
Address Space Relocation

Our first attempts at virtualizing memory will be very simple, almost laughably so. Go ahead, laugh all you want; pretty soon it will be the OS laughing at you, when things get quite a bit more complex.

Specifically, we will assume for now that the user's address space must be placed *contiguously* in memory. We will also assume, for simplicity, 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*. Don't worry if these assumptions sound unrealistic; we will relax all the assumptions as we go, thus building a real virtualization of memory.

Our goal for now: to be able to realize what we saw in the intro to virtualizing memory. Each process should think it has its own private memory, whereas the reality is that the address space of each process is sharing physical memory with other processes.

10.1 Static Relocation

Our first approach is one that is quite simple and requires no hardware support. It is called **static relocation**, as the OS will perform a one-time change of the addresses within the program

in order to relocate it into a different part of memory.

An Example

Let's look at an example. Imagine there is a process whose address space as indicated in Figure 10.1. What we are going to examine here is a short code sequence that loads a value from memory, increments it by three, and then stores the value back into memory. You can imagine the C-language representation of this code might look like this:

```
void func()
...
    x = x + 3; // this is the line of code we are interested in
...
```

The compiler turns this line of code into assembly, which might look something like this (in x86 assembly):

```
128: movl 0x00000000(%ebx), %eax ;load 0+ebx into eax
132: addl $0x03, %eax           ;add 3 to eax register
135: movl %eax, 0x00000000(%ebx) ;store eax back to mem
```

This code is quite simple; it presumes that the address of `x` has been placed in the register `ebx`, and then loads the value at that address into the general-purpose register `eax` using the `movl` instruction¹. Then, it adds 3 to `eax`, and finally stores the value in `eax` back into memory at that same location.

In Figure 10.1, you can how both the code and data are laid out in the process's address space; the three-instruction code sequence is located at address 128 (in the code section near the top), and the value of the variable `x` at address 15KB (in the stack near the bottom). Note that the initial value of `x` is set to 3307 before this code sequence runs.

When these instructions run, from the perspective of the process, the following memory accesses take place.

¹The "l" at the end indicates it is a longword move



Figure 10.1: A Process And Its Address Space

- Fetch instruction at address 128
- Execute this instruction (a load from address 15KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (a store to address 15KB)

However, the OS wishes to place this process somewhere in physical memory, not necessarily starting at address zero. Thus,

we have the problem: how can we place this process somewhere else in memory in a way that is transparent to the process?

One early solution to this problem is known as **static relocation**. When the user submits the program to the system to be run, the OS loader (the part of the OS that gets a process up and running) *rewrites* the addresses of the process so that they refer to addresses where the process was placed.

For example, if the OS wished to place the process above at the physical address 32KB, it would go through and change all relevant addresses to increment them by 32KB. For example, the three-instruction sequence above would now become:

```
32896: movl 0x00008000(%ebx), %eax
32900: addl $0x03, %eax
32903: movl %eax, 0x00008000(%ebx)
```

Note that not only have the instructions been relocated in this example, but also the load and store from and to memory. Thus, the process is loaded, the addresses rewritten, and the program runs otherwise unmodified at the desired offset in physical memory. Some form of memory sharing and multiprogramming has been achieved!

OS Issues

Note that now we have introduced static relocation (and multiprogramming in general), the OS has a new issue to deal with. Specifically, where should the OS place a new process's address space in physical memory?

Fortunately, given our assumptions that each address space is (a) smaller than the size of physical memory and (b) the same size, this is quite easy for the OS; it can simply view physical memory as an array of slots, and track whether each one is free or in use. When a new process is created, the OS will have to search some kind of data structure to find room for the new address space and then mark it used.

An example can of what physical memory might look like can be found in Figure 10.2. In the figure, you can see the OS

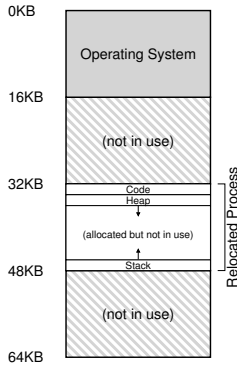


Figure 10.2: Physical Memory with a Single Relocated Process

using the first slot of physical memory for itself, and that it has relocated the process from the example above into the slot starting at physical memory address 32KB. The other two slots are free (16KB-32KB and 48KB-64KB); thus, the OS **free list** should have these two entries in it.

DATA STRUCTURE: THE FREE LIST

The OS must track which parts of free memory are not in use, so as to be able to allocate memory to processes. Many different data structures can of course be used for such a task; the simplest (which we will assume here) is a **free list**, which simply is a list of the ranges of the physical memory which are not currently in use.

Summary

The strengths of this approach are clear: it is *simple*. Unfortunately, there are many negatives. Most importantly, there is *no protection*; thus, a process could generate an address outside of its address space and potentially read or write values in the address spaces of another process. This is a huge problem for building a robust operating environment, and thus static relocation is simply not appropriate in general-purpose settings.

There are other problems with our approach as well. For example, as you can see in Figure 10.2, the relocated process is using physical memory from 32KB to 48KB; however, because the process stack and heap are not too big, all of the space between the two is simply *wasted*. This type of waste is usually called **internal fragmentation**, as the space *inside* the allocated unit is not all used (i.e., is fragmented) and thus wasted. In our current approach, although there might be enough physical memory for more processes, we are currently restricted to placing an address space in a fixed-sized slot and thus internal fragmentation can arise.

Finally, with static relocation, it is quite *difficult to move* an address space once it has been placed in memory [2]. The reason is simple: as the program is running, it may place addresses into registers. To then move the process, not only would the OS have to re-write all of the address in main memory but also in any registers that happen to contain addresses at the moment the process was stopped; knowledge of which registers have addresses in them, to say the least, is difficult to come by.

10.2 Dynamic Relocation

To remedy the weaknesses of static relocation (in particular to add protection), the OS needs some help from our old friend: the hardware. Specifically, we are going to add two registers: one is called the **base** register, and the other the **bounds** (sometimes called a **limit** register). This base/bounds pair is going to

allow us to both relocate a process and do so with protection. Note that this simple approach was used, for example, on the IBM 7090 and the CRAY-1 supercomputer [2,3].

The program is still written as if it is loaded at address zero. However, when it 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 wishes to load the process at physical address 32KB and thus sets the base register to this value.

Interesting things start to happen when the process is running. Now, when any memory reference is generated by the process, it is **translated** by the processor in the following manner:

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

The process generates a **virtual address**; the hardware in turn adds the contents of the base register to this address and the result is a real **physical address** that can be issued to the memory system.

Thus, when the following instruction is fetched and executed:

```
128: movl 0x00008000(%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 the base register value of 32KB (32784) to get a physical address of 32896; the hardware then fetches the instruction from that physical address. 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 something we call **address translation**; that is, we take a virtual address the process thinks it is issuing and turn 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, we call the entire process **dynamic relocation**.

Now you might be asking: what happened to that bounds (limit) register? After all, isn't this supposed to be the base-and-bounds approach? Indeed, it is. And as you might have guessed, the bounds register is there 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 is negative), the processor will signal some kind of fault and the process will likely be terminated. The point of the bounds is thus to make sure that all addresses generated by the process are legal and within the "bounds" of the process.

HARDWARE SUPPORT: DYNAMIC RELOCATION

With dynamic relocation, we can see how a little hardware goes a long way. Namely, a **base** register is used to transform virtual addresses (generated by the program) into physical addresses. A **bounds** (or **limit**) register ensures that such addresses are within the confines of the address space. Together, they combine to provide a simple and efficient virtualization of memory.

We should note that the base and bounds registers are hardware structures kept on the chip and managed by the CPU. Sometimes people call the part of the processor that helps with this type of address translation the **memory management unit (MMU)**; as we develop more sophisticated memory-management techniques, we will be adding more sophistication to the MMU.

One small note about bounds: you can build this in one of two ways. In one way (as above), it would hold the *size* of the address space, and thus the hardware would check the virtual address against it first before adding the base. In the second way, it holds the *physical address* of the end of the address space,

and thus the hardware would first add the base and then make sure the address is within bounds. There is no particular advantage to one way or the other.

An Example

Let's take a look at an example. Imagine a process with an address space of size 4KB (yes, again unrealistically small) has been loaded at physical address 16KB. Here are the results of a number of address translations:

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

As you can see from the example, it is very easy for you to simply add the base address to the virtual address (which can rightly be viewed as an *offset* into the address space) to get the resulting physical address. Only if the virtual address is "too big" will the result be a fault (e.g., 4400 is greater than the 4KB bounds), causing an exception to be raised by the hardware, which the OS will handle likely by killing the process.

OS Issues

From the perspective of the OS, the base and bounds introduces a small change to the context switch code. There is only one base and bounds register in the system, after all, and their values different for each running program, as each program is loaded at a different physical address in memory. Thus, the OS must *save and restore* the base/bounds pair when it switches between processes; they are now part of the PCB (process control block) of the process.

10.3 Summary

We have introduced the basic concept of relocation, which allows the OS to place a process in memory without changing the process; thus, as desired, both static and dynamic relocation are *transparent* to the process that has been relocated (particularly true for dynamic relocation). Both approaches are also *efficient*, in that they are fast; static relocation adds virtually no overhead (except at process load time, when the addresses must be rewritten), and dynamic relocation adds only a little more hardware logic to add a base register to the virtual address and check that it is in bounds (which is quite fast). Finally, dynamic relocation adds real *protection*; by using the base/bounds pair, the OS can ensure that a process cannot generate a reference to an address outside its own address space. Dynamic relocation also enables the OS to easily move an address space in memory; by stopping a process, copying its address space to a new location, and switching the base register, such a move is easily achieved.

Unfortunately, these simple techniques do have their inefficiencies. As we saw above, parceling out memory in fixed-sized units can be wasteful, particularly when the space is not fully utilized. This **internal fragmentation** is particularly a problem when we wish to support large address spaces which often will have huge amounts of space unused between the stack and the heap. Thus, we are going to need to do something more sophisticated, to try to better utilize physical memory and avoid internal fragmentation. Our first attempt will be a slight generalization of dynamic relocation known as **segmentation**, which we will discuss next.

References

[1] "Relocating loader for MS-DOS .EXE executable files"

Kenneth D. A. Pillay

Microprocessors & Microsystems archive

Volume 14, Issue 7 (September 1990) table of contents

Pages: 427-434

[2] "On Dynamic Program Relocation"

W.C. McGee

IBM Systems Journal

Volume 4, Number 3

pages 184-199

This paper is a nice summary of early work on dynamic relocation, as well as some basics on static relocation.

[2] "The CRAY-1 computer system"

Richard M. Russell

Communications of the ACM archive

Volume 21, Issue 1 (January 1978)

Special issue on computer architecture

Pages: 63-72