

CS 140: Operating Systems

Lecture 9: Virtual Memory

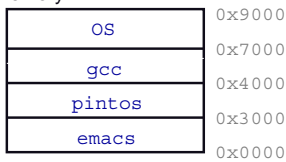
Mendel Rosenblum

Lecture overview

- Virtual memory
 - Maps virtual addresses to physical pages & disk blocks
 - Like processes, a killer OS abstraction: ~40 years old.
 - Today: what its good for, how to build one.
- Readings:
 - Silberschatz 6th ed. Chapter 9
 - Silberschatz 7th ed. Chapter 8

Problem: we want processes to co-exist

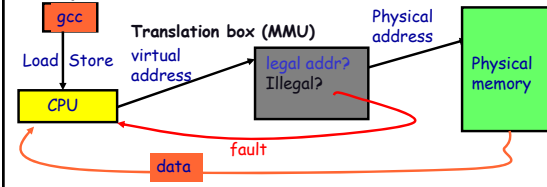
- Consider a primitive system running three processes in physical memory:



What happens if pintos needs to expand?
 If emacs needs more memory than is on the machine??
 If pintos has an error and writes to address 0x7100?
 When does gcc have to know it will run at 0x4000?
 What if emacs isn't using its memory?

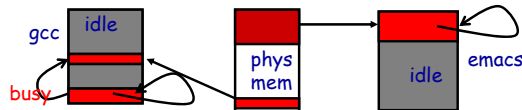
Issues in sharing physical memory

- Protection:** errors in one process should only affect it
 record process's legal address range(s), check that each load and store only references a legal address
- Transparency:** a process should be able to run regardless of its location in or the size of physical memory
 give each process a large, static "fake" address space; as process runs, relocate each load and store to its actual memory



Clever? We get both flexibility and speed!

- VM = indirection between apps and actual memory
 - Flexibility:** process can be moved in memory as it executes, run partially in memory and on disk, ...
 - Simplicity:** drastically simplifies applications
 - Efficiency:** most of a process's memory will be idle (80/20 rule).



Therefore, use fact that we can remap addresses to let other processes use idle parts. Makes memory seem much larger!
 Similar to CPU virtualization: when process not using CPU, switch. When not using page switch.

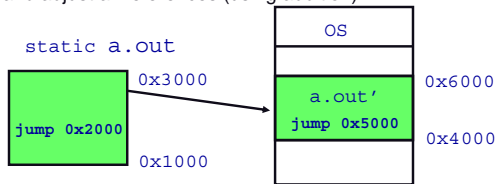
- Challenge: VM = extra layer. Careless = molasses slow.

Our main questions

- How is protection enforced?
- How are processes relocated?
- How is memory partitioned?

Simple idea 1: load-time linking

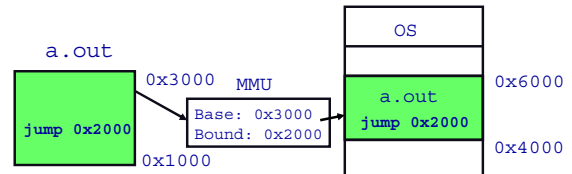
- ◆ Link as usual, but keep the list of references
- ◆ At load time, determine where process will reside in memory and adjust all references (using addition).



- Prob 1: protection?
- Prob 2: how to move in memory? (Consider: data pointers)
- Prob 3: more than one segment?

Simple idea 2: base + bound register

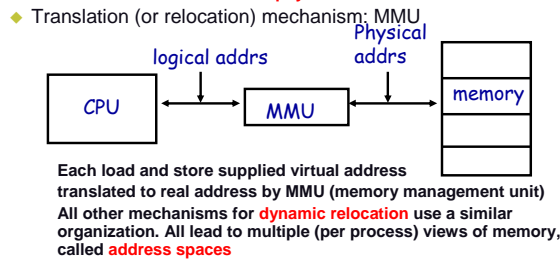
- ◆ Use hardware to solve problem: on every load and store
relocation: $\text{physical addr} = \text{virtual addr} + \text{base register}$
protection: check that address falls in $[\text{base}, \text{base} + \text{bound})$



- When process runs, base register = 0x3000, bounds register = 0x2000.
Jump addr = $0x2000 + 0x3000 = 0x5000$
How to move process in memory? What happens on process switch?

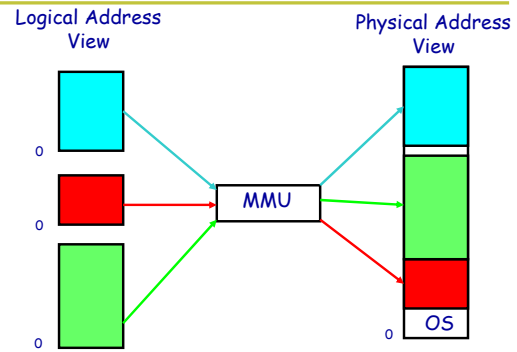
Some terminology

- ◆ Definitions:
program addresses are called **logical** or **virtual addresses**
actual addresses are called **physical** or **real addresses**



- Each load and store supplied virtual address translated to real address by MMU (memory management unit)
- All other mechanisms for **dynamic relocation** use a similar organization. All lead to multiple (per process) views of memory, called **address spaces**

Address spaces



Protection mechanics

- ◆ How to prevent users from changing base/bound register?
- ◆ General mechanism: **privileged instructions**
OS runs in **privileged mode** (set a bit in process status word)
application processes run in **user mode**
Certain instructions can only be issued in privileged mode (checked by hardware: illegal instruction trap)
- ◆ How to switch? ("usually" how its done, many variations)
User->OS: application issues a system call, hardware then:
sets program counter to known address (can't trust user to)
updates process status word
and disables relocation (OS has different address space)
OS->User:
OS sets base and bounds register (recall: relocation off)
issues an instruction that simultaneously (1) sets pc to given address, (2) turns relocation back on, and (3) lowers privilege.

Base&bound tradeoffs

- ◆ Pro:
cheap in terms of hardware: only two registers
cheap in terms of cycles: do add and compare in parallel
examples: Cray-1
- ◆ Con: only one segment
prob 1: growing processes.
How to expand gcc?
prob 2: how to share code and data??
how can nachos copies share code?
prob 3: how to separate code and data?
- ◆ A solution: multiple segments
"segmentation"



Segmentation

- Big idea: let processes have many base&bounds ranges
- Process address space built from multiple "segments". Each has its own base&bound values. Since we can now share, add protection bits for r/w

0x1000 gcc
Text seg r/o
0x3000
0x5000 Stack seg r/w
0x6000

Real memory
0x2000
0x8000
0x6000

Base&bound?

Big result: Lets processes be split across memory!
 Problem: how to specify what segment address refers to?

Segmentation Mechanics

- Each process has an array of its segments (segment table)
- Each memory reference indicates a segment and offset: Top bits of addr select seg, low bits select offset (PDP-10) Seg select by instruction, or operand (pc selects text)

Virtual addr: 3 | 128
 Seg# | offset

Seg table:

Prot	base	len
r	0x1000	512

no fault
 yes +
 0x1000 mem
 128 seg

Segmentation example

- 2-bit segment number (1st digit), 12 bit offset (last 3)

Seg	base	bounds	rw
0	0x4000	0x6fff	10
1	0x0000	0x4fff	11
2	0x3000	0xffff	11
3			00

logical physical

0x4000 0x4700
 0x3000 0x4000
 0x2000 0x3000
 0x1500
 0x1000
 0x0700 0x500
 0x0000 0x0000

- where is 0x0240?
- 0x1108?
- 0x265c?
- 0x3002?
- 0x1600?

Segmentation Tradeoffs

- Pro: Multiple segments per process Allows sharing! (how?) Don't need entire process in memory!!!
- Con: Extra layer of translation speed = hardware support More subtle: an "n" byte segment requires n *contiguous* bytes of physical memory. (why?) Makes fragmentation a real problem.

gcc
 gcc'
 emacs

where?

Fragmentation

- "The inability to use memory that is free".
- Over time: variable-sized pieces = many small holes (external frag) fixed-sized pieces = no external holes, but force internal waste (internal fragmentation)

Pintos 22
 gcc
 emacs
 External fragmentation
 Unused ("internal fragmentation")
 allocated stack doom

Page based virtual memory

- Quantize memory into fixed sized pieces ("pages")

Pages typical: 4k-8k
 gcc
 emacs
 internal frag

- Tradeoff: pro: eliminates external fragmentation pro: simplifies allocation, free and swapping con: internal fragmentation (~.5page per "segment")

Page-based mechanics

memory is divided into chunks of the same size (pages)
 each process has a table ("page table") that maps **virtual page numbers** to corresponding **physical page numbers**
 PT entry also includes protection bits (r, w, valid)
 translation process: virtual page number extracted from an address's upper bits and used as table index.

Page-based translation example

- MIPS R2000: 32 bit addr space, 20-bit VPN and 12-bit offset:

```

    Page number | 
    +-----+
    20 bits
    12 bits
    /* partial page table entry */
    struct pte { unsigned ppn:20, valid:1, writeable:1...; };

    /* given virtual address and r/w indication, return physical
    addr. Uses a simple "direct" page table (i.e., an array) with
    (conceptually) an entry for every possible vpn */
    unsigned xlate(unsigned va, int wr) {
        struct pte *pte = &page_table[va >> 12];
        if(!pte->valid || (wr && !pte->writeable))
            raise address_fault;
        return (pte->ppn << 12) | (va & 0xfff); }
    
```

Page tables vs segmentation

- Good:
 - Easy to allocate: keep a free list of available pages and grab the first one
 - easy to swap since everything is the same size and since pages usually same size as disk blocks
- Bad:
 - size: PTs need one entry for each page-sized unit of virtual memory, vs one entry for every contiguous range. e.g., given a range [0x0000, 0xffff] need one segment descriptor but, assuming 4K pages, 16 page table entries

Page size tradeoffs

- Small page = large PT overhead: 32-bit address space with 1k pages. How big PT?
- Large page = internal frag (doesn't match info. size) most Unix processes have few segments (code, data, stack, heap) so not much of a problem (except for page table memory itself...)

Paging + segmentation: best of both?

- Dual problems:
 - Paged VM: simple page table = lots of storage
 - Segmentation: All bytes in segment must map to contiguous set of storage locations. Makes paging problematic: all of seg in mem or none.
- Idea: use paging + segmentation!
 - Map program mem w/ page table
 - Map page table mem w/ seg table

Paging + segmentation tradeoffs

- Page-based virtual memory lets segments come from non-contiguous memory
 - makes allocation flexible
 - portion of segment can be on disk!
- Segmentation = way to keep page table space manageable
 - Page tables correspond to logical units (code, stack). Often relatively large.
 - (But what happens if page tables really large?)
- Going from paging to P+S is like going from single segment to multiple segments, except at a higher level.
 - Instead of having a single page table, have many page tables with base and bound for each.

Example: system 370

- System 370: 24 bit virtual address space (old machine):

Seg # (4 bits)	page # (8 bits)	page offset (12 bits)

4 bits of segment #
8 bits of page #
12 bits of offset:

- The mappings:
 - Segment table: maps segment number to physical address & size of that segment's page table.
 - Page table maps virtual page to 12 bit physical page #, which is concatenated to the 12 bit offset to get a 24 bit address

Example system 370 translation

Table entry addresses

Base bound prot	Page table (2-byte entries)	Table entry addresses
0 0x2000 0x14 R		0x001f 0x2020
1 0x0000 0x00		... 0x0003
2 0x1000 0x0d RW		0x002a 0x2000
		... 0x000c
		0x0007 0x1020
		... 0x0004
		0x000b 0x1000
		0x0006 0x1000

Segment table

Read of VA 0x2070
0x0 02 070

SEG: 0 page 2 * 2 bytes + 0x2000 = 0x2004
(0x3 << 12 | 0x070) = 0x3070

0x202016 read? 0x104c84 read?
0x011424 read? 0x210014 write?

P+S discussion

- If segment not used then no need to have page table for it
- Can share at two levels:
 - single page or single segment (whole page table)
- Pages eliminate external fragmentation and make it possible for segments to grow, page without shuffling
- If page size small compared to most fragments then internal fragmentation not too bad (.5 of page per seg)
- User not given write access to page tables
 - read access can be quite useful (e.g., to let garbage collectors see dirty bits), but only experimental OSes provide this...
- If translation tables kept in main memory, overhead high
 - 1 or 2 overhead reference for every real ref (i.e., mem op)
- Other example systems: VAX, Multics, x86, ...

Who won?

- Simplicity = good (and fast): Most new architectures have gone with pages
 - Examples: MIPS, SPARC, Alpha.
 - And many old architectures have added paging even if they started with segmentation!
- But: to efficiently map large regions, many new machines backsliding into "super pages"
 - large, multi-page pages: have strict alignment restrictions; (typically) small # of sizes

16K superpage

32K alignment (VA%32K = 0)

4K page

inflexibility = speed, but handles 80% of the cases we want (e.g., that 8MB framebuffer)

Virtual memory summary

- VM gives
 - flexibility + protection + speed (if clever)
- Base&bounds = simple relocation + protection
 - Pro: simple, fast
 - Con: inflexible
- Segmentation = generalization of base & bounds.
 - Pro: Gives more flexible sharing and space usage
 - Con: segments need contiguous physical memory ranges
- Paging: instead of using extents, use fixed sized units
 - Quantize memory into pages & use (page) table to map virtual to physical pages
 - Pro: Eliminates external fragmentation; flexible mappings
 - Con: internal frag; mapping contiguous ranges more costly