Virtualizing Memory

- Memory virtualization goals
- Address translation and dynamic relocation
- Segmentation
- Dealing with fragmentation

COSC 301: Operating Systems
28 September 2012

Joel Sommers
jsommers@colgate.edu
Colgate University

Learning goals

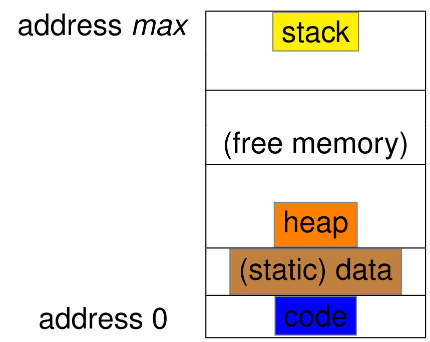
- Describe a base and bounds-based dynamic memory relocation scheme for supporting multiprogramming.
- Compare simple dynamic memory relocation and segmentation as approaches for allocating memory to user processes.
- Describe and compare algorithms for memory allocation, e.g., first, next, best fit, and the buddy algorithm.

Memory virtualization goals

Main goal: extend limited direct execution to memory references to create illusion of a private address space for each process

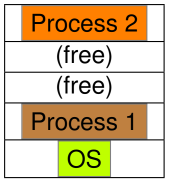
- Mechanism: address translation
- Want it to be efficient, transparent, and controlled (protected)
- What about isolation?

Process Logical Address Space



- logical address space == virtual address space
- Linear address space from address 0 to *max*
- Purely conceptual view of a process's memory
  - Physical memory address space ≠ logical process address space

Dynamic Relocation



- Assume that P1 base physical address is 1024, base physical address for P2 is 4096.
- Say P1 and P2 each execute `load 100, R1` instruction — what should happen?
  - How to map (translate) *virtual* memory reference to physical address?
  - What if P1 issues instruction `load 4000, R1`?

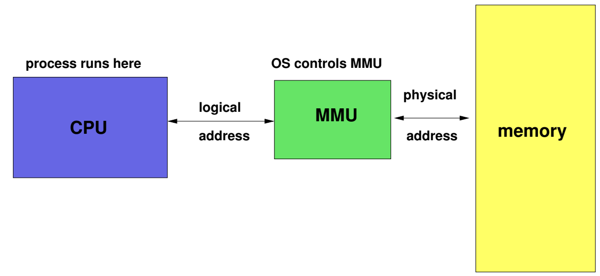
Dynamic Relocation

We need hardware support!

- Memory management unit (MMU)

MMU dynamically changes process address at every memory reference

- Process generates logical or virtual addresses
- Memory hardware uses physical or real addresses



Hardware Support for Dynamic Relocation

Two operating modes

- Privileged (protected, kernel) mode: OS
  - 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
- Use mode: user processes
  - Perform translation of logical address to physical address

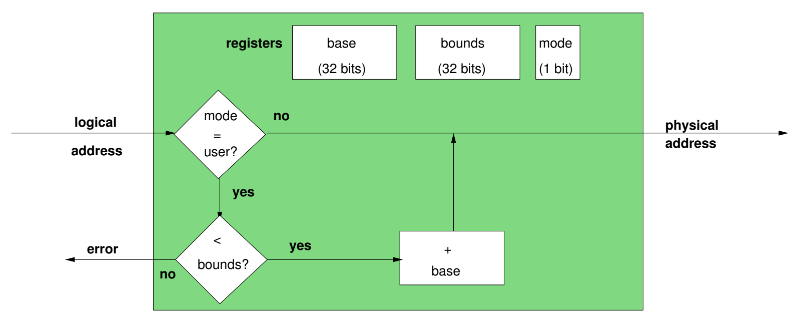
MMU contains base and bounds registers

- base: start location for process address space
- bounds: size limit of address space

Implementation of Dynamic Relocation

Translation on every memory access of user process

- MMU compares logical address to bounds register
  - If logical address is greater, then generate error (kill process)
- MMU adds base register to logical address to form physical address



Example of Dynamic Relocation

What are the physical addresses for the following 16-bit logical addresses?

- Process 1: base=0x4320, bounds=0x2220
  - 0x0000:
  - 0x1110:
  - 0x3000:
- Process 2: base=0x8450, bounds=0x3330
  - 0x0000:
  - 0x1110:
  - 0x3000:
- OS
  - 0x0000:

OS involvement with Dynamic Relocation

Process creation

- Initialize base and bounds (how?)

Growth of address space

- Need to track free/available physical memory
- Adjust bounds register as necessary, somehow

Context-switch

- Add base/bounds to PCB
- Save base/bounds of P1, restore base/bounds of P2

Base and Bounds Discussion

Advantages

- Fast and simple!
  - Add and compare can be done in parallel
- Supports memory virtualization with little overhead
  - Can change where a process is physically located
  - Why might we want to do this?

Disadvantages

- Each process must be allocated contiguously in physical memory
  - Must allocate memory that may not be used by process (how big should the heap be? the stack?)
- No fine-grain protection beyond protecting complete address spaces
  - E.g., read/write/execute permissions for different parts of memory
- No partial sharing; cannot share limited parts of an address space

Segmentation

Basic idea: divide address space into logical segments

- A *generalization* of dynamic relocation
- Each segment corresponds to logical entity in address space
  - Code, stack, heap

Each segment can independently:

- be placed (and relocated) separately in physical memory
- grow and shrink
- be protected (separate read/write/execute protection bits)

Segmented Addressing

How does process designate a particular segment?

- Explicit: 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
- E.g., if instruction fetch, then use code segment

Segmentation Implementation

Virtualizing Memory
Memory Virtualization
Dynamic relocation
Segmentation
Fragmentation

MMU contains segment table (per process)

- Each segment has own base and bounds, protection bits
- Example: 16 bit logical address, max segment address $2^{14} - 1$, 4 segments (2 high bits)

Segment	Base	Bounds	R W X
0	0x2000	0x06ff	1 0 0
1	0x0000	0x04ff	1 0 1
2	0x3000	0x0fff	1 1 0
3	0x0000	0xffff	0 0 0

Translate logical addresses to physical addresses:

- 0x0240
- 0x4508
- 0x865c
- 0xc002

Discussion of Segmentation

Virtualizing Memory
Memory Virtualization
Dynamic relocation
Segmentation
Fragmentation

Advantages

- Enables sparse allocation of address space
  - Stack and heap can grow independently
- Different protection for different segments
  - Read (R), Write (W), Execute (X) for each segment
  - E.g., RX for code, RW for heap and stack
- Enables sharing of selected segments
- Supports dynamic relocation of each segment

Disadvantages

- Each segment must be allocated contiguously
  - Can lead to external fragmentation
  - May not have sufficient physical memory for large segments

Fragmentation

Virtualizing Memory
Memory Virtualization
Dynamic relocation
Segmentation
Fragmentation

Fragment definition: free memory that is too small to be usefully allocated

- External: visible to allocator
- Internal: visible to requester (*e.g.*, if must allocate at some granularity; want 8 bytes but can only obtain in 16 byte quantities)

Goal: minimize fragmentation

- Few holes, each hole is large
- Free space is contiguous

Fragmentation can occur from the perspective of two different allocators:

- OS kernel: allocate memory to a process segment
- Heap allocation: allocate within the heap segment of a process

Memory allocation algorithms

Best fit

- Search entire list for each allocation
- Choose free block that most closely matches size of request
- Optimization: stop searching if exact match found

First fit

- Allocate first block that is large enough to fulfill request

Rotating first fit

- Variant of first fit, remember place in list
- Start with next free block after most recently allocated block

Worst fit

- Allocate largest block to request (leaving most left-over space)

Memory allocation examples

Scenario: two free blocks of size 20 and 15 bytes
Allocation stream: 10, 20

- Best
- First
- Worst

Allocation stream: 8, 12, 12

- Best
- First
- Worst

Buddy Allocation

Fast, simple allocation for blocks of 2^n bytes [Knuth]

- Raise allocation request to nearest $s = 2^n$
- Search free list for appropriate size
  - Represent free list with bitmap
  - Recursively divide larger free blocks until find free block of size s
  - “Buddy” block remains free
- Mark corresponding bits in free bitmap

To release memory:

- Mark bits as free
- Recursively coalesce freed block with buddy (if buddy if free)
  - May lazily coalesce to defer overhead

Buddy Allocation Example

Scenario: 1 MB of free memory
Request stream:

- Allocate 70 KB, 35 KB, 80 KB
- Free 35 KB, 80 KB, 70 KB

Comparison of Allocation Strategies

No optimal algorithm

- Fragmentation highly dependent on workload

Best fit

- Tends to leave some very large holds and some very small holes; can't use small holes easily

First fit

- Tends to leave "average" sized holes
- Advantage: faster than best fit
- Next fit used often in practice

Buddy allocation

- Minimizes external fragmentation
- Disadvantage: internal fragmentation when requests are not 2^n