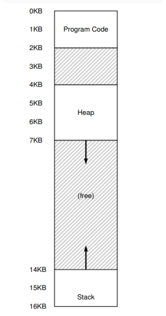
How do we support a large address space with (potentially) a lot of free space between the stack and the heap? Imagine a 32-bit address space (4 GB in size); a typical program will only use megabytes of memory, but still would demand that the entire address space be resident in memory.

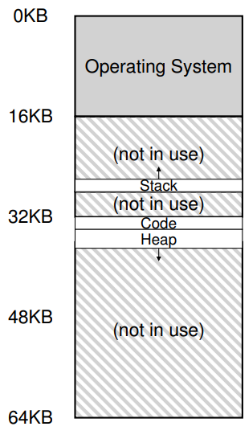
Segmentation: Generalized Base/Bounds

- In computing base and bounds refers to a simple form of virtual memory where access to computer memory is controlled by one or a small number of sets of processor registers called base and bounds registers.
- The idea of segmentation is simple: instead of having just one base and bounds pair in our memory management unit, why not have a base and bounds pair per logical segment of the address space?
- Essentially, with segmentation, only the used portions of the address space (stack, heap, code) are loaded into physical memory while unitized spaces, particularly between segments, don't consume physical memory.
  - What segmentation allows the OS to do is to place each one of those segments in different parts of physical memory, and thus avoid filling physical memory with unused virtual address space.



'igure 16.1: **An Address Space (Again)**

- Let's say from the figure to the right, we want to place the address space into physical memory; with a base and bounds pair, we can place each segment independently in physical memory.
- For example in the figure to the right we see a 64KB physical memory with those three segments in it (and 16KB reserved for the OS).
  - 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 spaces) can be accommodated.



16.2: **Placing Segments In Physical Memory**

- The hardware structure in our MMU required to support segmentation in this case, a set of three base and bounds register pairs.
- Doing an example translation, using the address space in the first image, assume a reference is made to virtual address 100. When the reference takes place (say, on an instruction fetch), the hardware will add the base value to the offset into this segment (100 in this case) to arrive at the desired physical address: 100 + 32KB, or 32868.
  - It will then check that the address is within bounds (100 is less than 2KB), find that it is, and issue the reference to physical memory address 32868.
- Let's look at an address in the heap, virtual address 4200. If we just add the virtual address 4200 to the base of the heap (34KB), we get a physical address of 39016, which is not the correct physical address. What we need to first do is extract the offset into the heap, i.e., which byte(s) in this segment the address refers to. Because the heap starts at virtual address 4KB (4096), the offset of 4200 is actually 4200 minus 4096, or

104. We then take this offset (104) and add it to the base register physical address (34K) to get the desired result: 34920.

- What if we tried to refer to an illegal address?
    - the hardware detects that the address is out of bounds, traps into the OS, likely leading to the termination of the offending process.
    - The term segmentation fault or violation arises from a memory access on a segmented machine to an illegal address.

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;
    - With the explicit approach, the hardware determines which segment an address belongs to by using bits of the address
    - With the implicit approach, the hardware determines the segment by noticing how the address was formed. If, for example, the address was generated from the program counter (i.e., it was an instruction fetch), 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.

What About The Stack?

- The stack is unique as it grows backwards.
- The MMU handles this by knowing the direction of the segment's growth and translating addresses accordingly
- This includes using a negative offset for address translation in the stack segment.

Support for Sharing

- Sharing and Protection:
    - Segmentation facilitates sharing segments (like code sharing) between processes. This is controlled through protection bits in segment registers that dictate read, write and execute permissions.
    - This setup allows multiple processes to share the same physical memory for common segments like code, enhancing memory efficiency.

Fine Grained vs Coarse Grained Segmentation

- Course grained segmentation dives the address space into a few segments (like code, heap, stack)
- Fine grained segmentation involves many small segments, offering more detailed control but required complex hardware support.
    - Supporting many segments requires even further hardware support, with a segment table of some kind stored in memory.

OS Support

- The OS is responsible for managing segment registers, particularly during context switching.
- It also handles segment growth or shrinking, which involves updating the segment's base and bounds.

Fragmentation:

- Segmentation can lead to external fragmentation. The solution to this includes memory compaction and smart allocation strategies to minimize wasted space.
- Fragmentation occurs in memory management when space is wasted due to how memory allocation and deallocation occur.
- External:
    - Occurs when free memory is divided into small non-contigious blocks, making it difficult or impossible to allocate contiguous blocks of memory for new processes or data, even when the total free memory might be sufficient.
    - Caused by allocation and deallocation of blocks of different sizes over time.
    - Can lead to inefficient utilization of memory, and in the inability to allocate memory for a process even though there is enough total free memory, simply because it's not contagious.
- Internal:
    - Occurs when allocated memory may contain unused space. This happens when the allocated memory is slightly larger than the requested memory.
    - Caused when process requests don't align perfectly with the fixed allocation unit size.
    - Common in systems using fixed-size allocation units.
    - Leads to wastage of memory within the allocated space.

Summary

- Beyond just dynamic relocation, segmentation can better support sparse address spaces, by avoiding the huge potential waste of memory between logical segments of the address space.
- It is also fast, as doing the arithmetic segmentation required is easy and well-suited to hardware; the overheads of translation are minimal. A fringe benefit arises too: code sharing. If code is placed within a separate segment, such a segment could potentially be shared across multiple running programs.
- However, as we learned, allocating variable-sized segments in memory leads to some problems that we'd like to overcome. The first is external fragmentation.
- Because segments are variable sized, free memory gets chopped up into odd-sized pieces, and thus satisfying a memory-allocation request can be difficult. One can try to use smart algorithms or periodically compact memory, but the problem is fundamental and hard to avoid.
- The second and perhaps more important problem is that segmentation still isn't flexible enough to support our fully generalized, sparse address space.
    - For example, if we have a large but sparsely-used heap all in one logical segment, the entire heap must still reside in memory in order to be accessed.
    - In other words, if our model of how the address space is being used doesn't exactly match how the underlying segmentation has been designed to support it, segmentation doesn't work very well.