Lecture 6: Memory Virtualization

Operating Systems

Content taken from: https://www.cse.iitb.ac.in/~mythili/os/

https://pages.cs.wisc.edu/~remzi/OSTEP/

Last Two Classes

- **Scheduling Policies:**
  - FIFO, SJF, STCF, RR
  - MLFQ
  - Lottery Scheduling
- **Scheduling metrics:**
  - Turnaround time
  - Response time
  - % share of CPU

Why virtualize memory?

- Because real view of memory is messy!
- Earlier, memory had only code of one running process (and OS code)
- Now, multiple active processes timeshare CPU
  - Memory of many processes must be in memory
  - Non-contiguous too
  - Protection
- Need to hide this complexity from user

Figure 13.2: **Three Processes: Sharing Memory**

Abstraction: (Virtual) Address Space

- Process' view of memory in the system
- Every process assumes it has access to a large space of memory from address 0 to a MAX
  - What is the reality?
- Each process memory contains
  - program code (and static data)
  - heap (dynamic allocations)
  - stack (used during function calls)
- CPU issues loads and stores to virtual addresses

Figure 13.3: **An Example Address Space**

Goals of memory virtualization

- **Transparency:** user programs should not be aware of the messy details
- **Efficiency:** minimize overhead and wastage in terms of memory space and access time
- **Isolation and protection:** a user process should not be able to access anything outside its address space

Example: Memory accesses

```
void func() {
    int x = 3000; // thanks, Perry.
    x = x + 3;    // line of code we are interested in
    ...

128: movl 0x0(%ebx), %eax      ;load 0+ebx into eax
132: addl $0x03, %eax          ;add 3 to eax register
135: movl %eax, 0x0(%ebx)      ;store eax back to mem
```

- Fetch instruction at address 128
- Execute this instruction (load from address 15 KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

How is actual memory reached?

- Address translation from virtual addresses (VA) to physical addresses (PA)
  - CPU issues loads/stores to VA but memory hardware accesses PA
- OS allocates memory and tracks location of processes
- Translation done by memory hardware called Memory Management Unit (MMU)
  - OS makes the necessary information available

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**
- Minimal MMU contains **base register** for translation
  - base: start location for address space

Implementation of Dynamic Relocation: BASE REGISTER

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

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

Example

Example

Example

(Decimal notation)

Virtual	Physical
P1: load 100, R1	

Virtual	Physical
P1: load 100, R1	load 1124, R1

(1024 + 100)

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

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

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

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

P2: load 1000, R1
=> 4096+1000 =
5096 and not 5196

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

P2: load 1000, R1
=> 4096+1000 =
5096 and not 5196

Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 1000, R1	load 2024, R1

P2: load 1000, R1
=> 4096+1000 =
5096 and not 5196

How well does dynamic relocation do with base register for protection?

Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1

Can P2 hurt P1?
Can P1 hurt P2?

P2: load 1000, R1 => 4096+1000 = 5096 and not 5196

P1: load 1000, R1 => 1024+1000 = 2024. It should be load 1000, R1 and not load 100, R1.

How well does dynamic relocation do with base register for protection?

Virtual	Physical	
P1: load 100, R1	load 1124, R1	
P2: load 100, R1	load 4196, R1	
P2: load 1000, R1	load 5196, R1	
P1: load 100, R1	load 2024, R1	
P1: store 3072, R1	store 4096, R1	(3072 + 1024)

Can P2 hurt P1?
Can P1 hurt P2?

P2: load 1000, R1 => 4096+1000 = 5096 and not 5196

P1: load 1000, R1 => 1024+1000 = 2024. It should be load 1000, R1 and not load 100, R1.

Dynamic Relocation with Base Reg + 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 Reg + 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

0 KB
1 KB
P1
2 KB
3 KB
4 KB
P2
5 KB
6 KB
base register
bounds register
P1 is running

0 KB
1 KB
P1
2 KB
3 KB
4 KB
P2
5 KB
6 KB
base register
bounds register
P2 is running

Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1
P1: store 3072, R1	

Can P1 hurt P2?

P2: load 1000, R1 => 4096+1000 = 5096 and not 5196

P1: load 1000, R1 => 1024+1000 = 2024. It should be load 1000, R1 and not load 100, R1.

Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1
P1: store 3072, R1	interrupt OS!

3072 > 1024

Can P1 hurt P2?

P2: load 1000, R1 => 4096+1000 = 5096 and not 5196

P1: load 1000, R1 => 1024+1000 = 2024. It should be load 1000, R1 and not load 100, R1.

Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1
P1: store 3072, R1	interrupt OS!

Can P1 hurt P2?

P2: load 1000, R1 => 4096+1000 = 5096 and not 5196

P1: load 1000, R1 => 1024+1000 = 2024. It should be load 1000, R1 and not load 100, R1.s

Managing Processes with Base and Bounds

Context-switch

- Add base and bounds registers to PCB
- 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

What if don’t change base and bounds registers when switching?

Protection requirement

- User process cannot change base and bounds registers
- User process cannot change to privileged mode

Base and Bounds Advantages

Advantages

- Provides protection (both read and write) across address spaces
- Supports dynamic relocation
  - Can place process at different locations initially and also move address spaces

- Simple, inexpensive implementation
  - Few registers, little logic in MMU
- Fast
  - Add and compare in parallel

Base and Bounds Disadvantages

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

Hardware Requirements	Notes
Privileged mode	*Needed to prevent user-mode processes from executing privileged operations*
Base/bounds registers	*Need pair of registers per CPU to support address translation and bounds checks*
Ability to translate virtual addresses and check if within bounds	*Circuitry to do translations and check limits; in this case, quite simple*
Privileged instruction(s) to update base/bounds	*OS must be able to set these values before letting a user program run*
Privileged instruction(s) to register exception handlers	*OS must be able to tell hardware what code to run if exception occurs*
Ability to raise exceptions	*When processes try to access privileged instructions or out-of-bounds memory*

Figure 15.3: **Dynamic Relocation: Hardware Requirements**

OS Requirements	Notes
Memory management	*Need to allocate memory for new processes;* *Reclaim memory from terminated processes;* *Generally manage memory via free list*
Base/bounds management	*Must set base/bounds properly upon context switch*
Exception handling	*Code to run when exceptions arise;* *likely action is to terminate offending process*

Figure 15.4: **Dynamic Relocation: Operating System Responsibilities**

What is the address space of the OS?

- OS is not a separate process with its own address space
- Instead, OS code is part of the address space of every process
- A process sees OS as part of its code (e.g., library)