Topic 4: Virtual Memory - How can one set of memory be shared among several processes? How can the operating system manage access to a limited amount of system memory?
Virtual Memory Introduction

Dynamic Address Translation

Paging

Demand Paging and the Clock Algorithm

Lecture 21

Today

Lecture 23

Lecture 24

**assign6**: implement *demand paging* system to translate addresses and load/store memory contents for programs as needed.
Learning Goals

• Understand the benefits of dynamic address translation
• Reason about the tradeoffs in different ways to implement dynamic address translation
Plan For Today

• **Recap:** virtual memory and dynamic address translation
• Approach #1: Base and Bound
• Approach #2: Multiple Segments
Plan For Today

• Recap: virtual memory and dynamic address translation
  • Approach #1: Base and Bound
  • Approach #2: Multiple Segments
Virtual memory is a mechanism for multiple processes to simultaneously use system memory.
We want to allow multiple processes to simultaneously use system memory. Our goals are:

- **Multitasking** – allow multiple processes to be memory-resident at once
- **Transparency** – no process should need to know memory is shared. Each must run regardless of the number and/or locations of processes in memory.
- **Isolation** – processes must not be able to corrupt each other
- **Efficiency** (both of CPU and memory) – shouldn’t be degraded badly by sharing
Load-Time Relocation

- When a process is loaded to run, place it in a designated memory space.
- That memory space is for everything for that process – stack/data/code
- Interesting fact – when a program is compiled, it is compiled assuming its memory starts at address 0. Therefore, we must update its addresses when we load it to match its real starting address.
- Use first-fit or best-fit allocation to manage available memory.
- **Problems:** isolation, deciding memory sizes in advance, fragmentation, updating addresses when loading
Idea: What if, instead of translating addresses when a program is loaded, the OS intercepted every memory reference and translated it?
Dynamic Address Translation

Let’s have the OS intercept every memory reference a process makes.
• The OS can prohibit processes from accessing certain addresses (e.g. OS memory or another process’s memory)
• Gives the OS lots of flexibility in managing memory
• Every process can now think that it is located starting at address 0
• The OS will translate each process’s address to the real one it’s mapped to
We will add a *memory management unit* (MMU) in hardware that changes addresses dynamically during every memory reference.

- *Virtual address* is what the program sees
- *Physical address* is the actual location in memory
Dynamic Address Translation

• Every process can think it starts at address 0 and is the only process in memory
• Behind the scenes, the OS can choose how it maps each process’s virtual addresses to real (“physical”) addresses
• As a result, a process’s virtual address space may look very different from how the memory is really laid out in the physical address space.
Key Idea: there are now two views of memory, and they can look very different:

- **Virtual address space** is what the program sees
- **Physical address space** is the actual allocation of memory
Key Question: How do the MMU/OS translate from virtual addresses to physical ones?
Plan For Today

• **Recap:** virtual memory and dynamic address translation
  • Approach #1: Base and Bound
  • Approach #2: Multiple Segments
  • Approach #3: Paging
Plan For Today

- Recap: virtual memory and dynamic address translation
- Approach #1: Base and Bound
- Approach #2: Multiple Segments
Key Idea: Let’s use the load-time relocation idea of contiguous allocation, but with the MMU.

• Every process’s virtual address space is mapped to a contiguous region of physical memory.

• When a program accesses a virtual address, translate it by adding the **base** for that process – the physical address its memory really starts at.

• We specify the process’s memory size by setting a **bound** for it; if a process accesses an invalid virtual address above the bound, OS triggers an error.
Approach #1: Base and Bound

- “base” is physical address starting point – corresponds to virtual address 0
- “bound” is highest allowable *virtual* memory address
- Each process has own base/bound. Stored in PCB and loaded into two registers when running.

On each memory reference:
- Compare virtual address to bound, trap if $\geq$ (invalid memory reference)
- Otherwise, add base to virtual address to produce physical address
Approach #1: Base and Bound

Example: let’s say process A has base = 1000, bound = 5000. What happens if:

• It accesses virtual address 6000?
• It accesses virtual address 0?
Example: let’s say process A has base = 1000, bound = 5000. What happens if:

- It accesses virtual address 6000? Invalid memory reference.
- It accesses virtual address 0? Accesses physical address 1000.
Process B has base = 6000, bound = 2000. What happens when it accesses virtual addresses 1) 6000 and 2) 1000?

- Accesses 1) physical address 12000 and 2) physical address 7000
- Accesses 1) physical address 0 and 2) physical address 3000
- 1) Invalid memory reference and 2) physical address 7000
- Gets memory errors for both references
Process B has base = 6000, bound = 2000. What happens when it accesses virtual addresses 1) 6000 and 2) 1000?

Accesses 1) physical address 12000 and 2) physical address 7000

Accesses 1) physical address 0 and 2) physical address 3000

1) Invalid memory reference and 2) physical address 7000

Gets memory errors for both references
Process B has base = 6000, bound = 2000. What happens when it accesses virtual addresses 1) 6000 and 2) 1000?

Accesses 1) physical address 12000 and 2) physical address 7000

Accesses 1) physical address 0 and 2) physical address 3000

1) Invalid memory reference and 2) physical address 7000

Gets memory errors for both references

✓ 0%
Approach #1: Base and Bound

• Key idea: each process appears to have a completely private memory whose size is determined by the bound register.

• The only physical address is in the base register, controlled by the OS. Process sees only virtual addresses!

• OS can update a process’s base/bound if needed! E.g. it could move physical memory to a new location or increase bound.
Approach #1: Base and Bound

What are some benefits of this approach?
• Inexpensive translation – just doing addition
• Doesn’t require much additional space – just per-process base + bound
• The separation between virtual and physical addresses means we can move the physical memory location and simply update the base, or we could even swap memory to disk and copy it back later when it’s actually needed.

What are some drawbacks of this approach?
• One contiguous region per program
• Fragmentation
• Growing can only happen upwards with the bound
• Doesn’t support read-only regions of memory within a process
Idea: what if we broke up the virtual address space into segments and mapped each segment independently?
Plan For Today

• **Recap:** virtual memory and dynamic address translation
• Approach #1: Base and Bound
• **Approach #2:** Multiple Segments
Key Idea: Each process is split among several variable-size areas of memory, called segments.

• E.g. one segment for code, one segment for data/heap, one segment for stack.

• The OS maps each segment individually – each segment would have its own base and bound, and these are stored in a segment map for that process.

• We can also store a protection bit for each segment; whether the process is allowed to write to it or not in addition to reading.

• Now each segment can have its own permissions, grow/shrink independently, be swapped to disk independently, be moved independently, and even be shared between processes (e.g. shared code).
Approach #2: Multiple Segments

On each memory reference:

- Look up info for the segment that address is in
- Compare virtual address to that segment’s bound, trap if \( \geq \) (invalid memory reference)
- Add segment’s base to virtual address to produce physical address

**Problem:** how do we know which segment a virtual address is in?
Approach #2: Multiple Segments

Problem: how do we know which segment a virtual address is in?

One Idea: make virtual addresses such that the top bits of the address specify its segment, and the low bits of the address specify the offset in that segment.

Another possibility: deduce from machine code instruction executing
Approach #2: Multiple Segments

What are some benefits of this approach?
• Flexibility – can manage each segment independently
• Can share segments between processes
• Can move segments to compact memory and eliminate fragmentation

What are some drawbacks of this approach?
• Variable-length segments result in memory fragmentation – can move, but creates friction
• Typically small number of segments
• Encoding segment + offset rigidly divides virtual addresses (how many bits for segment vs. how many for offset?)
Idea: what if we broke up the virtual address space not into variable-length segments, but into fixed-size chunks?
Recap

- Recap: virtual memory and dynamic address translation
- Approach #1: Base and Bound
- Approach #2: Multiple Segments

Next time: approach #3 - paging

Lecture 22 takeaway: Dynamic Address translation means that the OS intercepts and translates each memory access. Initial approaches to this include base+bound per process, and then expanding that to be base+bound per variable-length segment.