**Chapter 16: Segmentation**

How do we support a large address space with (potentially) a lot of free space between the stack and the heap?

**16.1 Segmentation: Generalized Base/Bounds**

To solve this problem, we utilize a method called segmentation. The idea is that instead of having just one base and bounds pair in our MMU, we will 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. Segmentation allows the OS to place each one of those segments in different parts of physical memory. Therefore, this avoids filling physical memory with unused virtual address space.

For example, if we have the address space in Figure 16.1

**Chart, box and whisker chart

Description automatically generatedDiagram

Description automatically generated with medium confidence**

With a base and bounds pair per segment, we can place each segment independently as Figure 16.2. The size of segment here is exactly the same as the bounds register introduced previously.

When we try to refer to an illegal address, we will get **segmentation fault**

**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 approach is **explicit approach**, which is to chop up the address space into segments based on the top few bits. In our above example, since we have 3 segments, we will need two bits to do so:

Diagram

Description automatically generated with low confidence

For example, if the first two bits are 00, then it is code segment (01 for heap and 10 for stack). For example,

A picture containing diagram

Description automatically generated

The offset is 0x068 in hex. Thus, the hardware simply takes the first two bits to indicate the segment (heap) and takes the offset to get the physical address. Some problems are unused segment or limiting the use of the virtual address space

The other approach is **implicit** approach where the hardware determines the segment by noticing how the address was formed. If the address was generated from the program counter, then the address is within the code segment. If the address is based off of the stack or base pointer, it must be in the stack segment; any other address must be in the heap.

**16.3 What about the stack?**

The problem with the stack is that it grows backward. Therefore, the first little extra support is to know which way the segment grows. For example:

Table

Description automatically generated

**16.4 Support for Sharing**

To save memory, it is useful to **share** certain memory segments between address spaces. 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 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. For example,

Text

Description automatically generated

In addition to checking whether a virtual address is within bounds, 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 the examples we discussed were **coarse-grained** as the segments chop up the address space into large chunks. Some early systems were more flexible and allowed for address spaces to consist of a large number of smaller fragments, referred to as **fine-grained** segmentation.

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

**16.6 OS Support**

What should the OS do on a context switch? The segment registers must be saved and restored.

When physical memory quickly becomes full of little holes of free spaces, it is difficult to allocate new segments, or to grow existing ones. We call this **external fragmentation**.

Chart

Description automatically generated with low confidence

The solution would be to compact physical memory by rearranging the existing segments.

Another solution would be using a free-list management algorithm to keep large extents of memory available for allocation.