CS111, Lecture 21
Virtual Memory Introduction

Optional reading:
Operating Systems: Principles and Practice (2nd Edition): Chapter 8
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 - 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?

Why is answering this question important?

• We can understand one of the most “magical” responsibilities of OSes – making one set of memory appear as several!

• Exposes challenges of allowing multiple processes to share memory while remaining isolated

• Allows us to understand exactly what happens when a program accesses a memory address

assign6: implement demand paging system to translate addresses and load/store memory contents for programs as needed.
Virtual Memory Introduction

Dynamic Address Translation

Demand Paging

The Clock Algorithm

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

• Understand the goals of sharing memory
• Reason about the tradeoffs in implementing memory sharing mechanisms
• Understand what impact virtual memory has on our programs
Plan For Today

• Goals of sharing memory
• Single-tasking
• Load-time relocation
• Introducing virtual memory
• Dynamic address translation
  • Approach #1: Base and Bound

```
cp -r /afs/ir/class/cs111/lecture-code/lect21 .
```
Plan For Today

• Goals of sharing memory
  • Single-tasking
  • Load-time relocation
  • Introducing virtual memory
  • Dynamic address translation
    • Approach #1: Base and Bound

```bash
cp -r /afs(ir/class/cs111/lecture-code/lect21 .
```
So far, we’ve seen how the OS can run multiple threads or processes concurrently by sharing CPU cores (e.g. taking turns with a single core). Another key sharing aspect: they must share a limited amount of system memory.
Virtual memory is a mechanism for multiple processes to simultaneously use system memory.
What are our goals for sharing memory?

- **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

To understand how we can share memory, let’s first look at what a single process’s memory needs are.
Plan For Today

• Goals of sharing memory
• Single-tasking
• Load-time relocation
• Introducing virtual memory
• Dynamic address translation
  • Approach #1: Base and Bound

```bash
cp -r /afs(ir/class/cs111/lecture-code/lect21 .
```
Let’s start with a system that can just run one user process at a time. What does memory look like?

- A process’s memory is a collection of segments (sections)
- **Code** (“text”) – program code
- **Data** – constants, heap
- **Stack** – stack frames for functions
- Stack grows down, heap grows up as more space is needed

(for Unix/Linux – Windows essentially the same)
Let’s start with a system that can just run one user process at a time. What does memory look like?

- The OS also needs memory space!
- Reserve highest memory addresses for OS
- **Problem:** rogue programs could mess with OS memory, corrupt the system

**Challenge:** to run multiple processes, how can we split up memory to give each process space?
Pre-virtual-memory-idea
#1: Let’s reserve contiguous blocks in memory for each process.
Plan For Today

- Goals of sharing memory
- Single-tasking
- **Load-time relocation**
- Introducing virtual memory
- Dynamic address translation
  - Approach #1: Base and Bound

```bash
cp -r /afs/ir/class/cs111/lecture-code/lect21 .
```
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.

What are the problems with this approach?
What are the problems with this approach?

- No isolation – one process can corrupt another or the OS
- Must decide process memory size ahead of time
- Challenges with allocating memory for new processes – memory fragmentation
- Can’t move once we load the process
- Need to update pointers in executable before running
Idea #2: Instead of translating addresses at load-time, let’s translate **on the fly**. We’ll put every process in its own “virtual world” where it thinks it can access all addresses, and the OS will translate every memory access to a “real address”.
Plan For Today

- Goals of sharing memory
- Single-tasking
- Load-time relocation
- **Introducing virtual memory**
- Dynamic address translation
  - Approach #1: Base and Bound

```bash
cp -r /afs/ir/class/cs111/lecture-code/lect21 .
```
Introducing Virtual Memory

Virtual memory is a mechanism that allows multiple processes to simultaneously use system memory.

• Program addresses are *virtual* (fake) – the OS maps them to *physical* (real) addresses in memory.

• The OS must keep track of virtual -> physical “translations” and translate every memory access.

• The OS doesn’t need to map all virtual addresses unless needed – it can provide new memory on the fly, and can give more than computer actually has.

• The OS can even temporarily kick memory contents to disk until a program needs it again.

• Example of *virtualization* – making one thing look like another, or many of them
Demo: Virtual Memory Implications

memory.c and htop

cp -r /afs(ir/class/cs111/lecture-code/lect21 .
Introducing Virtual Memory

Virtual memory is a mechanism that allows multiple processes to simultaneously use system memory.

Three key questions:
• What are the benefits of the OS intercepting memory addresses?
• How does the OS translate from virtual to physical addresses?
• What are the tradeoffs in different virtual memory implementations?
Plan For Today

• Goals of sharing memory
• Single-tasking
• Load-time relocation
• Introducing virtual memory

• **Dynamic address translation**
  • Approach #1: Base and Bound
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, and can have different translations for each process

**Problem:** intercepting and translating *every* memory reference is expensive!

**How can we do this?**

**Solution:** hardware support
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

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
Dynamic Address Translation

- **Transparency** – virtual addresses allow a program’s view of memory to be different than the real view; doesn’t know its memory is e.g., split up.
- **Isolation** – OS intercepts memory references and can prevent rogue accesses

**Key question:** how does the MMU translate from a virtual address to a physical address? *We’ll see several different approaches over the next few lectures.*
Dynamic Address Translation

**Key question:** how do the MMU / OS translate from virtual addresses to physical ones? Three designs we’ll consider:

1. Base and bound
2. Multiple Segments
3. Paging
Plan For Today

• Goals of sharing memory
• Single-tasking
• Load-time relocation
• Introducing virtual memory

• **Dynamic address translation**
  • Approach #1: Base and bound

```bash
cp -r /afs(ir/class/cs111/lecture-code/lect21 ./
```
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 >= bound, OS triggers an error.
Base and Bound

Process A Virtual Address Space

Process A

Physical Address Space
Approach #1: Base and Bound

- “base” is physical address starting point – corresponds to virtual address 0
- “bound” is one greater than the 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 >= (invalid memory reference)
- Otherwise, add base to virtual address to produce physical address
Example: let’s say process A has \textbf{base} = \textit{1000}, \textbf{bound} = \textit{5000}. What happens if:

- It accesses virtual address \textit{6000}?
- It accesses virtual address \textit{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.
Approach #1: Base and Bound

Example: let’s say process B has base = 6000, bound = 2000. What happens if:

• It accesses virtual address 6000?
• It accesses virtual address 1000?

Respond on PollEv: pollev.com/cs111 or text CS111 to 22333 once to join.
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: 0%
- Accesses 1) physical address 0 and 2) physical address 3000: 0%
- 1) Invalid memory reference and 2) physical address 7000: 0%
- Gets memory errors for both references: 0%
Example: let’s say process B has \textbf{base} = 6000, \textbf{bound} = 2000. What happens if:

• It accesses virtual address \textbf{6000}? Invalid memory reference.
• It accesses virtual address \textbf{1000}? Accesses physical address \textbf{7000}.
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
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
Base and Bound

- Must map entire virtual address space contiguously in physical memory.
- Move it in physical memory by modifying the base.
- The base is pinned to virtual address 0.
- Make more space by adjusting the bound.
Changing the base means changing where in physical memory it’s allocated – there’s no change to the virtual address space.
Changing the base means changing where in physical memory it’s allocated – there’s no change to the virtual address space.
Changing the bound lets the process use higher virtual addresses, which are now mapped to physical addresses.
Changing the bound lets the process use higher virtual addresses, which are now mapped to physical addresses.
Problem: hard to make use of upward space, as we cannot move existing stack/other data up after the program starts.
Recap

• Introducing virtual memory
• Single-tasking
• Goals of sharing memory
• Load-time relocation
• Dynamic address translation
  • Approach #1: Base and Bound

Next time: more about dynamic address translation

Lecture 21 takeaway: Virtual memory is a mechanism that allows multiple processes to simultaneously use system memory. There are two views of memory: virtual and physical. The hardware MMU translates from virtual to physical addresses. Base and bound is one approach to implement virtual memory.