Memory Virtualization
3) Dynamic Relocation

Goal: Protect processes from one another

Requires hardware support
- Memory Management Unit (MMU)

MMU dynamically changes process address at every memory reference
- Process generates logical or virtual addresses (in their address space)
- Memory hardware uses physical or real addresses
Hardware support for Dynamic Relocation

Two operating modes

• Privileged (protected, kernel) mode: OS runs
  • When enter OS (trap, system calls, interrupts, exceptions)
  • Allows certain instructions to be executed
    • Can manipulate contents of MMU
  • Allows OS to access all of physical memory
• User mode: User processes run
  • Perform translation of logical address to physical address

A minimal MMU contains **base register** for translation
• base: start location for address space
Implementation of Dynamic Relocation: BASE REG

- Translation on every memory access of user process
  - MMU adds base register to logical address to form physical address

\[ \text{logical address} \rightarrow \text{mode} \rightarrow \text{physical address} \]

- **MMU**
  - 32 bits
  - 1 bit
  - **base**
  - **mode**

\[ \text{mode} \begin{cases} \text{no} & \text{if user?} \\ \text{yes} & \text{if user?} \end{cases} \]

\[ \text{physical address} = \text{base} \]
Dynamic Relocation with Base Register

Idea: translate virtual addresses to physical by adding a fixed offset each time.

Store offset in base register

Each process has different value in base register
VISUAL Example of DYNAMIC RELOCATION: BASE REGISTER
P1 is running
P2 is running

base register
P1: load 100, R1

(Decimal notation)

Virtual Physical
Virtual  | Physical
P1: load 100, R1 | load 1124, R1  | (1024 + 100)
Virtual
P1: load 100, R1
P2: load 100, R1

Physical
load 1124, R1
Virtual  |  Physical
---|---
P1: load 100, R1 | load 1124, R1
P2: load 100, R1 | load 4196, R1  (4096 + 100)
Virtual
P1: load 100, R1
P2: load 1000, R1

Physical
load 1124, R1
load 4196, R1
Virtual
P1: load 100, R1
P2: load 1000, R1

Physical
load 1124, R1
load 4196, R1
load 5196, R1
Virtual
P1: load 100, R1
P2: load 100, R1
P2: load 1000, R1
P1: load 1000, R1

Physical
load 1124, R1
load 4196, R1
load 5196, R1
Virtual
P1: load 100, R1
P2: load 100, R1
P2: load 1000, R1
P1: load 1000, R1

Physical
load 1124, R1
load 4196, R1
load 5196, R1
load 2024, R1
Who Controls the Base Register?

What entity should do translation of addresses with base register? (1) process, (2) OS, or (3) HW?

What entity should modify the base register? (1) process, (2) OS, or (3) HW?
Can P2 hurt P1?
Can P1 hurt P2?

Does the base register mechanism protect processes from each other?
Can P2 hurt P1?
Can P1 hurt P2?

Does the base register mechanism protect processes from each other?
4) Dynamic with Base+Bounds

- Idea: limit the address space with a bounds register
- Base register: smallest physical addr (or starting location)
- **Bounds register**: size of this process’s virtual address space
  - Sometimes defined as largest physical address (base + size)
- OS kills process if process loads/stores beyond bounds
Implementation of BASE+BOUNDS

Translation on every memory access of user process

- MMU compares logical address to bounds register
- if logical address is greater, then generate error
- MMU adds base register to logical address to form physical address

![Diagram of BASE+BOUNDS implementation]

- **Logical Address**: The address as provided by the process.
- **Mode**: User or supervisor.
- **Base**: 32-bit register.
- **Bounds**: 32-bit register.
- **Mode**: 1-bit register indicating user or supervisor mode.
- **Physical Address**: The final address after translation.

Error flow:
- If logical address is greater than bounds in mode user, generate error.
- Else, compare logical address against base.
- If logical address is less than base, generate error.
- Else, form physical address as logical address + base.
P1 is running

- base register
- bounds register
P2 is running

- base register
- bounds register
Can P1 hurt P2?
Can P1 hurt P2?

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: load 100, R1</td>
<td>load 1124, R1</td>
</tr>
<tr>
<td>P2: load 100, R1</td>
<td>load 4196, R1</td>
</tr>
<tr>
<td>P2: load 1000, R1</td>
<td>load 5196, R1</td>
</tr>
<tr>
<td>P1: load 1000, R1</td>
<td>load 2024, R1</td>
</tr>
<tr>
<td>P1: store 3072, R1</td>
<td>interrupt OS! 3072 &gt; 1024</td>
</tr>
</tbody>
</table>
Can P1 hurt P2?

- **Virtual**
  - P1: load 100, R1
  - P2: load 100, R1
  - P2: load 1000, R1
  - P1: load 1000, R1
  - P1: store 3072, R1

- **Physical**
  - load 1124, R1
  - load 4196, R1
  - load 5196, R1
  - load 2024, R1
  - **interrupt OS!**
Managing Processes: Base & Bounds

Context-switch
- Add base and bounds registers to Process Control Block
- Steps
  - Change to privileged mode
  - Save base and bounds registers of old process
  - Load base and bounds registers of new process
  - Change to user mode and jump to new process

Protection requirements
- User process cannot change base and bounds registers
- User process cannot change to privileged mode
Base and Bounds Advantages

- Provides protection (both read and write) across address spaces
- Supports dynamic relocation
  - Can place process initially at locations different from assumed in the program code
  - Also, move address spaces later if needed
- Simple, inexpensive implementation
  - Few registers, little logic in MMU
- Fast
  - Add and compare in parallel
Base and Bounds DISADVANTAGES

- Each process must be allocated contiguously in physical memory
- Must allocate memory that may not be used by process
- No partial sharing: Cannot share limited parts of address space
5) Segmentation

Divide address space into logical segments
- Each segment corresponds to logical entity in address space
  - code, stack, heap

Each segment can independently:
- be placed separately in physical memory
- grow and shrink
- be protected (separate read/write/execute bits)
Segmented Addressing

Process now specifies segment and offset within segment

How does process designate a particular segment?

- Use part of logical (virtual) address
  - High-order bits of logical address select segment
  - Low-order bits of logical address select offset within segment

What if small address space, not enough bits?

- Implicitly by type of memory reference
- Special registers
Segmentation Implementation

MMU contains Segment Table (per process)
- Each segment has own base and bounds, protection bits
- Example: 14-bit logical address, 4 segments; how many bits for segment? How many bits for offset?

<table>
<thead>
<tr>
<th>Segment</th>
<th>Base</th>
<th>Bounds</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x2000</td>
<td>0x6ff</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0x0000</td>
<td>0x4ff</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0x3000</td>
<td>0xffff</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0x0000</td>
<td>0x000</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

remember:
1 hex digit -> 4 bits
Segmentation Implementation

MMU contains Segment Table (per process)
• Each segment has own base and bounds, protection bits
• Example: 14-bit logical address, 4 segments; how many bits for segment? How many bits for offset?

<table>
<thead>
<tr>
<th>Segment</th>
<th>Base</th>
<th>Bounds</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x2000</td>
<td>0x6ff</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0x0000</td>
<td>0x4ff</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0x3000</td>
<td>0xffff</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0x0000</td>
<td>0x000</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Remember: 1 hex digit -> 4 bits

Translate logical addresses (in hex) to physical addresses
0x0240:
0x1108:
0x265c:
0x3002:
Assume 14-bit virtual addresses, high 2 bits indicate segment

Where does segment table live?

All registers, MMU

<table>
<thead>
<tr>
<th>Seg Base</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4000</td>
<td>0xffff</td>
</tr>
<tr>
<td>0x5800</td>
<td>0xffff</td>
</tr>
<tr>
<td>0x6800</td>
<td>0x7ff</td>
</tr>
</tbody>
</table>
Segment numbers:
0: code+data
1: heap
2: stack

Virtual (hex)
load 0x2010, R1

Physical
Segment numbers:
0: code+data
1: heap
2: stack

Virtual (hex)
load 0x2010, R1

Physical
0x1600 + 0x010 = 0x1610
Segment numbers:
  0: code+data
  1: heap
  2: stack

Virtual (hex)
load 0x2010, R1
load 0x1010, R1

Physical
0x1600 + 0x010 = 0x1610
heap (seg1)

Virtual (hex)
load 0x2010, R1
load 0x1010, R1

0x400 + 0x010 = 0x410

Physical
0x1600 + 0x010 = 0x1610
0x400 + 0x010 = 0x410

Segment numbers:
0: code+data
1: heap
2: stack
Segment numbers:
  0: code+data
  1: heap
  2: stack

Virtual
load 0x2010, R1
load 0x1010, R1
load 0x1100, R1

Physical
0x1600 + 0x010 = 0x1610
0x400 + 0x010 = 0x410
Virtual
load 0x2010, R1
load 0x1010, R1
load 0x1100, R1

Physical
0x1600 + 0x010 = 0x1610
0x400 + 0x010 = 0x410
0x400 + 0x100 = 0x500

Segment numbers:
0: code+data
1: heap
2: stack
Memory accesses every instruction

0x0010: movl 0x1100, %edi
0x0013: addl $0x3, %edi
0x0019: movl %edi, 0x1100

Physical Memory Accesses?
1) Fetch instruction at logical addr 0x0010
   • Physical addr: 0x4010
   Exec, load from logical addr 0x1100
   • Physical addr: 0x5900
2) Fetch instruction at logical addr 0x0013
   • Physical addr: 0x4013
   Exec, no load
3) Fetch instruction at logical addr 0x0019
   • Physical addr: 0x4019
   Exec, store to logical addr 0x1100
   • Physical addr: 0x5900

<table>
<thead>
<tr>
<th>Seg</th>
<th>Base</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x4000</td>
<td>0xfff</td>
</tr>
<tr>
<td>1</td>
<td>0x5800</td>
<td>0xfff</td>
</tr>
<tr>
<td>2</td>
<td>0x6800</td>
<td>0x7ff</td>
</tr>
</tbody>
</table>

Seg Base Bounds
0 0x4000 0xfff
1 0x5800 0xfff
2 0x6800 0x7ff

%rip: 0x0010
Total of 5 memory references (3 instruction fetches, 2 movl)
Advantages of Segmentation

- Enables **sparser allocation** of memory address space than one base+bounds
  - Stack and heap can grow independently
  - Heap: If no data on free list, dynamic memory allocator requests more from OS (e.g., UNIX: malloc calls sbrk())
  - Stack: OS recognizes reference outside legal segment, extends stack implicitly

- Different protection for different segments
  - Read-only status for code

- Enables sharing of some segments as desired

- Supports dynamic relocation of each segment
Disadvantages of Segmentation?

Each segment must be allocated contiguously
• May not have sufficient physical memory for large segments!

• Cannot support holding a part of a large segment in memory
Disadvantages of Segmentation?

**Fragmentation:** Free memory that can’t be usefully allocated

Why? Free memory (hole) is too small and scattered
- Segmentation prohibits using this free space since segment is “indivisible”

Types of fragmentation
- External: Visible to allocator (e.g., OS)
- Internal: Visible to requester (e.g., if must allocate at some granularity)
HW+OS work together to virtualize memory
  • Give illusion of private address space to each process

Add MMU registers for base+bounds so translation is fast
  • OS not involved with every address translation, only on context switch or errors

Dynamic relocation with segments is good building block
  • Next: Solve fragmentation with paging
Review: Match Description

- Description
  - one process uses RAM at a time
  - rewrite code & addresses before running
  - add per-process starting location to virt addr to obtain phys addr
  - dynamic approach that verifies address is in valid range
  - several base+bound pairs per process

- Name of approach (covered previous lecture):
  - Segmentation
  - Base
  - Static Relocation
  - Time sharing
  - Base + Bounds
Paging

Questions we answer:
  What is paging?
  Where are page tables stored?
  What are advantages and disadvantages of paging?
Paging

Goal: Eliminate requirement that address space is contiguous
- Eliminate external fragmentation
- Grow segments as needed

Idea: Divide address spaces and physical memory into fixed-sized pages
- Size: $2^n$, Example: 4KB
- Physical page: page frame
Translation of Page Addresses

• How to translate logical address to physical address?
  • High-order bits of address designate page number
  • Low-order bits of address designate offset within page

No addition needed; just append bits correctly…

How does format of address space determine number of pages and size of pages?
Impact of Address Format

Given known page size, how many bits are needed in address to specify offset in page?

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
</tr>
<tr>
<td>512 bytes</td>
<td>9</td>
</tr>
<tr>
<td>4 KB</td>
<td>12</td>
</tr>
</tbody>
</table>
## Impact of Address Format

Given number of bits in virtual address and bits for offset, how many bits for virtual page number?

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
<th>Virt Addr Bits</th>
<th>High Bits (vpn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
<td>32</td>
<td>12</td>
</tr>
<tr>
<td>512 bytes</td>
<td>9</td>
<td>16</td>
<td>7</td>
</tr>
<tr>
<td>4 KB</td>
<td>12</td>
<td>32</td>
<td>20</td>
</tr>
</tbody>
</table>
# Impact of Address Format

Given number of bits for vpn, how many virtual pages can there be in an address space?

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Low Bits (offset)</th>
<th>Virt Addr Bits</th>
<th>High Bits (vpn)</th>
<th>Virt Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bytes</td>
<td>4</td>
<td>10</td>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>1 KB</td>
<td>10</td>
<td>20</td>
<td>10</td>
<td>1 K</td>
</tr>
<tr>
<td>1 MB</td>
<td>20</td>
<td>32</td>
<td>12</td>
<td>4 K</td>
</tr>
<tr>
<td>512 bytes</td>
<td>9</td>
<td>16</td>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>4 KB</td>
<td>12</td>
<td>32</td>
<td>20</td>
<td>1 MB</td>
</tr>
</tbody>
</table>