Address Translation

March 5, 2018
http://cs162.eecs.Berkeley.edu

Administrivia

- Deadline for midterm regrade requests: this Friday, 3/9
- Project 1 final reports: tonight!
- Project 2, Homework 3: out now

Virtualizing Resources

- Physical Reality: Different Processes/Threads share the same hardware
  - Need to multiplex CPU (done)
  - Need to multiplex use of Memory (Today)
  - Need to multiplex disk and devices (later in term)
- Why worry about memory sharing?
  - The complete working state of a process and/or kernel is defined by its data in memory (and registers)
  - Consequently, two different processes cannot use the same memory
    » Physics: two different data cannot occupy same locations in memory
  - May not want different threads to have access to each other’s memory
Next Objective

- Dive deeper into the concepts and mechanisms of memory sharing and address translation
- Enabler of many key aspects of operating systems
  - Protection
  - Multi-programming
  - Isolation
  - Memory resource management
  - I/O efficiency
  - Sharing
  - Inter-process communication
  - Demand paging
- Today: Linking & Loading, Segmentation, Paging

Recall: Single and Multithreaded Processes

- Threads encapsulate concurrency
  - “Active” component of a process
- Address spaces encapsulate protection
  - Keeps buggy program from trashing the system
  - “Passive” component of a process

Important Aspects of Memory Multiplexing (1/2)

- Protection: prevent access to private memory of other processes
  - Kernel data protected from User programs
  - Programs protected from themselves
  - May want to give special behavior to different memory regions (Read Only, Invisible to user programs, etc)

- Controlled overlap: sometimes we want to share memory across processes.
  - E.g., communication across processes, share code
  - Need to control such overlap

Important Aspects of Memory Multiplexing (2/2)

- Translation:
  - Ability to translate accesses from one address space (virtual) to a different one (physical)
  - When translation exists, processor uses virtual addresses, physical memory uses physical addresses
  - Side effects:
    » Can be used to give uniform view of memory to programs
    » Can be used to provide protection (e.g., avoid overlap)
    » Can be used to control overlap
Recall: Loading

Binding of Instructions and Data to Memory

Process view of memory

Physical addresses

Second copy of program from previous example

Need address translation!
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

Recall: Uniprogramming

- 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
- Application given illusion of dedicated machine by giving it reality of a dedicated machine

Multiprogramming (primitive stage)

- Multiprogramming without Translation or Protection
  - Must somehow prevent address overlap between threads
  - 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 (relocation)
    - Common in early days (… till Windows 3.x, 95?)
- 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 \texttt{BaseAddr} and \texttt{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

Recall: General Address translation

• Recall: 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 makes it 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
• With translation, every program can be linked/loaded into same region of virtual address space

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

Issues with Simple B&B Method

• Fragmentation problem over time
  - Not every process is same size \(\Rightarrow\) 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

Intel x86 Special Registers

- Typical Segment Register
  Current Priority is RPL
  Of Code Segment (CS)
### 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>

**Virtual Address Format**

<table>
<thead>
<tr>
<th>Seg Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 0</td>
</tr>
</tbody>
</table>

**Physical Address Space**

<table>
<thead>
<tr>
<th>Seg Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 0x0400 0x0480 0x0800 0x0C00</td>
</tr>
</tbody>
</table>

**Example of Segment Translation (16b address)**

```assembly
0x240 main:  ja $a0, varx
0x244 jal strlen
... 0x360 strlen: li ... $v0, 0 ;count
0x364 loop: lb $t0, ($a0)
0x368 beq $r0,$t0, done
... 0x4050 varx  dw 0x314159
```

Let's simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x240 (0000 0010 0100 0000). 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

### Example of Segment Translation (16b address)

<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>

**Virtual Address Format**

<table>
<thead>
<tr>
<th>Seg Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 0</td>
</tr>
</tbody>
</table>

**Physical Address Space**

<table>
<thead>
<tr>
<th>Seg Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 0x0400 0x0480 0x0800 0x0C00</td>
</tr>
</tbody>
</table>

**Example of Segment Translation (16b address)**

```assembly
0x240 main:  ja $a0, varx
0x244 jal strlen
... 0x360 strlen: li ... $v0, 0 ;count
0x364 loop: lb $t0, ($a0)
0x368 beq $r0,$t0, done
... 0x4050 varx  dw 0x314159
```

Let's simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x240 (0000 0010 0100 0000). 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
Example of Segment Translation (16b address)

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 (0100 0000 0101 0000). Virtual segment #? 1; Offset? 0x50
   Physical address? Base=0x4800, Physical addr=0x4850,
   Load Byte from 0x4850 → $t0, Move PC+4 → PC

Seg ID # Base Limit
0 (code) 0x4000 0x0800
1 (data) 0x4800 0x1400
2 (shared) 0xF000 0x1000
3 (stack) 0x0000 0x3000

Seg ID # Base Limit
0 (code) 0x4000 0x0800
1 (data) 0x4800 0x1400
2 (shared) 0xF000 0x1000
3 (stack) 0x0000 0x3000
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

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:
      \[00110001110001101 \ldots 110010\]
    » Each bit represents page of physical memory
      \[1 \Rightarrow \text{allocated}, \ 0 \Rightarrow \text{free}\]
- 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 \( \Rightarrow \) 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

What about Sharing?

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

Summary: Paging
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 multi-level paging or combining paging and segmentation?
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):

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

Process A

```
<table>
<thead>
<tr>
<th>Virtual Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>page #0</td>
<td>VR</td>
</tr>
<tr>
<td>page #1</td>
<td>VR</td>
</tr>
<tr>
<td>page #2</td>
<td>VR,W</td>
</tr>
<tr>
<td>page #3</td>
<td>VR,W</td>
</tr>
<tr>
<td>page #4</td>
<td>N</td>
</tr>
<tr>
<td>page #5</td>
<td>VR,W</td>
</tr>
</tbody>
</table>
```

Shared Segment

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

Process B

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

### 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