MEMORY VIRTUALIZATION

Shivaram Venkataraman
CS 537, Spring 2020
- Project 1b is due Wednesday
- Project 1a grades this week

- Midterm makeup requests (email or Piazza)
AGENDA / LEARNING OUTCOMES

Memory virtualization
   What are main techniques to virtualize memory?
   What are their benefits and shortcomings?
RECAP
MEMORY VIRTUALIZATION

Transparency: Process is unaware of sharing

Protection: Cannot corrupt OS or other process memory

Efficiency: Do not waste memory or slow down processes

Sharing: Enable sharing between cooperating processes
ABSTRACTION: ADDRESS SPACE

- Program Code: where instructions live
- Heap: contains malloc'd data, dynamic data structures (it grows downward)
- Stack: contains local variables, arguments to routines, return values, etc.

Operating System (code, data, etc.):
- 0KB:
- 64KB: (free)
- 128KB: Process C (code, data, etc.)
- 192KB: Process B (code, data, etc.)
- 256KB: (free)
- 320KB: Process A (code, data, etc.)
- 384KB: (free)
- 448KB: (free)
- 512KB: (free)
MEMORY ACCESS

Initial %rip = 0x10
%rbp = 0x200

0x10: movl 0x8(%rbp), %edi
0x13: addl $0x3, %edi
0x19: movl %edi, 0x8(%rbp)

%rbp is the base pointer:
points to base of current stack frame

%rip is instruction pointer (or program counter)

Fetch instruction at addr 0x10
Exec:
    load from addr 0x208

Fetch instruction at addr 0x13
Exec:
    no memory access

Fetch instruction at addr 0x19
Exec:
    store to addr 0x208
MEMORY VIRTUALIZATION: MECHANISMS
Problem: How to run multiple processes simultaneously?
Addresses are “hardcoded” into process binaries
How to avoid collisions?

Possible Solutions for Mechanisms (covered in this class):
1. Time Sharing
2. Static Relocation
3. Base
4. Base+Bounds
TIME SHARE MEMORY: EXAMPLE
PROBLEMS WITH TIME SHARING?

Ridiculously poor performance

Better Alternative: space sharing!
   At same time, space of memory is divided across processes
   Remainder of solutions all use space sharing
2) STATIC RELOCATION

Idea: OS rewrites each program before loading it as a process in memory
Each rewrite for different process uses different addresses and pointers
Change jumps, loads of static data

```
0x10: movl 0x8(%rbp), %edi
0x13: addl $0x3, %edi
0x19: movl %edi, 0x8(%rbp)
0x1010: movl 0x8(%rbp), %edi
0x1013: addl $0x3, %edi
0x1019: movl %edi, 0x8(%rbp)
```

```c
0x3010: movl 0x8(%rbp), %edi
0x3013: addl $0x3, %edi
0x3019: movl %edi, 0x8(%rbp)
```
Process 1

4 KB
- (free)
- Program Code
- Heap
- (free)

8 KB
- stack
- (free)

12 KB
- Program Code
- Heap
- (free)

16 KB
- stack
- (free)

Process 2

0x1010: movl 0x8(%rbp), %edi
0x1013: addl $0x3, %edi
0x1019: movl %edi, 0x8(%rbp)

0x3010: movl 0x8(%rbp), %edi
0x3013: addl $0x3, %edi
0x3019: movl %edi, 0x8(%rbp)
STATIC RELOCATION: DISADVANTAGES

No protection
  – Process can destroy OS or other processes
  – No privacy

Cannot move address space after it has been placed
  – May not be able to allocate new process
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

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
TRANSLATION ON EVERY MEMORY ACCESS OF USER PROCESS
MMU adds base register to logical address to form physical address

**MMU**

- **registers**
  - **base**
  - **mode**

- **logical address**
- **physical address**

- **mode** = user?
  - yes
  - no

- **32 bits**
- **1 bit**

**Diagram:**
- Logical address flows to the mode decision point.
- If mode is **user**, it flows to the physical address.
- If mode is **not user**, it flows to the base register and then to the physical address.
DYNAMIC RELOCATION WITH BASE REGISTER

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
   Dynamic relocation by changing value of base register!
VISUAL EXAMPLE OF DYNAMIC RELOCATION: BASE REGISTER

Base Register for P1

Base Register for P2

Virtual
P1: load 10, R1
P1: load 200, R1
P2: load 500, R1

Physical
Virtual

P1: load 100, R1

P2: load 1000, R1

P1: store 3072, R1
Virtual  |  Physical  
--- | ---  
P1: load 100, R1  |  load 1124, R1  
P2: load 1000, R1  |  load 5096, R1  
P1: store 3072, R1  |  store 4096, R1  
(3072 + 1024)
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](image)

<table>
<thead>
<tr>
<th>registers</th>
<th>base</th>
<th>bounds</th>
<th>mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bits</td>
<td>32 bits</td>
<td>1 bit</td>
<td></td>
</tr>
</tbody>
</table>
base register
bounds register
Can P1 hurt P2?

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

Physical
load 1124, R1
load 4196, R1
load 5196, R1
load 2024, R1
MANAGING PROCESSES WITH BASE AND BOUNDS

Context-switch: Add base and bounds registers to proc struct
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 requirement
• 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 at different locations initially and move address spaces

  Simple, inexpensive implementation: Few registers, little logic in MMU

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 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 has separate base + bounds register
SEGMENTED ADDRESSING

Process now specifies segment and offset within segment
How does process designate a particular segment?
  – Use part of logical address
    • Top bits of logical address select segment
    • Low 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;

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

How many bits for segment?

How many bits for offset?

Remember:
1 hex digit $\rightarrow$ 4 bits
Segment numbers:
0: code+data
1: heap
2: stack

Virtual (hex)  
Physical

load 0x2010, R1
Segment numbers:
0: code+data
1: heap
2: stack

Virtual
load 0x2010, R1

Physical
0x1600 + 0x010 = 0x1610

load 0x1010, R1
0x400 + 0x100 = 0x500
## Segment Base Bounds RW

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

Remember:
1 hex digit → 4 bits

Translate logical (in hex) to physical

- **0x0240:**
- **0x1108:**
- **0x265c:**
- **0x3002:**
HOW DO STACKS GROW?

Stack goes 16K → 12K, in physical memory is 28K → 24K
Segment base is at 28K

Virtual address 0x3C00 = 15K
  → top 2 bits (0x3) segment ref, offset is 0xC00 = 3K
How do we make CPU translate that?

Negative offset = subtract max segment from offset
    = 3K – 4K = -1K

Add to base = 28K – 1K = 27K
HOW DOES THIS LOOK IN X86

Stack Segment (SS): Pointer to the stack
Code Segment (CS): Pointer to the code
Data Segment (DS): Pointer to the data
Extra Segment (ES): Pointer to extra data
F Segment (FS): Pointer to more extra data
G Segment (GS): Pointer to still more extra data
ADVANTAGES OF SEGMENTATION

Enables sparse allocation of address space

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
- Enables sharing of selected segments
- Read-only status for code

Supports dynamic relocation of each segment
DISADVANTAGES OF SEGMENTATION

Each segment must be allocated contiguously

May not have sufficient physical memory for large segments?

External Fragmentation
NEXT STEPS

Project 1b: Due Wednesday!

Next class: Paging, TLBs and more!