CS111ACE Week 6

Intro to Virtual Memory

https://tinyurl.com/CS111A-is
Icebreaker

- Most famous person you’ve encountered at (within 5 miles) of Stanford
Game Plan

- VM Motivation
- Load Time Relocation
- Dynamic Address Translation
- Paging
Disclaimer:

Virtual memory is *a lot* of material and concepts at once. There’s lots of jargon in this unit, so please stop me if I’m ever unclear!

Also, this stuff is hard, and I’m trying to give a crash course before the long break. It’s okay if not everything sticks, but I’ll try my best!
Review: Computer Memory

● We discussed *main memory* in our very first class together!
Review: Computer Memory

- We discussed *main memory* in our very first class together!

![Diagram of computer memory]

High addresses reserved for OS

I never showed the text on the right before… now it should make more sense!
Review: Computer Memory

- We discussed *main memory* in our very first class together!

How many unique memory addresses are there?

*(You're not expected to know the answer to this right now!)*
Review: Computer Memory

- There’s a problem… DRAM is expensive ($100 per 32GB)
Review: Computer Memory

- There’s a problem… DRAM is expensive ($100 per 32GB)
  - Let’s say that we’re using 4-byte addresses ($2^{32}$ bytes per address space). How many private address spaces can we support at one time with 32GB of RAM ($2^{35}$ bytes)

\[
\frac{2^{35}}{2^{32}} = \frac{2^3}{2^3} = 8
\]

I can represent $2^{32}$ unique #’s
Review: Computer Memory

- There’s a problem… DRAM is expensive ($100 per 32GB)
  - There’s a problem… we use 8-byte addresses in modern computing.
  - How much money would it cost to have enough DRAM for a single 64 bit (8 byte) address space?

\[
\frac{2^{64}}{2^{38}} \text{ unique addresses} = 2^{26} \approx 50 \text{ billion dollars}
\]
Computer Memory Issue #1

- We don’t have enough DRAM to accommodate even a *single* 64-bit process!
  - How is it that we always manage to accommodate multiple at the same time?
Computer Memory Issue #2

- If we only have *one* actual memory but multiple processes, how do we ensure process “isolation?”

Isolation here refers to each process having an independent and private address space (like the sandbox example)

\[ 2^{20} \sim \text{Megabyte} \]
\[ 2^{30} \sim \text{Gigabyte} \]
\[ 2^5 = 32 \quad 2^5 \cdot 2^{30} = 2^{35} = 524288 \]
Computer Memory Issue #2

- If we only have *one* actual memory but multiple processes, how do we ensure process “isolation?”

  Isolation here refers to each process having an independent and private address space (like the sandbox example)

- Similarly, we need to ensure that processes cannot reach into memory that *should* be reserved for the **Operating System**.
The 2 Core Design Principles of Computer Memory

1. Offer enough space to each process in a scalable + performant way.
2. Offer privacy to each process so that one process cannot access / modify the data of another process.
The 2 Core Design Principles of Computer Memory

1. Offer enough space to each process in a scalable + performant way.
2. Offer privacy to each process so that one process cannot access / modify the data of another process.

Another way to think about this: each process should think it’s the only process on the computer -> it has the entire address space at its disposal.
The 2 Core Design Principles of Computer Memory

1. Offer enough space to each process in a scalable + performant way.
2. Offer privacy to each process so that one process cannot access / modify the data of another process.

Another way to think about this: each process should think it’s the only process on the computer -> it has the entire address space at its disposal.

How do we design a system to satisfy these constraints?
Game Plan

- VM Motivation
- Load Time Relocation
- Dynamic Address Translation
- Paging
Quick Review: Internal vs External Fragmentation

**External Fragmentation**
- Poor memory compaction causing reduced effective capacity

**Internal Fragmentation**
- Fragmentation that is within a unit we allocate
Idea #1: Load-Time Relocation

- Main idea: Assign processes a chunk of physical memory to use; need to predict how much memory a process will need.

By “physical” I mean the memory that’s offered to us by our physical RAM (the actual amount of memory we have)
Idea #1: Load-Time Relocation

- Main idea: Assign processes a chunk of physical memory to use; need to predict how much memory a process will need.
- Issues?

- Isolation? Not isolated
Idea #1: Load-Time Relocation

- Main idea: Assign processes a chunk of physical memory to use; need to predict how much memory a process will need.
- Issues?

Need to predict how much memory a process will need

You’ll get **external** fragmentation in your system as you allocate / deallocate process address spaces. (**how could you get internal fragmentation?**)

You can’t move the entire address space mid-execution (space is allocated when the process is loaded)

Processes are only *kind of* isolated (why??)
Idea #1: Load-Time Relocation

- Let’s be honest: Load-Time relocation is kinda garbage.
  - At the very least, it doesn’t appear to behave like our current memory system, which is flexible + fast (ish).
Idea #2: Dynamic Address Translation

- Here’s what we actually do.

We lie
Idea #2: Dynamic Address Translation

- Here’s what we actually do.

We lie map addresses used by our processes to actual addresses in hardware.
Idea #2: Dynamic Address Translation

- Here’s what we actually do.

**Dynamic**: In real-time,
**Address**: when an address is referenced by a program,
**Translation**: the OS translates it to a real address in hardware to perform the read/write.
Idea #2: Dynamic Address Translation

- Here’s what we actually do.
- Isn’t this slow?

**Dynamic**: In real-time,
**Address**: when an address is referenced by a program,
**Translation**: the OS translates it to a real address in hardware to perform the read / write.
Idea #2: Dynamic Address Translation

- Here’s what *we actually do.*
- Isn’t this slow?
  - *Depends on the implementation…*
    - **Dynamic**: *In real-time,*
    - **Address**: *when an address is referenced by a program,*
    - **Translation**: *the OS translates it to a real address in hardware to perform the read / write.*
Idea #2: Dynamic Address Translation

- Here’s what we actually do.
- Isn’t this slow?
  - Depends on the implementation…
    - Dynamic: In real-time,
    - Address: when an address is referenced by a program,
    - Translation: the OS translates it to a real address in hardware to perform the read / write.

- We call addresses that are visible to the process virtual addresses. Addresses in our actual DRAM are called physical addresses.
Idea #2: Dynamic Address Translation

- Here’s what *we actually do*.
- Isn’t this slow?
  - *Depends on the implementation…*
    - **Dynamic**: In real-time,
      **Address**: when an address is referenced by a program,
      **Translation**: the OS translates it to a real address in hardware to perform the read / write.

- We call addresses that are visible to the process **virtual addresses**. Addresses in our actual DRAM are called **physical addresses**.
- There is a piece of hardware whose job it is to make these address translations. It is called the **MMU (Memory Management Unit)**.
Attempt 1: Base + Bound

- **Base and Bound** - Each process is assigned a contiguous region of memory offset by a **Base** value and Limited by a **Bound** value.

Base gives the physical location for effective (virtual) address 0

Bound gives the highest possible virtual address (which you can also think of as the size of the region)

Throw a SIGSEGV if the virtual address is >= the bound

Disclaimer, this is a very coarse example with unrealistic numbers
Attempt 1: Base + Bound

- **Base and Bound** - Each process is assigned a contiguous region of memory offset by a *Base* value and Limited by a *Bound* value.

  - **Pros:** Easy to implement, very fast translation. Also super easy to update records if we move memory. It's also pretty safe (why?)

  - **Cons:** Each program gets a single region, can only grow upwards

Base gives the physical location for *effective (virtual) address* 0

Bound gives the highest possible virtual address (which you can also think of as the size of the region)
Attempt 1: Base + Bound

- **Base and Bound** - Each process is assigned a contiguous region of memory offset by a *Base* value and Limited by a *Bound* value.

When a process is inactive (i.e. not running), we can **swap** its memory region onto disk.

Although this is a little bit slow, it frees up the entire memory region! Because relocation is easy to do, this is a great way to take advantage of idle processes.

Pros:
- Easy to implement, very fast translation. Also super easy to update records if we move memory. It's also pretty safe (why?)

Cons:
- Each program gets a single region, can only grow upwards

Fragmentation (internal)
Attempt 1: Base + Bound

- **Base and Bound** - Each process is assigned a contiguous region of memory offset by a **Base** value and Limited by a **Bound** value.

Base gives the physical location for *effective (virtual) address 0*

Bound gives the highest possible virtual address (which you can also think of as the size of the region)

The OS runs with relocation disabled. Why is that?

Relocation is toggleable via one bit in the processor status register (this is only modifiable in **kernel mode**).
Segments - Break process memory into segments (stack, heap, code, etc…). Segments do not need to be contiguous in memory (unlike Base + Bound)
Segments - Each process has a segment map that tracks mappings for each segment. These maps are per-address space (per-process).

<table>
<thead>
<tr>
<th>Base</th>
<th>Bound</th>
<th>Read-Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>1</td>
</tr>
<tr>
<td>9000</td>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td>18000</td>
<td>2000</td>
<td>0</td>
</tr>
</tbody>
</table>
Attempt 2: Segments

How do you go from an address to an entry in the segmentation map?
Attempt 2: Segments

How do you go from an address to an entry in the segmentation map?
- Option 1 (simple): segment is “implicit” in the instruction.
  - Runtime infers what segment you’re referring to by what is accessed (i.e. am I accessing code? Heap data? Stack data?)
Attempt 2: Segments

How do you go from an address to an entry in the segmentation map?

- Option 1 (simple): segment is “implicit” in the instruction.
  - Runtime infers what segment you’re referring to by what is accessed (i.e. am I accessing code? Heap data? Stack data?)

- Option 2 (scalable): Upper bits of the provided virtual address are turned into an “index” into the segmentation map.
  - For example, reserve the leftmost 1 bit of an address. When that bit is 1, map segment A. When that bit is 0, map to segment B.
Attempt 2: Segments

Segments Pros:
- Segments are flexible
- Better fragmentation
- Easy to share memory

Segments Cons:
Segment map needs to be fairly fixed

Can segmentation support dynamically shared libraries? No
(Segments cannot be added @ runtime)
Game Plan

- VM Motivation
- Load Time Relocation
- Dynamic Address Translation
- Paging
Idea #2: Dynamic Address Translation -> Paging

- Paging is an implementation of Dynamic Address Translation commonly used today (Intel chips do this)
Idea #2: Dynamic Address Translation -> Paging

- Paging is an implementation of Dynamic Address Translation commonly used today (Intel chips do this)
- Let’s introduce it with an example:

```c
int buf[8192]; // That's a big buffer!
int *my_int = new int; // Requires heap space!
```

**High Level Idea:** Break memory into fixed-size regions called **Pages**

**Exercize:** Assuming 4kb page sizes and no pages currently allocated, how many pages need to be allocated for the above code?
Idea #2: Dynamic Address Translation -> Paging

- Paging is an implementation of Dynamic Address Translation commonly used today (Intel chips do this)
- Let’s introduce it with an example:

*High Level Idea: Break memory into fixed-size regions called **Pages***
Idea #2: Dynamic Address Translation -> Paging

- Paging is an implementation of Dynamic Address Translation commonly used today (Intel chips do this)
- Let’s introduce it with an example:

  **High Level Idea:** Break memory into fixed-size regions called **Pages**

- Back in the day, people picked 4kb for a page size. Why not bigger? Why not smaller?
Idea #2: Dynamic Address Translation -> Paging

• Terminology review:
Idea #2: Dynamic Address Translation -> Paging

- Terminology review:
- All addresses visible to the processes are **virtual addresses**. The OS facilitates translation of these addresses into **physical addresses** in DRAM.
Idea #2: Dynamic Address Translation -> Paging

- Terminology review:
- All addresses visible to the processes are **virtual addresses**. The OS facilitates translation of these addresses into **physical addresses in DRAM**
- The **Page Map** is a per-process table that performs the mapping between **virtual addresses** and **physical addresses**.

<table>
<thead>
<tr>
<th>Physical Page #</th>
<th>Read Only</th>
<th>Present</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Upper bits of a physical address</em></td>
<td><em>Read only bit</em></td>
<td><em>Present in RAM?</em></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Example Page Map
Idea #2: Dynamic Address Translation -> Paging

Let’s get a better look at the page map:

<table>
<thead>
<tr>
<th>Physical Page #</th>
<th>Read Only</th>
<th>Present</th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Page Map represents the entire virtual address space for a single process.

The first entry (at index 0) corresponds to the first page of the virtual address space (bytes 0 -> 4095).

Therefore, virtual addresses 0 -> 4095 will map to the 0th index of the page map. This pattern continues for all other pages.

Conceptually, you can think of the entire virtual address space as being “divided up” into 4kb regions.
Idea #2: Dynamic Address Translation

- Let’s step back for a second… What does Paging offer us? Why might it be a good idea?
  - A Page Map can allocate non-contiguous regions of memory to a single process, allowing lots of flexibility (This potentially eliminates external fragmentation!) We call these regions *pages* (typically 4kb of space per page)
Idea #2: Dynamic Address Translation

Let’s step back for a second… What does Paging offer us? Why might it be a good idea?

○ The MMU actually allows you to have separate mappings for every process. This means that it’s **impossible** for Process A to touch a **physical address** assigned to Process B. Now our processes are **isolated**!

Green = Memory Allocated
Orange = Memory not allocated

Stanford University
Idea #2: Dynamic Address Translation

Remember our 2 core design principles? Does paging satisfy these?

The 2 Core Design Principles of Computer Memory

1. Offer enough space to each process in a scalable and performant way.
2. Offer privacy to each process so that one process cannot access or modify the data of another process.
Idea #2: Dynamic Address Translation

Remember our 2 core design principles? Does paging satisfy these?

The 2 Core Design Principles of Computer Memory

1. Offer enough space to each process in a scalable + performant way.
2. Offer privacy to each process so that one process cannot access / modify the data of another process.

Although caching common mappings can improve our performance, we’re still limited by the amount of RAM we have. In high-scale computers, this is 32GB. Is that enough?

How big does our page table need to be? Right now our map needs 1 entry per page (this is LARGE)
Administrativa

- Reminder that we’ll be having a midterm review session today in Thornton208 from 1:30-3:30
  - I will go over questions that people have, and I’d be happy to hand-pick problems from the problem bank that I think are important to ask.
- We’re collecting student testimonials for ACE students! If you’re interested in writing about your experience in an ACE class, please let me know via Slack!
  - Anonymous testimonials are fine!
Disclaimer

Virtual Memory is surprisingly math (memory address math) intensive.

Luckily, there’s quite a finite amount of this math, but it can be very strange to learn (it can feel like powers of 2 are coming out of nowhere)

Please stop me if you have questions, review these slides later on, and feel free to DM me if anything is unclear!
Address Resolution (Translation)

Let’s take a breather and think about addresses for a second:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0
```
Address Resolution (Translation)

Let’s take a breather and think about addresses for a second:

![Possible memory address]

This is a possible memory address. Don’t worry about the difference in the colors!
Address Resolution (Translation)

Let’s take a breather and think about addresses for a second:

This is a possible memory address. Don’t worry about the difference in the colors!

If we look at addresses close to the above address, which bits change? Which bits don’t change?
Address Resolution (Translation)

Let’s take a breather and think about addresses for a second:

This is a possible memory address. Don’t worry about the difference in the colors!

If we look at addresses close to the above address, which bits change? Which bits don’t change?

The lower (right-side) bits will change very frequently. The upper (left) bits will not change unless we jump to an address further away. This will be important soon…
Address Resolution (Translation)

Now let’s see this address with a little more context…
Address Resolution (Translation)

It’s worth thinking about how we actually turn a virtual address into a physical address. This is called **address resolution**.

We do **address resolution** with a data structure called a **Page Table**. It’s basically map from *virtual addresses* to *physical address*.
Address Resolution (Translation)

It’s worth thinking about how we actually turn a virtual address into a physical address. This is called address resolution.

We do address resolution with a data structure called a Page Map. It’s basically map from virtual addresses to physical address.

Notably, the Page Map only has to translate the upper bits of the address (the ones in green here). We’ll learn why soon.
It’s worth thinking about how we actually turn a virtual address into a physical address. This is called **address resolution**.

Recall from the *previous* example that our virtual address space is broken up into 4kb regions. Each 4kb region is represented by a single index \( i \) in the **Page Map**.
Address Resolution (Translation)

It’s worth thinking about how we actually turn a virtual address into a physical address. This is called **address resolution**.

Take a look at the virtual address provided. Can you tell which index in the Page Map our virtual address will map to?

In decimal, our address is 98308/4096 = 24
Address Resolution (Translation)

It’s worth thinking about how we actually turn a virtual address into a physical address. This is called **address resolution**.

How many bytes is this address?
Address Resolution (Translation)

It’s worth thinking about how we actually turn a virtual address into a physical address. This is called **address resolution**.

How many bytes is this address?
32 bits = 4 bytes! (32/8 = 4).
This example uses a 32 bit address space.
Address Resolution (Translation)

It’s worth thinking about how we actually turn a virtual address into a physical address. This is called **address resolution**.

At a high level, for any virtual address:

We reserve the upper (left / green) bits to identify **which physical page** we’re mapping to.

We use the lower (right / orange) bits to locate our **offset** within the page.

Key idea: The number of Page Offset bits tells us **exactly** how many bytes a page is

(ex.) If my virtual address is 1008 bytes into a 4096 byte page, that offset **should not change** regardless of what the **physical address** is!
Address Resolution (Translation)

Using powers of 2 math, we can learn more things about our virtual memory system...
Address Resolution (Translation)

Using powers of 2 math, we can learn more things about our virtual memory system…

<table>
<thead>
<tr>
<th>Virtual Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page number (20 bits)</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 4 4</td>
</tr>
</tbody>
</table>

How big are our pages? \(2^{12}\) bytes

How many pages can a process have? \(2^{20}\) pages
Address Resolution (Translation)

Using powers of 2 math, we can learn more things about our virtual memory system...

How big are our pages? $2^{12}$ unique page offsets -> $2^{12}$ or 4096 bytes

How many pages can a process have? $2^{20}$ unique page numbers -> $2^{20}$ pages (1M)
Next consideration: how big is this address translator data structure (called the Page Map)? This is stored in memory.

Trick: We don’t need to translate the 12 offset bits (why???)

- 4096 page-aligned contiguous virtual addresses will map to the same page.
- How many bits (or bytes, whichever is easier) do we need to represent a page?
- Assume we need to represent page numbers, as well as 1 bit for ReadOnly and 1 bit for present
Address Resolution (Translation)

Next consideration: how big is this address translator data structure (called the page table)? This is stored in memory.

Virtual Address

<table>
<thead>
<tr>
<th>Page number (20 bits)</th>
<th>Page offset (12 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>

Trick: We don’t need to translate the 12 offset bits (why???)

Hint 1: 4096 page-aligned contiguous virtual addresses will map to the same page.
Hint 2: How many bits (or bytes, whichever is easier) do we need to represent a page?
Hint 3: Only focus on the number of page numbers we’ll need to store, nothing else!
Answer: $2^{32} / 2^{12} = 2^{20}$ page numbers. $2^{20} \times 22$ bits = ~2.8MB
Address Resolution (Translation)

For a 32-bit address space, needing 2.6MB of Page Table storage doesn’t sound too bad…

Your turn! Calculate the size of the page table for a 64-bit address space. You may assume a 4kb page size, and keep the same metadata bits.

Hint: This is the 32-bit address breakdown from last slide. Drawing a similar diagram (not exactly the same) for the 64 bit address may be helpful (but not necessary).
Address Resolution (Translation)

For a 32-bit address space, needing 2.6MB of Page Table storage doesn’t sound too bad…

Your turn! Calculate the size of the page table for a 64-bit address space. You may assume a 4kb page size.

Still 12 offset bits, so we can use 50 page number bits. This is $2^{50}$ pages per process :(

Using the same math as last time…

$2^{64} / 2^{12} = 2^{50}$ pages. $2^{50}$ pages * 52 bits per page => 6 Petabytes…

Using a previous calculation, this would cost > 20 million dollars…
Multi Level Page Tables

The problem we have is that the page map must be stored in **contiguous memory** for the address resolution trick to work.

Even if we only create entries on-demand, if the user requests 1 page at the **bottom** of the virtual address space and 1 page at the **top** of the virtual address space, the page map needs to represent space for the **entire** address space, even though we have only allocated 2 pages!
Multi Level Page Tables

The problem we have is that the page map must be stored in contiguous memory for the address resolution trick to work.

Even if we only create entries on-demand, if the user requests 1 page at the bottom of the virtual address space and 1 page at the top of the virtual address space, the page map needs to represent space for the entire address space, even though we have only allocated 2 pages!

The fix relies on a very important fact – processes never will use up the entire address space, so as long as we present a scalable system for storing page map entries, we can get away with not having enough space for the entire map.
Using 4kb pages, we can split the higher 36 bits of the virtual address into 4 9-bit pieces. Each piece indexes into something called a page directory.

**Page Directories** contain $2^{9}$ (512) entries in them. This way, we can say that the pointer to the next page directory is at $PML[9\text{-bit\text{-segment}]}$.

Intel does this in 4 levels. At the lowest level, we have normal Page Maps (or page tables) that contain information about our pages.

Map entries are 8 bytes -> This means that a page directory fits in a single page!