)
    - User not allowed to change base/limit registers

### Simple Base and Bounds (CRAY-1)

- Could use base/limit for dynamic address translation (often called "segmentation") – 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

### Issues with Simple Segmentation Method

- Fragmentation problem
  - Not every process is the same size
  - Over time, memory space becomes fragmented
- 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

Example: Four Segments (16 bit addresses)

<table>
<thead>
<tr>
<th>Seg ID</th>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (code)</td>
<td>0x4000</td>
<td>0x0800</td>
</tr>
<tr>
<td>1 (data)</td>
<td>0x4800</td>
<td>0x1400</td>
</tr>
<tr>
<td>2 (shared)</td>
<td>0xF000</td>
<td>0x1000</td>
</tr>
<tr>
<td>3 (stack)</td>
<td>0x0000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>

Physical Address Space

Schematic View of Swapping

- 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

Administrivia

- Project #1 code due Tuesday Feb 25 by 11:59pm
- Design doc (submit proj1-final-design) and group evals (Google Docs form) due Wed 2/26 at 11:59PM

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)
  - Consequentially: need multiple pages/segment

5min Break
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
  - 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
  - Resides in physical memory
  - Check Page Table bounds and permissions

Simple Page Table Example

Virtual Address (Process A):

<table>
<thead>
<tr>
<th>Page Table</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>page #0</td>
<td>V,R,W</td>
</tr>
<tr>
<td>page #1</td>
<td>V,R</td>
</tr>
<tr>
<td>page #2</td>
<td>V,R,W</td>
</tr>
<tr>
<td>page #3</td>
<td>V,R,W</td>
</tr>
<tr>
<td>page #4</td>
<td>N</td>
</tr>
<tr>
<td>page #5</td>
<td>V,R,W</td>
</tr>
</tbody>
</table>

Virtual Address (Process B):

<table>
<thead>
<tr>
<th>Page Table</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>page #0</td>
<td>V,R</td>
</tr>
<tr>
<td>page #1</td>
<td>N</td>
</tr>
<tr>
<td>page #2</td>
<td>V,R,W</td>
</tr>
<tr>
<td>page #3</td>
<td>N</td>
</tr>
<tr>
<td>page #4</td>
<td>V,R</td>
</tr>
</tbody>
</table>

What about Sharing?

- Virtual Address (Process A):
- Virtual Address (Process B):

- 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

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?
Multi-level Translation

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

Virtual Address:

- What must be saved/restore 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 Seq # Virtual Page # Offset

Page #0 V.R.
Page #1 V.R.
Page #2 V.R.
Page #3 V.R.
Page #4 N
Page #5 V,R,W

Shared Segment

Base0 Limit0 V
Base1 Limit1 V
Base2 Limit2 V
Base3 Limit3 N
Base4 Limit4 V
Base5 Limit5 N
Base6 Limit6 N
Base7 Limit7 V

Process B

Virtual Seq # Virtual Page # Offset

Page #0 V.R.
Page #1 V.R.
Page #2 V.R.
Page #3 V.R.
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 – 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!
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

Summary: Address Segmentation

Recap: Address Segmentation

What happens if stack grows to 1110 0000?

No room to grow!! Buffer overflow error or resize segment and move segments around to make room.
**Summary: Paging**

Virtual memory view |
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 1111</td>
</tr>
<tr>
<td>1111 0000</td>
</tr>
<tr>
<td>1110 0000</td>
</tr>
<tr>
<td>1100 0000</td>
</tr>
<tr>
<td>1000 0000</td>
</tr>
<tr>
<td>0100 0000</td>
</tr>
<tr>
<td>0000 0000</td>
</tr>
</tbody>
</table>

Page Table

| 1111 1111 |
| 1110 0000 |
| 1100 0000 |
| 1000 0000 |
| 0100 0000 |
| 0000 0000 |

Physical memory view

| 1111 1111 |
| 1110 0000 |
| 1100 0000 |
| 1000 0000 |
| 0100 0000 |
| 0000 0000 |

- **Stack**: 1111 0000
- **Heap**: 1110 0000
- **Data**: 1100 0000
- **Code**: 1000 0000

**Challenge:**

- **Stack**: 0000 0000
- **Heap**: 0100 0000
- **Data**: 1100 0000
- **Code**: 1000 0000

**What happens if stack grows to 1110 0000?**

**Summary: Two-Level Paging**

Virtual memory view

| 1111 1111 |
| 1111 0000 |
| 1100 0000 |
| 1000 0000 |
| 0100 0000 |
| 0000 0000 |

Page Table (level 2)

| 1111 1111 |
| 1111 0000 |
| 1100 0000 |
| 1000 0000 |
| 0100 0000 |
| 0000 0000 |

Physical memory view

| 1111 1111 |
| 1111 0000 |
| 1100 0000 |
| 1000 0000 |
| 0100 0000 |
| 0000 0000 |

- **Stack**: 1111 0000
- **Heap**: 1100 0000
- **Data**: 1000 0000
- **Code**: 0100 0000

**Page Table (level 1)**

| 1111 1111 |
| 1111 0000 |
| 1100 0000 |
| 1000 0000 |
| 0100 0000 |
| 0000 0000 |

- **Stack**: 1111 0000
- **Heap**: 1100 0000
- **Data**: 1000 0000
- **Code**: 0100 0000

**Allocate new pages where room!**
In best case, total size of page tables \( \approx \) number of pages.

Requires two additional memory access!

Virtual memory view

- **stack**
- **heap**
- **data**
- **code**

Physical memory view

- **stack**
- **heap**
- **data**
- **code**

**Page Table**

- **(level 1)**
- **(level 2)**

**Address Translation Comparison**

<table>
<thead>
<tr>
<th></th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segmentation</td>
<td>Fast context switching: Segment mapping</td>
<td>External fragmentation</td>
</tr>
<tr>
<td></td>
<td>maintained by CPU</td>
<td></td>
</tr>
<tr>
<td>Paging (single-level page)</td>
<td>No external fragmentation, fast easy allocation</td>
<td>Large table size ( \sim ) virtual memory</td>
</tr>
<tr>
<td>Paged segmentation</td>
<td>Table size ( \sim ) # 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 ( \sim ) # of pages in physical memory</td>
<td>Hash function more complex</td>
</tr>
</tbody>
</table>

**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