【OS】Day17(3)

【Ch15】Address Translation

In virtualizing memory, we will pursue a similar strategy as virtualizing CPU, attaining both efficiency and control while providing the desired virtualization.(*Each process has its own private memory, where its own code and data reside*)

Control implies that the OS ensures that no application is allowed to access any memory but its own; thus, to protect applications from one another.

The technique we will use is referred to as hardware-based address translation, or just address translation. With address translation, the hardware transforms each memory access(e.g. an instruction fetch, load, or store), changing the virtual address provided by the instruction to a physical address where the desired information is actually located.

The OS must also get involved at key points to set up the hardware so that the correct translation take place; it must thus manage memory, keeping track of which locations are free and which are in use, and judiciously intervening to maintain control over how memory is used.

5.1 Assumptions

- For now, we will assume that the user's address space must be placed contiguously in physical memory.
- We will also assume that the size of the address space is not too big; specifically, that it is less than the size of physical memory.
- Finally, we will also assume that each address space is exactly the same size.

5.2 An Example

Imagine we want to load a value from memory, increments it by three, and then stores the value back in.

Here is the C code:

```
void func() {
  int x = 3000;
  x += 3;
}
```

The corresponding assembly code:

```
movl $0x00(%ebx), %eax ;load 0+ebx(3000) into eax
addl $0x3, %eax ;add 3 to eax register
movl %eax, 0x00(%ebx) ;store eax back to mem
```

Figure 15.1: **A Process And Its Address Space**

When the instruction run, from the perspective of the process, the following memory accesses take place:

- Fetch instruction at address 128
- Execute this instruction(Fetch data from 15KB)
- Fetch instruction at address 132
- Execute this instruction(Add 3 to register eax, no memory reference)
- Fetch instruction at address 135
- Execute this instruction(Store value back to 15KB)

From the program's perspective, its address space starts at address 0 and grow to a maximum of 16 KB; all memory references it generates should be within these bounds.

However, to virtualize memory, the OS want to place the process somewhere else in physical memory, not necessarily at address 0.

Thus, how can we relocate this process in memory in a way that is transparent to the process?

The following figure shows how the memory might look like once the relocation is performed:

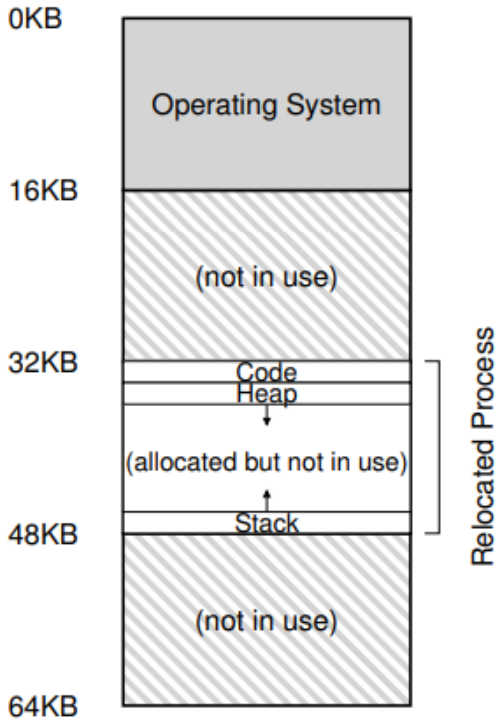

Figure 15.2: **Physical Memory with a Single Relocated Process**

15.3 Dynamic(Hardware-based) Relocation

To gain some understanding of hardware-based address translation, we'll first discuss its first incarnation, referred to as base and bounds; the technique is also referred to as dynamic relocation.

Specifically, we'll need two hardware registers within each CPU: one is called the base register, and the other the bounds(sometimes called a limit register).

This base and bounds registers are allowing us to put address space anywhere we'd like in physical memory, and do so while ensuring that the process can only access its own address space.

In this setup, each program is written and compiles as if it is loaded at address zero. However, when a program starts running, the OS decides where in physical memory it should be loaded and sets the base register to that value.

In the example above, the OS decides to load the process at physical address 32 KB and thus sets the base register to this value.

Now, when any memory reference is generated by the process, it is translated by the processor in the following manner:

$$physical\ memory = virtual\ address + base$$

Each memory reference generated by the process is a virtual address; the hardware in turn adds the contents of the base register to this address and the result is a physical address that can be issued to the memory system.

To understand this better, let's trace through what happens when a single instruction is executed. Specifically, let's look at one instruction from our earlier sequence:

```
128: movl 0x00(%ebx), %eax
```

The program counter(PC) is set to 128; when the hardware needs to fetch this instruction, it first adds the value to the base register value of 32KB(32768) to get a physical address of 32896; the hardware then fetches the instruction from that physical address. Next, the processor begins executing the instruction.

At some point, the process then issues the load from virtual address 15KB, which the processor takes and again adds to the base register(32KB), getting the final physical address of 47KB and thus the desired contents.

Transforming a virtual address into a physical address is exactly the technique we refer to as address translation; that is, the hardware takes a virtual address the process thinks it is referencing and transforms it into a physical address which is where the data actually resides.

Because this relocation of the address happens at runtime, and because we can move address spaces even after the process has started running, the technique is often referred to as dynamic relocation.

The bounds register is to help with protection. Specifically, the processor will first check that the memory reference is within bounds to make sure it is legal; in the simple example above, the bounds register would always be set to 16KB.

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

Sometimes people call the part of the processor that helps with address translation the memory management unit(MMU); as we develop more sophisticated memory-management techniques, we will be adding more circuitry to the MMU.

Bound registers can be defined in one of two ways:

1. In one way, it holds the size of the address space, and thus the hardware checks the virtual address against it first before adding the base
2. In the second way, it holds the physical address of the end of the address space, and thus the hardware first adds the base and then makes sure the address is within bounds.

Example Translations

Let’s take a look at an example. Imagine a process with an address space of size 4KB has been loaded at physical address 16KB. Here are the results of a number of address translations:

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

It is easy to simply add teh base address to the virtual address to get the resulting physical address.