Overview

Let’s explore how to support a larger address space.

This section should help us answer the following questions:

- **How can we support a large address space that could possibly have a lot of free space between the stack and the heap**

Introduction

So far, we've stored each process's complete address space in memory.

The OS can simply move processes around in physical memory using the **base and bounds registers**.

You may have noticed something unique about our address spaces: a large *"free"* area between the stack and the heap.

In the graphic to the left, even though the space is "free", it's still taking up physical memory when we move the whole address space to another spot in physical memory.

In this case, using a base and bounds register for memory virtualization is wasteful. This also makes is difficult to run a program when the whole address space doesn't fit into memory.

Let's further explore how to support a larger address space.

Segmentation: Generalized Base/Bounds

Segmentation was created to address this issue. The idea is to have a base and bounds pair for each logical **segment** of the address space instead of having just one base and bounds pair in the **MMU**.

A **segment** is an uninterrupted piece of the address space of a specific length.

There are three logically-different setment in our contiguous address space:
* Code
* Stack
* Heap

Segmentation lets the OS put each one of those segments into different parts of physical memory and avoid filling the physical memory with unused virtual address space.

Let's say we want to put the address space from our previous graphic into physical memory. If we have a base and bounds pair for every segment, we can put each one *independently* into physical memory.

In the graphic to the left, we see a $64KB$ physical memory with our three segments in it. Huge address spaces with large amounts of **sparse address space** can be accommodated because only used memory is allocated space in physical memory.

Our MMU's hardware structure needs a set of three base and bounds register pairs to handle segmentation. The table shows below the register values for this example. Each bounds register holds the size of a segment.

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

- The code segment is placed at physical address $32KB$ and has a size of $2KB$. The
- The heap segment is placed at $34KB$ and has a size of $3KB$.

The size segment is identical to the bounds register we mentioned previously. It tells the hardware of the number of valid bytes in the segment. This allows the hardware to detect when a software has accessed

data outside of these bounds without permission.

Fill in the blank to complete the statement below.

Segmentation Fault

Let's translate the address space in the graphic to the left.

Assume 100 is the virtual address (which is inside the code segment). When the reference happens (like on an instruction fetch), **the hardware will add the base value to the offset into this segment (**100**) to get the desired physical address**:

$$100 + 32KB = 32868$$

The hardware will then check if the address is within bounds (is $100 < 2KB$), see that is is, and issue the reference to physical memory address 32868

Let's look at virtual address 4200 in the heap. Adding the virtual address 4200 to the heap's base ($34KB$) gives us a physical address of 39016, **which is incorrect**.

The first step is to get the **offset** into the heap, which tells us which byte(s) in this segment the address belongs to. Since the heap begins at virtual address $4KB$ (4096), the 4200 offset is really $4200 - 4096$, or 104. **We then add this offset (**104**) to the physical address of the base register (** $34K$**) to get the desired result:** 34920.

What if we tried to refer to an illegal address past the end of the heap(i.e., a virtual address of $7KB$ or more)?

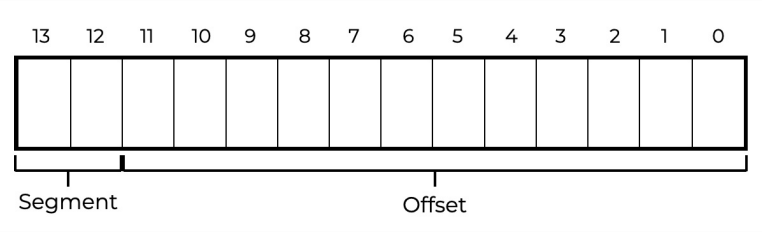
You can probably guess what happens next. The hardware identifies the out-of-bounds address and traps the OS, likely terminating the offending process. This is the origin of the dreaded C term: segmentation violation or **segmentation fault**.

Which of the following occurs when we refer to an address space that lies beyond the end of the heap?

Which Segment Do We Mean?

During translation, the hardware makes use of **segment registers**. How does it determine the offset into a segment, as well as which segment an address relates to? One common approach, known as an **explicit approach**, is to divide the address space into segments based on the first few bits of the virtual address.

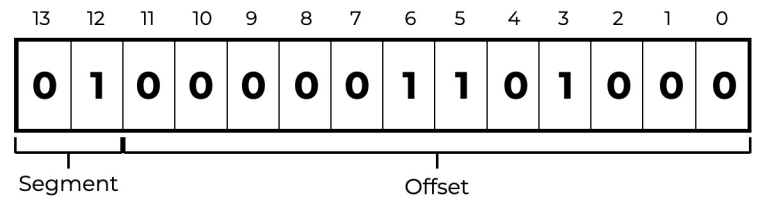
In our previous example, we have three segments, so we only need two bits to complete our assignment. If we pick the segment using the first two bits of our 14-bit virtual address, our virtual address will look like this:



.guides/img/segmentation3

If the top two bits are 00, the hardware understands the virtual address is in the **code** segment and uses the code base and bounds pair to relocate it. If the top two bits are 01, the hardware uses the **heap** base and bounds.

To clarify, let's translate our previous heap virtual address (4200). Here is the virtual address 4200 in binary form:



.guides/img/segmentation4

So, the first two bits (01) indicate the hardware which section we're talking about. The last 12 bits indicate the segment offset: 000001101000, hex $0x068$, decimal 104.

So the hardware uses the first two bits to **select the segment register**, and the next 12 bits to **offset into the segment**. The final physical address is obtained by adding the base register to the offset.

The offset also simplifies the bounds check. If the offset is not less than the bounds, the address is illegal.

Fill in the blank to complete the statement below.

Example

To get the desired physical address, the hardware would do something like the code segment to the left if the base and bounds were arrays (with one entry per segment).

In our continuous example, we can fill in the values for the constants in the code segment to the left.

- `SEG_MASK` would be set to `0x300`
- `SEG_SHIFT` is set to `12`
- `OFFSET_MASK` is set to `0xFFF`

You may have noticed that when we use the top two bits and there are just three segments (code, heap, and stack), **one segment of the address space is left unused**. Some systems put code in the same segment as the heap to fully use the virtual address space (and avoid an unused segment) and use only one bit to decide which segment to use.

Using so many bits to pick a segment also limits the use of virtual address space. Each segment is limited to a maximum size. In our example, $4KB$ (using the top two bits to choose segments implies the $16KB$ address space gets chopped into four pieces, or $4KB$ in this example). A program that wants to expand a segment (like the heap or the stack) beyond that limit is out of luck.

The hardware can also determine the segment an address belongs to. The implicit approach determines the segment by examining the address. If the address came from the program counter (i.e., an instruction fetch), it's in the **code** segment. If it came from the stack or base pointer, it's in the **stack** segment. All others are in the **heap**.

What About the Stack?

So far, we've ignored one crucial aspect of the address space: the **stack**. If we revisit our previous graphic, the stack has been shifted to physical address $28KB$, but with one major difference: it now grows backwards (towards lower addresses). It "starts" at $28KB$ and expands back to $26KB$ in physical memory, corresponding to virtual addresses $16KB$ to $14KB$. Translation has to proceed in a different way.

The first thing we need is some more hardware support. In addition to the base and boundary numbers, the hardware also has to know which way the segment will grow (a bit, for example, that is set to 1 when the segment grows in the positive direction, and 0 for negative). The table below shows our modified view of the hardware tracks:

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

Because the hardware now recognizes that segments can grow in the opposite direction, it has to now translate virtual addresses in a different way. Let's look at an example stack virtual address and translate it.

Say we want to access virtual address $15KB$, which should correspond to physical address $27KB$ in this example. In binary representation, our virtual address looks like this:

```
11 1100 0000 0000 (hex 0x3C00)
```

The first two bits (11) are used by the hardware to designate the segment, but we are left with a $3KB$ offset. To get the right negative offset, subtract the maximum segment size from $3KB$. A segment can be $4KB$ in this case, so the correct negative offset is:

$$3KB - 4KB = -1KB$$

To get the right physical address, we add the negative offset ($-1KB$) to the base ($28KB$). The bounds check is done by confirming that the negative offset's absolute value is less than or equal to the segment's current size (in this case, $2KB$).

Support for Sharing

Sometimes it is useful to **share** memory segments between address spaces in order to save memory. **Code sharing** is a common method that is still used in systems today.

Sharing requires a little extra help from the hardware in the form of **protection bits**. To indicate whether a program can read, write, or possibly execute code contained within a segment requires basic support. While each process believes it is accessing its own private memory, the OS is secretly sharing memory that the process cannot modify, perpetuating the illusion.

Setting a code segment to read-only allows the same code to be shared across several processes without jeopardizing isolation. Although each process believes it is accessing its own private memory, the OS is secretly sharing memory that the process cannot modify, maintaining our illusion.

The table below shows an example of the additional information tracked by the hardware (and OS). The code segment is configured to read and execute, allowing the same physical memory segment to be mapped into multiple virtual address spaces.

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

With **protection bits**, the hardware algorithm would need to change. Aside from checking whether a virtual address is valid, hardware has to also determine whether an access is permitted.

If a user process tries to write to or execute from a read-only segment, the hardware should raise an exception and let the OS handle the offending process.

Fine-grained vs. Coarse-grained Segmentation

The examples we've looked at thus far have been for systems that have only a few segments (code, stack, heap). We could refer to this segmentation as **coarse-grained** because it cuts the address space up into relatively big chunks.

Earlier systems could separate address spaces into a large number of smaller pieces, a technique called **fine-grained** segmentation.

Supporting a large number of segments requires additional hardware, such as a segment database stored in memory. Typically, segment tables like these allow for the generation of a large number of segments, allowing a system to use segments in a more flexible way than we have discussed so far.

OS Support

You should now understand the basics of **segmentation**. As the system operates, pieces of the address space are relocated into physical memory, saving a lot of space compared to using a single base/bounds pair for the whole address space. Specifically, the empty space between the stack and the heap does not need to be allocated in physical memory, allowing us to support larger virtual address spaces per process.

But segmentation presents new challenges for the OS.

1. **What should the OS do on a context switch?** The segment registers must be saved and restored. Each process has its own virtual address space, which the OS must correctly set up before continuing execution.
2. **When segments grow, the OS interacts (or perhaps shrink)**. To allocate an object, a program can use `malloc()`. In some cases, the current heap can satisfy the request, and `malloc()` will find free space for the object and return a pointer to the caller. In others, the heap segment may need to increase.
    - In this case, the memory-allocation library will use a system call to expand the heap (e.g., `sbrk()`). As a result, the OS normally provides more space, updating the segment size register to the new (larger) size, and informing the library of success. The OS may deny the request if no more physical memory is available or if the calling process has too much.
3. Finally, and maybe most importantly, managing physical memory free space. The OS must be able to find physical memory space for new address spaces. Before, we assumed that each address space had the same size, and physical memory could therefore be described as a series of slots for processes. Now we have multiple segments per process, each with a different size.

The main issue is that **physical memory soon fills up with pockets of free space**, making it impossible to assign new segments or expand old ones. We call this **external fragmentation**. We can see an example of this in the figure to the left.

In this case, a process requests a $20KB$ section. In this case, there is $24KB$ free, but not in one piece (rather, in three non-contiguous chunks). So the OS can't handle the $20KB$. If the next so many bytes of physical space are not accessible, **the OS must deny the request**, even if there are free bytes elsewhere in physical memory.

Rearranging existing memory parts could help compact physical memory. For example, the OS may stop all current processes, copy their data to one contiguous region of memory, and update their segment register values to point to the new physical addresses. So the OS lets the next allocation request succeed. However, copying segments is memory-intensive and takes up a lot of processor time. Compaction also makes requests to grow current segments difficult to meet.

It may be simpler to use a **free-list** management strategy that keeps huge amounts of RAM accessible for allocation. Because external fragmentation will always exist, a good algorithm just seeks to minimize it.

When physical memory is so full of empty space that assigning new segments or expanding old ones becomes impossible, which of the following happens?

Summary

Segmentation solves many difficulties and improves memory virtualization.

- It is also fast since the arithmetic segmentation is simple and hardware-friendly. Translation overheads are minor.
- Code sharing. Code stored into a separate section may be shared by several running programs.
- Beyond dynamic relocation, segmentation can help sparse address spaces by reducing the amount of memory wasted across logical address space segments.
- However, as we discovered, allocating variable-sized segments in memory causes certain issues.
- The first is external fragmentation. Because segments vary in size, free memory is divided into odd-sized chunks, making memory allocation challenging.

The problem is fundamental and hard to avoid using smart algorithms or periodically compact memory.

- Second, and perhaps more importantly, segmentation still isn't flexible enough to support our fully generalized, sparse address space.
- If we have a huge but rarely used heap in one logical segment, we must access the entire heap in memory.

The address space model we use does not exactly match how the underlying segmentation was designed to support it, so we need fresh solutions.

Lab Intro

This program shows how address translations work in a system with segmentation. This system uses simple segmentation: a memory address space consists of just two segments; the top bit of the virtual address determines which segment of the address it is located in: 0 for segment 0 (where the code is) and 1 for segment 1 (where the stack lives). Segment 0 grows positively (to higher addresses), while segment 1 grows negatively.

The address space looks similar to the figure below.

```
--------------- virtual address 0
|    seg0     |
|             |
|             |
|-------------|
|             |
|             |
|             |
|             |
|(unallocated)|
|             |
|             |
|             |
|-------------|
|             |
|    seg1     |
|-------------| virtual address max (size of address space)
```

With segmentation, each segment has a base/bounds pair of registers. This problem has two base/bounds pairings. The segment-0 base tells us where the top of segment 0 is physically located in memory, and the limit tells us how big the segment is (or how far it grows in the negative direction).

There are two steps to running the program to test your segmentation knowledge. 1. R un without the “-c” flag to generate a set of translations and check your own translations.

2. Run with a “-c” flag to check your answers.

To run with default flags, type:

```
./segmentation.py
```

or

```
python ./segmentation.py
```

You should see something that looks similar to the following:

```
  ARG seed 0
  ARG address space size 1k
  ARG phys mem size 16k

  Segment register information:

    Segment 0 base  (grows positive) : 0x00001aea (decimal 6890)
    Segment 0 limit                  : 472

    Segment 1 base  (grows negative) : 0x00001254 (decimal 4692)
    Segment 1 limit                  : 450

  Virtual Address Trace
    VA  0: 0x0000020b (decimal:  523) --> PA or segmentation
violation?
    VA  1: 0x0000019e (decimal:  414) --> PA or segmentation
violation?
    VA  2: 0x00000322 (decimal:  802) --> PA or segmentation
violation?
    VA  3: 0x00000136 (decimal:  310) --> PA or segmentation
violation?
    VA  4: 0x000001e8 (decimal:  488) --> PA or segmentation
violation?

  For each virtual address, either write down the physical
address it translates
  to OR write down that it is an out-of-bounds address (a
segmentation
  violation). For this problem, you should assume a simple
address space with
  two segments: the top bit of the virtual address can thus be
used to check
  whether the virtual address is in segment 0 (topbit=0) or
segment 1
  (topbit=1). Note that the base/limit pairs given to you grow
in different
  directions, depending on the segment, i.e., segment 0 grows in
the positive
  direction, whereas segment 1 in the negative.
```

Run the program again with the “-c” flag after computing the translations in the virtual address trace. You should see something similar to the following:

```
  Virtual Address Trace
    VA  0: 0x0000020b (decimal:  523) --> SEGMENTATION VIOLATION
(SEG1)
    VA  1: 0x0000019e (decimal:  414) --> VALID in SEG0:
0x00001c88 (decimal: 7304)
    VA  2: 0x00000322 (decimal:  802) --> VALID in SEG1:
0x00001176 (decimal: 4470)
    VA  3: 0x00000136 (decimal:  310) --> VALID in SEG0:
0x00001c20 (decimal: 7200)
    VA  4: 0x000001e8 (decimal:  488) --> SEGMENTATION VIOLATION
(SEG0)
```

With -c, the program translates the addresses for you, allowing you to test your understanding of how segmentation works.

You can use various settings to create new challenges. The -s or -seed parameter allows you to generate different problems by providing a random seed. Use the same random seed while creating and solving problems.

You can also use settings to experiment with different address spaces and physical memories. For example, to test segmentation in a small system, type:

```
prompt> ./segmentation.py -s 100 -a 16 -p 32
ARG seed 0
ARG address space size 16
ARG phys mem size 32

Segment register information:

  Segment 0 base  (grows positive) : 0x00000018 (decimal 24)
  Segment 0 limit                  : 4

  Segment 1 base  (grows negative) : 0x00000012 (decimal 18)
  Segment 1 limit                  : 5

Virtual Address Trace
  VA  0: 0x0000000c (decimal:   12) --> PA or segmentation
violation?
  VA  1: 0x00000008 (decimal:    8) --> PA or segmentation
violation?
  VA  2: 0x00000001 (decimal:    1) --> PA or segmentation
violation?
  VA  3: 0x00000007 (decimal:    7) --> PA or segmentation
violation?
  VA  4: 0x00000000 (decimal:    0) --> PA or segmentation
violation?
```

This tells the program to create virtual addresses for a 16-byte address space contained within a 32-byte physical memory. The resulting virtual addresses are small (12, 8, 1, 7, and 0). It also chooses tiny base register and limit values, as shown. Use -c to see the results.

This example should also explain each base pair. Segment 0's base, for example, is 24 (decimal) and 4 bytes. These correspond to physical addresses 24, 25, 26, and 27, respectively.

The negative-growing section 1 is a little trickier. So, segment 1's base register is set to physical address 18 and is 5 bytes long. In this scenario, 11, 12, 13, 14, and 15 are valid virtual addresses, and they map to physical addresses 13, 14, 15, 16, and 17.

If it doesn't make sense, reread it. You'll need to understand how this works to solve any of these issues.

You can specify larger quantities by appending "k", "m", or "g" to the -a or -p flag values, as in "kilobytes", "megabytes", and "gigabytes". So, to translate a 1-MB address space into 32-MB physical memory, you might type:

```
./segmentation.py -a 1m -p 32m
```

You can also set the base and limit register values manually using the —b0, —l0, —b1, and —l1 registers.

Lastly, you can run

```
./segmentation.py -h
```

for a full list of flags and options.

Lab

1. Let's start by translating some addresses using a small address space. Here's a set of parameters with a variety of random seeds. Can you translate the addresses?

```
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2
```

2. Let's see if we can decipher this teeny-tiny address space we've created (using the parameters from the question above).
    - In segment 0, what is the highest allowed virtual address?
    - What about the segment 1's lowest legal virtual address?
    - What are the smallest and largest illegal addresses in this address space?
    - How would you test your hypothesis by running `segmentation.py` with the `-A` flag?
3. Imagine we have a 128-byte physical memory with a small 16-byte address space inside.
    - What would you set as the simulator's base and bounds to get the following translation results for the specified address stream:
        - valid, valid, violation,..., violation, valid, valid?

Assuming the following conditions:

```
segmentation.py -a 16 -p 128
    -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
    --b0 ? --l0 ? --b1 ? --l1 ?
```

4. Say we want to create a problem where about 90 of the randomly-generated virtual addresses are valid (not segmentation violations).
    - How would you configure this simulator to to this?
    - Which parameters are important to get this outcome?
5. Can you run the simulator in such a way that there would be no valid virtual addresses?