【OS】Day19(2)

【Ch16】Segmentation

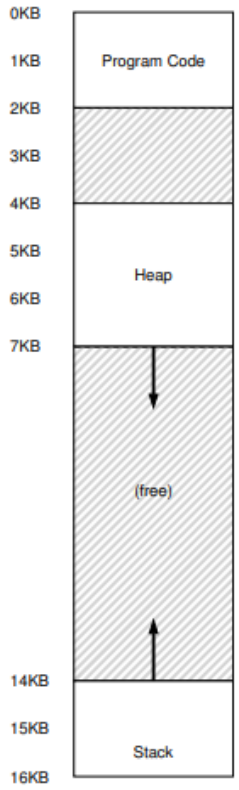

Figure 16.1: **An Address Space (Again)**

16.3 What about the stack?

We have relocated the stack to physical address 28KB in the diagram below, but with one critical difference: it grows backwards(i.e. towards lower addresses).

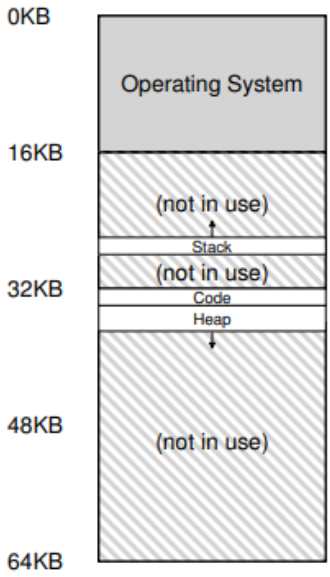

Figure 16.2: **Placing Segments In Physical Memory**

The first thing we need is little extra hardware support. The hardware also needs to know which way the segment grows(a bit, for example, that is set to 1 when the segment grows in the positive direction, and 0 for negative)

Segment	Base	Size (max 4K)	Grows Positive?
$Code_{00}$	32K	2K	1
$Heap_{01}$	34K	3K	1
$Stack_{11}$	28K	2K	0

Figure 16.4: **Segment Registers (With Negative-Growth Support)**

With the hardware understanding that segments can grow in the negative direction, the hardware must now translate such virtual addresses slightly differently.

- In this example, assume we wish to access virtual address 15KB, which should map to physical address 27KB. Our virtual address, in binary form, thus looks like this: 11 1100 0000 0000(hex 0x3C00).

- The hardware uses the top two bits(11) to designate the segment, but then we are left with an offset of 3KB.
- To obtain the correct negative offset, we must subtract the maximum segment size from 3KB: In this example, a segment can be 4KB, and thus the correct negative offset is 3KB minus 4KB which equals -1KB.
- We add the negative offset(-1KB) to the base(28KB) to arrive at the correct physical address: 27KB.
- The bounds check can be calculated by ensuring the absolute value of the negative offset is less than or equal to the segment's current size.

16.4 Support for Sharing

To save memory, sometimes it is useful to share certain memory segments between address spaces. In particular, code sharing is common and still in use in systems today.

To support sharing, we need a little extra support from the hardware, in the form of protection bits.

Basic support adds a few bits per segment, indicating whether or not a program can be read or write a segment, or perhaps execute code that lies within the segment.

- By setting a code segment to read-only, the same code can be shared across multiple processes, without worry of harming isolation
- Each process still thinks that it is accessing its own private memory, the OS is secretly sharing memory which cannot be modified by the process.

Segment	Base	Size (max 4K)	Grows Positive?	Protection
Code_{00}	32K	2K	1	Read-Execute
Heap_{01}	34K	3K	1	Read-Write
Stack_{11}	28K	2K	0	Read-Write

Figure 16.5: **Segment Register Values (with Protection)**

Now the hardware also has to check whether a particular access is permissible. If a user process tries to write to a read-only segment, or execute from a non-executable segment, the hardware should raise an exception, and thus let the OS deal with the offending process.

16.5 Fine-grained vs. Coarse-grained Segmentation

Most of our examples thus far have focused on systems with just a few segments(i.e. code, stack, heap); we can think of this segmentation as coarse-grained, as it chops up the address space into relatively large, coarse chunks.

However, some early systems were more flexible and allowed for address spaces to consist of a large number of smaller segments, referred to as fine-grained segmentation.

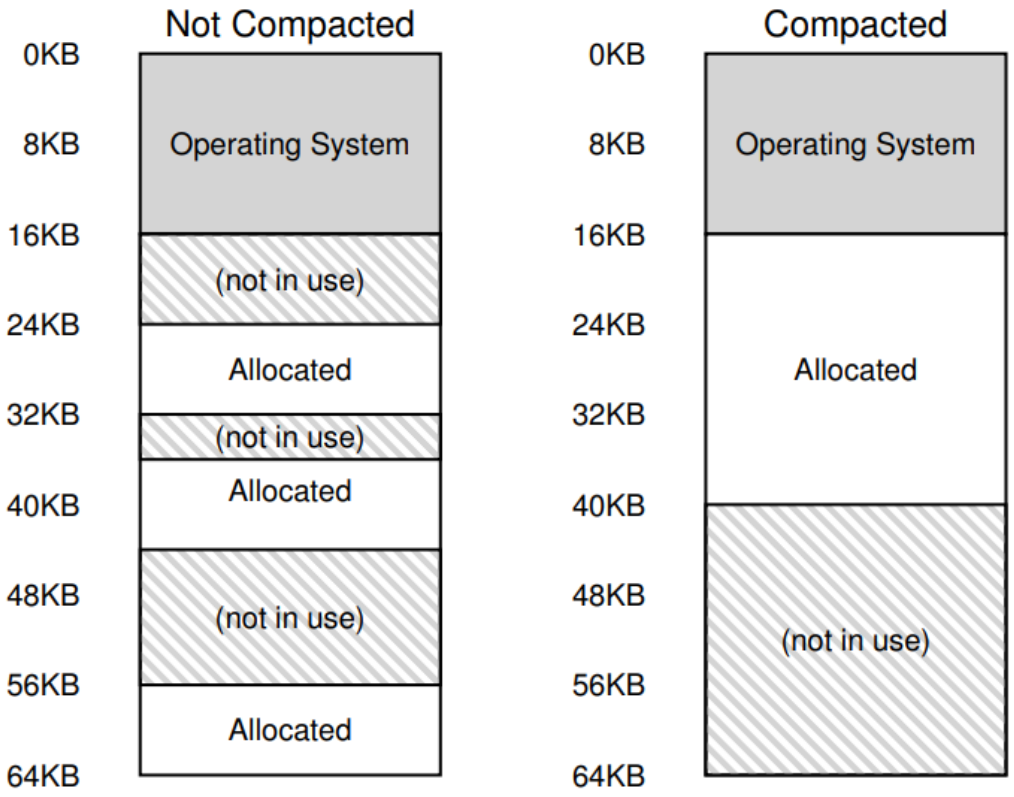

Figure 16.6: **Non-compacted and Compacted Memory**

Supporting many segment requires even further hardware support, with a segment table of some kind stored in memory. Such segment tables usually support the creation of a very large number of segments.

With fine-grained segmentation, the OS could better learn about which segments are in use and which are not and thus utilize main memory more effectively.