【OS】Day19

【Ch16】Segmentation

From the figure below, we can see that our base-bounds pair approach is wasting quite a lot of space between the heap and the stack.

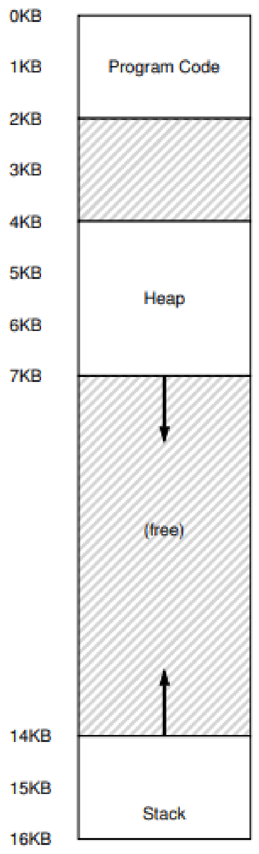


Figure 16.1: **An Address Space (Again)**

It also makes it quite hard to run a program when the entire address space doesn't fit into memory.

Thus, base and bounds is not as flexible as we would like.

16.1 Segmentation: Generalized Base/Bounds

To solve this problem, an idea was born, and it is called segmentation.

The idea is simple: instead of having just one base an bounds pair in our MMU, why not have a base and bounds pair per logical segment of the address space?

A segment is just a contiguous portion of the address space of a particular length, and in our canonical address space, we have three logically-different segments: code, stack, and heap.

What segmentation allows the OS to do is to place each one of those segmentation different parts of physical memory, and thus avoid filling physical memory with unused virtual address space.

Let's look at an example. Assume we want to place the address space from the figure above into physical memory.

With a base and bounds pair per segment, we can place each segment independently in physical memory.

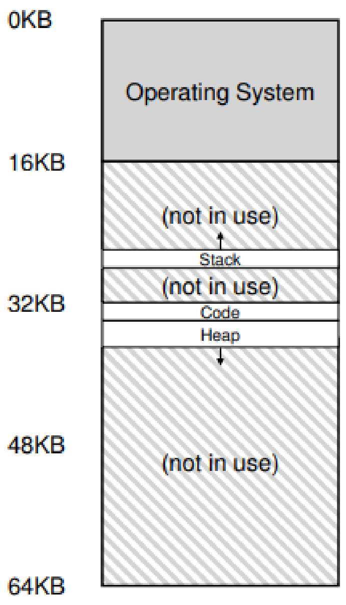
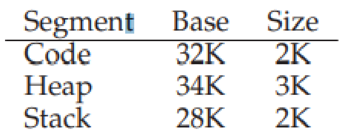


Figure 16.2: **Placing Segments In Physical Memory**

As we can see in the diagram, only used memory is allocated space in physical memory, and thus large address spaces with large amounts of unused address space(which we sometimes call sparse address space) can be accommodated.

The figure below shows the register values for the example above; each bounds register holds the size of a segment.



Segment	Base	Size
Code	32K	2K
Heap	34K	3K
Stack	28K	2K

Figure 16.3: **Segment Register Values**

The size segment here is exactly the same as the bounds register introduced previously; it tells the hardware exactly how many bytes are valid in this segment.

Let's do an example translation, using the address space in Figure 16.1.

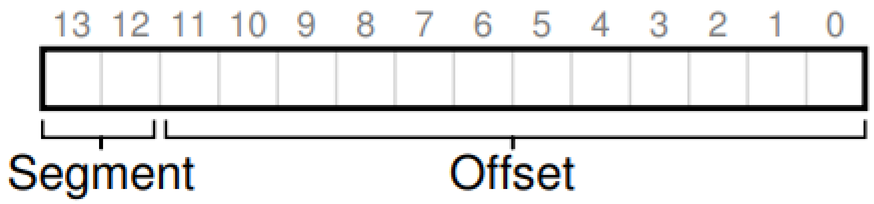
Assuming a reference is made to virtual address 100. When the reference takes place, the hardware will add the base value to *the offset* into this segment (100 in this case, comes from 100 - 0) to arrive at the desired physical address: 100 + 32KB. It will then check that the address is within bounds(100 is less than 2KB).

If we want to refer to virtual address 4200(again refer to Figure 16.1). We cannot just add 4200 to the base as that will create an invalid address. We need to *add the offset* of the virtual address and the heap starting address. In this case, because the heap starts at 4KB, the offset of 4200 is actually 4200 - 4096 = 104. We then take this offset(104) and add it to the base register(34K) and get the desired result: 34920.

16.2 Which Segment Are We Referring To?

The hardware uses segment registers during translation. *How does it know the offset into a segment, and to which segment an address refers?*

One common approach, sometimes referred to as an explicit approach, is to chop up the address space into segments based on the top few bits of the virtual address; this technique was used in the VAX/VMS system.

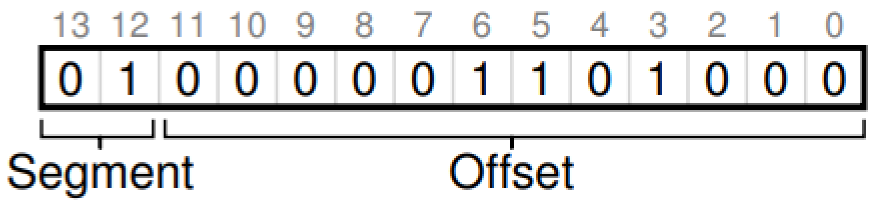


In our example above, we have three segments; thus we need two bits to accomplish our task. If we use the top two bits of our 14-bit virtual address to select the segment, our virtual address looks like the figure above.

In out example, then, if the top two bits are 00, the hardware knows the virtual address is in the code segment, and thus uses the code base and bounds pair to relocate the address to the correct physical location.

If the top two bits are 01, the hardware knows the address is in the heap, and thus uses the heap base and bounds.

Let's take our example heap virtual address from above(4200) and translate it. The virtual address 4200, in binary form, can be seen here.



The top two bits(01) tell the hardware we are referring to the heap segment. The bottom 12 bits are the offset into the segment: 0000 01110 1000, or hex 0x068, or 104 in decimal.

Thus, the hardware simply takes the first two bits to determine which segment register to use, and then takes the next 12 bits as the offset into the segment.

The offset also eases the bounds check: we just need to check if the offset is less than the value in the bounds register.

The hardware would be doing something like this to obtain the desired physical address:

```
//get top 2 bits of 14-bit VA
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT;
Offset = VirtualAddress & OFFSET_MASK;
if(Offset >= Bounds[Segment])
  RaiseException(PROTECTION_FAULT);
else {
  PhyAddr = Base[Segment] + Offset;
  Register = AccessMemory(PhysAddr);
}
```

In our running example, the `SEG_MASK` would be set to 0x3000, `SET_SHIFT` to 12, and `OFFSET_MASK` to 0xFFF.