

Principles of Operating Systems

Virtual Memory – Segmentation

Thomas Ropars

`thomas.ropars@univ-grenoble-alpes.fr`

2024

References

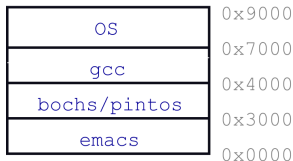
- These slides are adapted from the slides of Renaud Lachaize
- Chapters of *Operating Systems: Three Easy Pieces*
 - ▶ Chapter 15: Address Translation
 - ▶ Chapter 16: Segmentation

Outline

- **The need for virtual memory**
- How to implement virtual memory?
 - 1st attempt: Load-time linking
 - 2nd attempt: Registers and MMU
 - 3rd attempt: Segmentation

Motivating example

Processes coexisting in memory



- Consider multiprogramming in physical memory
 - What happens if one application needs to expand?
 - What happens if one application needs more memory than what is on the machine?
 - What happens if pintos is buggy and writes to 0x7100?
 - When does gcc have to know that it will run at 0x4000?
 - What if emacs is not using its whole memory range?

Issues in sharing physical memory

- **Protection**

- A bug in one process can corrupt memory in another
- How to prevent process A from trashing B's memory?
- How to prevent A from observing B's memory?

- **Transparency**

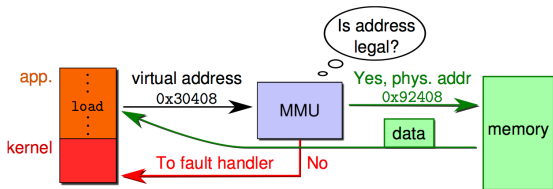
- A process should not require particular/fixed memory locations
- Processes often require large amount of contiguous memory (for stack, large data structures, etc.)

- **Resource exhaustion**

- Programmers typically assume that a machine has “enough” memory
- The sum of sizes of all processes is often greater than physical memory

Introducing virtual memory

Goals

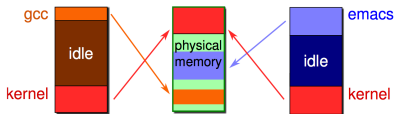


- **Give each program its own “virtual” address space**
 - At run time, redirect each load/store instruction to its actual memory
 - ... So that the application does not care what physical memory it is using
- **Enforce protection**
 - Prevent one application from messing with another’s memory
- **Allow programs to see more memory than exists**
 - Somehow relocate some memory accesses to disk

Introducing virtual memory

Advantages

- Can re-locate program (code/data) while running
 - Run partially in memory, partially on disk
- In many cases, most of the memory of a process is idle (80/20 rule)



- Write idle part to disk until needed
- Let other processes use memory for idle part
- Analogy with CPU virtualization:
 - When process not using CPU, switch
 - When not using a physical page, switch it to another process
- Challenge: the virtual memory subsystem is an extra layer
 - Could cause slowdown

Introducing virtual memory

How to implement it it?

We will consider several approaches.

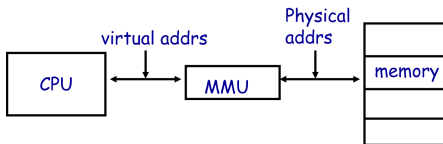
Idea 2: base + bound registers

- Introduce two special privileged (hardware) registers: base and bound
- On each load/store:
 - Compute phys. addr. = virt. addr. + base
 - Check $0 \leq \text{virt. addr.} < \text{bound}$, else trap to kernel
- How to move a process in memory?
 - Change base register
- What happens on context switch?
 - OS must reload/modify base and bound registers

Virtual memory

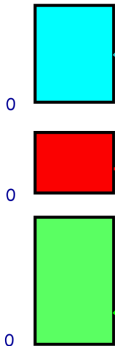
Definitions

- Programs manipulate **virtual** (a.k.a. “logical”) addresses
- The actual memory uses **physical** (a.k.a. “real”) addresses
- Hardware uses a special component: **Memory Management Unit (MMU)**
 - Usually part of the CPU
 - Accessed with privileged instructions
 - Translates from virtual to physical addresses
 - Provides a per-process view of the memory, called address space

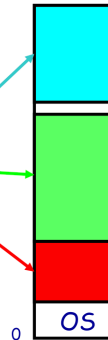


Address space

Virtual Address View



Physical Address View



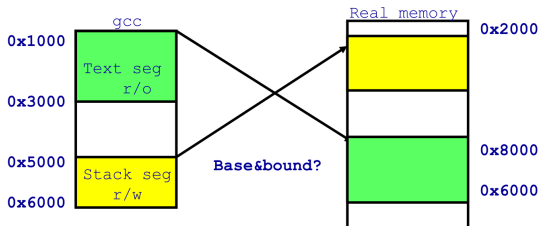
MMU

Base+bound trade-offs

- Advantages
 - Cheap to implement in hardware
 - Cheap in terms of cycles: do add and compare in parallel
- Disadvantages
 - Growing the memory of a process is expensive or impossible
 - No way to share code or data
 - (e.g., multiple copies of the same application and/or multiple applications accessing the same file)
- One solution: Multiple segments
 - E.g., separate code, stack and data segments
 - Possibly multiple data segments per process

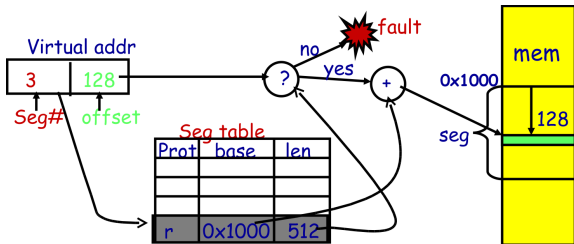


Segmentation



- Let processes have many base/bound registers
 - Address space built from many segments
 - Can share/protect memory on segment granularity
- Segment must be specified as part of virtual address

Segmentation mechanics



- Each process has a segment table
- Each virt. addr. (VA) indicates a segment and an offset
 - Top bits of addr. select segment, low bits select offset
 - Or segment selected implicitly by instruction or operand
 - This means you need wider pointers (“far pointers”) to specify segment

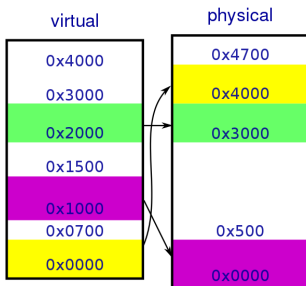
Segmentation example

Seg	base	bounds	rw
0	0x4000	0x6ff	10
1	0x0000	0x4ff	11
2	0x3000	0xfff	11
3			00

- 2-bit segment number (1st digit), 12-bit offset (last 3 digits)
 - Where is 0x0240? 0x1108? 0x265c? 0x3002? 0x1600?

Segmentation example

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



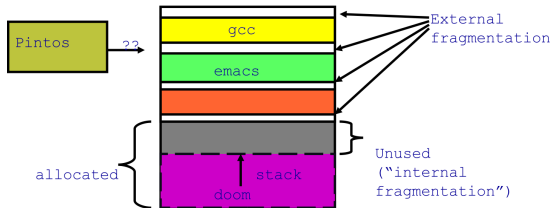
- 2-bit segment number (1st digit), 12-bit offset (last 3 digits)
 - Where is 0x0240? 0x1108? 0x265c? 0x3002? 0x1600?

Segmentation trade-offs

- Advantages
 - Multiple segments per process
 - Allows sharing (how?)
 - Does not need to store entire process in memory at any moment
- Disadvantages
 - Requires translation hardware, which could limit performance
 - N-byte segment needs N contiguous bytes of physical memory
 - Makes fragmentation a real problem

Fragmentation

- Fragmentation: inability to use free memory
- Over time :
 - Variable-sized pieces: many small holes (**external fragmentation**)
 - Fixed-size pieces: no external holes, but force internal waste (**internal fragmentation**)



- In the next lecture, we will study a better solution for the virtual memory implementation problem, which does not suffer from fragmentation