Overview

Let's explore **how to efficiently virtualize memory**. This section will help us answer the following questions:

- **How can we efficiently virtualize memory?**
- **How do we keep control of which memory locations a program can access, and so restrict memory accesses?**
- **How do we do these things efficiently?**

Introduction

We previously explored CPU virtualization using a general approach called **limited direct execution** (or **LDE**) where we let the program run on the hardware most of the time, but have the OS take **control** when needed (like when a process submits a system call or a timer interrupt). The OS makes sure that it maintains control over the hardware by interjecting at those critical points.

Modern operating systems aim for efficiency and control. We will use a similar strategies to virtualize memory and achieve both **efficiency** and **control**.

- **Efficiency** requires hardware assistance, which starts out simple (e.g., a few registers) but grows to be pretty complicated (e.g., TLBs, page-table support, etc.)
- **Control** implies that the OS assures that no program can access memory other than its own. We will need support from the hardware to protect programs from each other and the OS from applications

What does it mean for the OS to maintain control when virtualizing memory?

Address Translation

The VM system needs to be more flexible, allowing programs to use their address spaces anyway they choose, making the system easier to program.

We'll use **hardware-based address translation**, or simply **address translation**, which converts each memory access (e.g., fetch, load, or store) from a virtual to a **physical** address.

The hardware performs address translations on every memory reference to direct them to their correct memory locations.

Hardware can't virtualize memory. It just provides the low-level mechanism.

The OS has to get involved at certain points to set up the hardware for memory translation. The OS has to **manage memory** by:
* Keeping track of which places are free and which are in use, and
* Getting involved only when needed to keep control over memory usage.

Everything we do is to **create the illusion that the program has its own private memory, with its own code and data**. In reality, several programs share memory at the same time when the CPU (or CPUs) shifts between running one program and the next.

Virtualization transforms the ugliness of machine reality into a useful, powerful, and easy-to-use abstraction.

Which of the following is responsible for converting each memory access from a virtual address to a physical address?

Assumptions

For our first tries at virtualizing memory, we will start off by assuming a few things:

1. **We'll assume that the user's address space must be *placed adjacently in physical memory*.**
2. **We'll also assume the size of the address space is *smaller than physical memory*.**
3. **Finally, We'll assume that each address space is *exactly the same size*.**

We will gradually relax these assumptions to obtain a more realistic memory virtualization.

An Example

Let's look at a simple example to **how to implement address translation**.

Assume a process has the address space shown in the graphic to the left.

The short code sequence in the graphic:
* Loads a value from memory,
* Increments it by three,
* Then saves it back into memory.

The code's C-language representation would look like this:

```
void func() {
  int x = 3000; //
  x=x+3; //line of code we are interested in
...
```

This line of code is converted to assembly by the compiler, which may look like this (in x86 assembly):

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

Use `objdump` on Linux or `otool` on a Mac to disassemble it.

This code assumes the address of `x` has been placed in the register `ebx` and:
* Loads the value at that location into the general-purpose register `eax` using the `movl` instruction (for "longword" move).
* On the next instruction, we add 3 to `eax`, and
* On the last instruction, we store `eax` back into memory.

In the figure to the left, the three-instruction code sequence is placed at address 128 (in the code section towards the top), while the value of the variable `x` is located at address $15KB$ (in the stack near the bottom). In the figure, the initial value of `x` is 3000, as seen on the stack.

When these instructions are executed, the process makes the following memory accesses:

1. Get instruction at address 128
2. Execute this instruction (load from address $15KB$)

3. Get instruction at address 132
4. Execute this instruction (no memory reference)
5. Get the instruction from address 135
6. Execute this instruction (store to address $15KB$)

From the program's point of view, its address space starts at address 0 and grows to a maximum of $16KB$. These bounds should apply to any memory references it generates.

The OS, on the other hand, intends to virtualize memory by putting the process somewhere else in physical memory, not necessarily at address 0.

Transparent Relocation

- **What is the best way to relocate this process in memory in a way that is transparent to the process?**
- **How can we create the illusion of a virtual address space starting at** 0 **when the address space is actually located somewhere else?**

The graphic to the left shows an example of what physical memory might look like once this process' address space has been stored.

The OS is using the first slot of physical memory for itself, and the process from the previous example has been shifted to the slot starting at physical memory address $32KB$, as shown in the image.

The other two spaces are free.
* $16KB - 32KB$ and $48KB - 64KB$).

Dynamic (Hardware-based) Relocation: Base & Bounds

The term **base and bounds** or **dynamic relocation** is sometimes used to describe early hardware-based address translation technique.

We'll use both terms interchangeably.

Each CPU will require two hardware registers:

- One is called the **base**
- The other is called the **bounds** (sometimes called a limit register)

This base-and-bounds pair will let us put the address space wherever we want in physical memory while still making sure that the process can only access its own address space

Each program is written and compiled *as if* it were loaded at address 0 in this setup.

When a program begins to run, **the OS determines where it should be loaded in physical memory and sets the base register to that value**.

In the example to the left, the OS decides that the process should be loaded at physical address $32KB$ and sets the **base register** to this value.

When the process generates a memory reference, the processor now **translates** it as follows:

```
virtual address + base = physical address
```

Each **virtual address** generated by the process is added to the contents of the **base register** by the hardware, providing a **physical address** that may be issued to the memory system.

Let's look at what happens when a single instruction is performed.

Consider one instruction from our earlier sequence:

```
128: movl 0x0(%ebx), %eax
```

The **program counter (PC)** is set to 128.

When the hardware needs to fetch this instruction:
1. It first adds the **PC** value to the $32KB$ (32768) **base register** value to get a **physical address** of 32896
1. It then uses the address to retrieve the instruction.
1. The CPU then begins to carry out the instruction.
1. The process then issues a load from **virtual address** $15KB$
1. The processor subsequently takes this address and adds to the **base register** $(32KB)$
1. This results in the final physical address of $47KB$ and the desired contents.

Address translation is this process of **taking a virtual address that a process thinks it is referring and transforming it into a physical address where the data truly lives.**

Because we can relocate address spaces even after the process has started, this is often referred to as **dynamic relocation**.

What is the benefit of using the base-and-bounds technique?

How are virtual memory addresses translated using the base & bounds technique?

Address Translation

What happened to that bounds (limit) register? Isn't this the base *AND* bounds method?

The **bounds register** helps in protection and **are used to ensure that any addresses generated by the process are legal and inside the process' "bounds"**.

If a process generates a virtual address that is greater than the bounds value, the CPU will raise an exception and the process will likely be terminated.

The **memory management unit (MMU)** is the component of the processor that assists with address translation.

As we create more advanced memory-management techniques, we add additional circuitry to the MMU.

Let's look at an example of address translation using base-and-bounds. Assume a process with a $4KB$ (yes, extremely tiny) address space has been loaded at $16KB$.

The following are the results of several address translations:

Virtual Address		Physical Address
0	$\rightarrow$	$16KB$
$1KB$	$\rightarrow$	$17KB$
3000	$\rightarrow$	19384
4400	$\rightarrow$	*Fault (out of bounds)*

Adding the base address to the virtual address (which is actually an offset into the address space) gives us the physical address. If the virtual address is *"too big"* or *"negative"* the result will be a **fault**, which causes an exception.

info

More on Bounds Registers

The base and bounds registers are hardware structures on the chip (one pair per CPU).

Bound registers can also have two definitions.

In the second, it stores the physical address of the address space's end, and the hardware adds the base first, then checks to see if the address is inside boundaries.

How does the bounds register help provide protection for processes?

Hardware Support

Let's summarize the hardware support that we need to make this process efficient.

1. **Privileged (Kernel) Mode**
   - We need this to prevent user mode applications from performing privileged operations
2. **Base & Bounds Registers**
   - We need this pair of registers to support address translation and bounds checks.
3. **The ability to translate virtual addresses and check to see if they are within bounds**
   - We need good circuitry to do translations and check limits.
4. **Privileged instructions to update bounds**
   - The OS has to be able to set these values before allowing a user program to run.
5. **Instruction to register exception handlers**
   - The OS has to be able to tell the hardware what code to run if an exception occurs.
6. **The ability to raise exceptions**
   - When processes try to access privileged instructions or access out of bounds memory, the OS has to be able to raise exceptions.

The graphics to the left and below show a timeline of hardware/OS interaction.

.guides/img/addresstrans5

In the graphic to the left, when a process (Process A) starts, its memory translations are handled by the hardware without any OS intervention. Because Process B is executing a “bad load” (to an unauthorized memory address), the OS must intervene to terminate the process and clean up the mess by releasing B’s memory and deleting B’s entry from the process table.

As shown in the images, we are still using a **limited direct execution** strategy.

In most circumstances, the OS simply configures the hardware and lets the process operate directly on the CPU. It only gets involved when the process misbehaves.

Which of the following is NOT a necessary hardware support for efficient address translation?

Summary

This chapter expanded the concept of **limited direct execution** to virtual memory using the **address translation** mechanism.

- With **address translation**, the OS can **regulate every memory access made by a process**, keeping them within the address space.
- We've also seen **base and bounds** virtualization, or dynamic relocation, that **protects against memory references outside a process's address area**.
- While there is sufficient physical memory for more processes, we are currently limited to putting an address space in a fixed-sized slot, which can cause **internal fragmentation**.

Lab Intro

In this lab, we'll use a simulator, `relocation.py` to show how address translations work in a system with base and bounds registers.
* First, run without the `-c` flag to generate a set of translations and see if you can do the address translations yourself.
* Then, run again with the `-c` flag to check your answers.
##
#

This lab assumes a slightly different address space with a heap and stack at opposing ends. Assume the address space has a code section, a fixed-size (small) stack, and a heap that grows downward, as shown in the Figure below.

In this setup, there is only one way to grow, toward higher areas of the address space.

```
-------------- 0KB
|    Code    |
-------------- 2KB
|   Stack    |
-------------- 4KB
|    Heap    |
|     |      |
|     v      |
-------------- 7KB
|   (free)   |
|     ...    |
```

The bounds register in the figure would be set to $7KB$ to represent the end of the address space. The hardware would raise an exception if any address within the boundaries was referenced.

To run with default flags, type `relocation.py` in the terminal.

```
./relocation.py
```

The output should be similar to the following:

```
Base-and-Bounds register information:

  Base   : 0x00003082 (decimal 12418)
  Limit  : 472

Virtual Address Trace
  VA  0: 0x01ae (decimal:430) -> PA or violation?
  VA  1: 0x0109 (decimal:265) -> PA or violation?
  VA  2: 0x020b (decimal:523) -> PA or violation?
  VA  3: 0x019e (decimal:414) -> PA or violation?
  VA  4: 0x0322 (decimal:802) -> PA or violation?
```

Write down the physical address of each virtual address OR that it is an out-of-bounds address (a segmentation violation). Assume a basic virtual address space of a given size.

It produces random virtual addresses. Determine whether each is in bounds and, if yes, which physical address it maps to. By running with the `-c` flag, we get the results of these translations, whether they are valid or not, and if they are, their physical addresses. All numbers are supplied in hexadecimal for convenience.

```
./relocation.py -c
```

With a base address of 12418 (decimal), 430 is within bounds (less than the limit register of 472) and so it converts to $430 + 12418 = 12848$. Some of the addresses (523, 802) are out of bounds (523, 802).

You can use the `-h` flag for a list of all of the flags and options for this simulator.

Particularly:

- `-a` - Controls the size of the virtual address space
- `-p`- Controls the size of physical memory
- `-n` - The number of virtual address to generate
- `-b`, `-l` - the values of the base and bounds registers for this process

Lab

1. Run the simulation with the seeds `1`, `2`, and `3` and compute whether each virtual address generated by the process is in or out of bounds. If it is in bounds, compute the translation
2. Run the simulator with the following flags:

```
-s 0 -n 10
```

What value does `-l` have to be set to in order to make sure that all generated virtual addresses are within bounds?

3. Run the simulator with the following flags:

```
-s 0 -n 10
```

What is the maximum value that base can be set to so the address space still fits entirely into physical memory?

4. Run some of the same problems above with:

- Larger address spaces `-a`
- Larger physical memories `-p`

As a function of the limits register value, what percentage of randomly generated virtual addresses are valid?