CS 537
Lecture 5
Memory

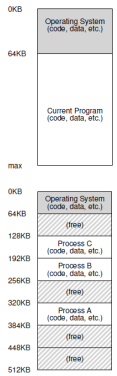
Michael Swift & Aaron Gember

Recall & Assumptions

- What do you already know about memory?
  - Divided into bytes and words
    - Each byte is uniquely addressable
  - Each process has its own memory space
    - Code, global data, heap, stack

Motivation

- Does a process get all of memory?
  - In early systems, yes
    - Requires storing memory to disk to switch processes
- Why do we want to share memory?
  - Multiple processes can reside in memory at once
    - Reduces overhead when we context switch
- How do we allow processes to still have their own view of memory?
  - Virtual memory
- What happens if we don't do this?
  - Hard code physical memory addresses in programs



Virtual Memory

- Basic abstraction OS provides for managing memory
- What should virtual memory provide?
  - Transparency
    - Process should not realize it is sharing memory or that memory is virtualized
  - Efficiency
    - Memory access should be quick
    - Memory should be appropriately divided between processes
    - Do not allocate space a process does not need
    - Program can execute without entire its address space in RAM
  - Protection
    - Protect processes from changing each other's memory or the OS's memory
    - Protect processes from itself – e.g., code is read-only

Key Terms

- Virtual address
  - Identifier used by a processes (instructions) to access data
  - Independent of location of data in physical memory
  - Range of virtual addresses depends on hardware (e.g., 32-bit)
- Address space
  - Range of virtual addresses a process can reference
  - Using an invalid address causes a segmentation fault

5

Key Terms

- Translation
  - Conversion between virtual address and physical address
  - Performed by hardware, with help from the OS
  - Can happen in many different ways
  - Not all virtual addresses need to map to real memory
    - These are untranslated or invalid addresses

6

Example Program Execution

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

- Read instruction from memory
- Read data from memory
- Read instruction from memory
- Read instruction from memory
- Write data to memory

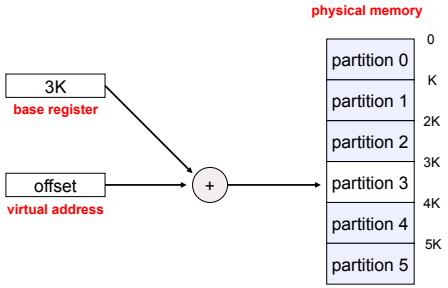
7

Old technique #1: Fixed Partitions

- Physical memory is broken up into fixed partitions
  - all partitions are equally sized, partitioning never changes
  - hardware requirement: base register
    - physical address = virtual address + base register
    - base register loaded by OS when it switches to a process
- Example (base=3K, partition size=1K)
  - Read instruction from memory (3072+128 = 3200)
  - Read data from memory (3072+896 = 3968)
  - Read instruction from memory (3072+132 = 3204)
  - Read instruction from memory (3072+136 = 3208)
  - Write data to memory (3072+896 = 3968)
- What if we were in partition 4?

8

Fixed Partitions (K bytes)



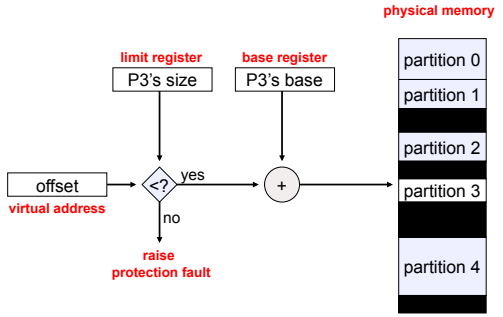
Old technique #1: Fixed Partitions

- How can we ensure protection?
- Advantages
  - simple, ultra-fast context switch
- Problems
  - internal fragmentation: memory in a partition not used by its owning process isn't available to other processes
  - partition size problem: no one size is appropriate for all processes
    - fragmentation vs. fitting large programs in partition

Old technique #2: Variable Partitions

- Obvious next step: physical memory is broken up into variable-sized partitions
  - hardware requirements: base register, limit register
  - physical address = virtual address + base register
  - how do we provide protection?
    - if (physical address > base + limit) then… ?
- Example (base=3K, bound = 512)
  - Read instruction from memory (3072+128 = 3200)
  - Read data from memory (3072+896 = 3968) FAULT
  - Read instruction from memory (3072+132 = 3204)
  - Read instruction from memory (3072+136 = 3208)
  - Write data to memory (3072+896 = 3968) FAULT

Variable Partitions



Old technique #2: Variable Partitions

- Advantages
  - no internal fragmentation
    - simply allocate partition size to be just big enough for process (assuming we know what that is!)
- Problems
  - external fragmentation
    - as we load and unload jobs, holes are left scattered throughout physical memory

OS Responsibilities

- Maintain a free list
  - On process create, find a free slot, allocate to process, mark as "used"
  - On process termination, reclaim memory by putting back on free list
- On context switch, save the base and bounds registers for P1, restore for P2

Recap Example #1

Code

```
0: movl 0x0(%ebx), %eax
4: addl $0x01, %eax
8: movl %eax, 0x0(%ebx)
```

- How might physical memory be laid out while the process is running?
- List all memory-related actions performed by the OS as the process runs
- List all memory-related actions performed by hardware as the process runs

Recap Example #2

Virtual Address	Physical Address
0x0000 (0)	0x1000 (4096)
0x03FC (1020)	0x13FC (5116)
0x0400 (1024)	FAULT
0x0300 (768)	0x1300 (4864)

- What part of a processes address space is located at each of the first two virtual addresses?
- What are the base and bounds values?