15

Segmentation

So far we have been putting the entire address space of each process in memory. With the base and bounds registers, the OS can easily relocate processes to different parts of physical memory. However, you might have noticed something interesting about these address spaces of ours: there is a big chunk of "free" space right in the middle, between the stack and the heap.

As you can imagine from Figure 15.1, although the space between the stack and heap is not being used by the process, it is still taking up physical memory when we relocate the entire address space somewhere in physical memory; thus, the simple approach of using a base and bounds register pair to virtualize memory is wasteful. 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. And thus, a problem:

THE CRUX: HOW TO SUPPORT A LARGE ADDRESS SPACE
How do we support a large address space with (potentially) a lot of free space between the stack and the heap? Note that in our examples, with tiny (pretend) address spaces, the waste doesn't seem too bad. Imagine, however, 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.
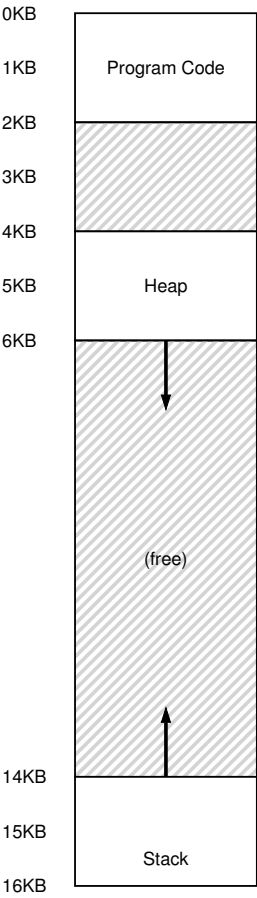
Figure 15.1: An Address Space (Again)

15.1 Segmentation: Generalized Base/Bounds

To solve this problem, an idea was born, and it is called **segmentation**. It is quite an old idea, going at least as far back as the very early 1960's [H61, G62]. The idea is simple: instead of having just one base and 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 segments in different parts of physical memory, and thus avoid filling physical memory with unused virtual address space.
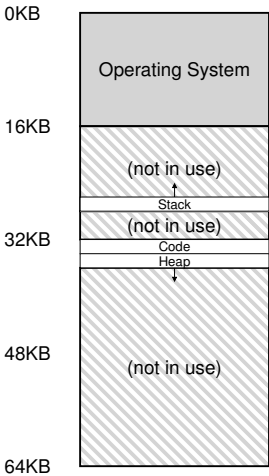

Figure 15.2: Placing Segments In Physical Memory

Let's look at an example. Assume we want to place the address space from Figure 15.1 into physical memory. With a base and bounds pair per segment, we can place each segment *independently* in physical memory. For example, see Figure 15.2; there you will see a 64-KB physical memory with those three segments placed within it (as well as 16KB reserved for the OS).

TIP: BE CAREFUL WITH GENERALIZATION
As you can see, segmentation is just a generalization of dynamic re-
location; instead of a single base and bounds register, we use a few
(or even many). The abstract technique of generalization can thus
be quite useful in systems design, where one good idea can be made
slightly broader and thus solve a larger class of problems. However,
be careful when generalizing; as Lampson warns us "Don't general-
ize; generalizations are generally wrong." [L83]

As you 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 spaces**) can be accommodated.

The hardware structure in our MMU required to support segmen-
tation is just what you'd expect: in this case, a set of three base and
bounds register pairs. Table 15.1 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	2K
Stack	28K	2K

Table 15.1: Segment Register Values

You can see from the table that the code segment is placed at phys-
ical address 32KB and has a size of 2KB and the heap segment is
placed at 34KB and also has a size of 2KB.

Let's do an example translation. Assume that a reference is made
to virtual address 100 (which is in the code segment). When the ref-
erence 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.

Now let's look at an address in the heap, say virtual address 4200.
Note that 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 – 4096 or 104. We then take this offset (104) and add it to the base register physical address (34K or 34816) to get the desired result: 34920.

    What if we tried to refer to an illegal address, such as 7KB which is beyond the end of the heap? You can imagine what will happen: the hardware detects that the address is out of bounds, traps into the OS, likely leading to the termination of the offending process. And now you know the origin of the famous term that all C programmers learn to dread: the **segmentation violation** or **segmentation fault**.

ASIDE: **THE SEGMENTATION FAULT**

The term segmentation fault or violation arises from a memory access on a segmented machine to an illegal address. Humorously, the term persists, even on machines where there is no support for segmentation at all. Or perhaps not so humorous, if you can't figure why your code keeps segfaulting.
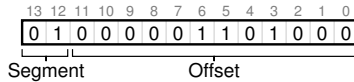
15.2 Which Segment Are We Referring To?

    The hardware uses segment registers to perform translations. But how does it know the offset into a segment? How does it know which segment an address refers to?

    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 [LL82]. 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 a virtual address to select the segment, our virtual address looks like this (assuming a 16KB address space):
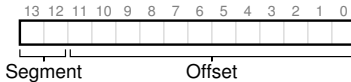
    In our example, then, if the top two bits are 00, the hardware knows the virtual address is in the code segment, and thus uses the

code base and bounds pair to relocate the address to the correct phys-
ical location. If the top two bits are 01, the hardware knows the ad-
dress is in the heap, and thus uses the heap base and bounds. Let's
take our example heap virtual address from above (4200) and trans-
late it, just to make sure this is clear. The virtual address 4200, in
binary form, can be seen here:

As you can see from the picture, the top two bits (01) tell the hard-
ware which *segment* we are referring to. The bottom 12 bits are the
offset into the segment: 0000 0110 1000, or hex 0x068, or 104 in deci-
mal. 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. By adding the base register to the offset, the
hardware arrives at the final physical address. Note the offset eases
the bounds check too: we can simply check if the offset is less than
the bounds; if not, the address is illegal. Thus, if base and bounds
were arrays (with one entry per segment), the hardware would be
doing something like this to obtain the desired physical address:

```
1   // get top 2 bits of 14-bit VA
2   Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3   // now get offset
4   Offset  = VirtualAddress & OFFSET_MASK
5   if (Offset >= Bounds[Segment])
6       RaiseException(PROTECTION_FAULT)
7   else
8       PhysAddr = Base[Segment] + Offset
9       Register = AccessMemory(PhysAddr)
```

In our running example, we can fill in values for the constants
above. Specifically, SEG_MASK would be set to 0x3000, SEG_SHIFT
to 12, and OFFSET_MASK to 0xFFF.

You may also have noticed that when we use the top two bits, and
we only have three segments (code, heap, stack), one segment of the
address space goes unused. Thus, some systems would put the code
in the same segment as the heap and thus use only 1 bit to select
which segment to use [LL82].

There are other ways for the hardware to determine which seg-
ment a particular address is in. In the **implicit** approach, the hard-
ware 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.

15.3 What About The Stack?

Thus far, we've left out one important component of the address space: the stack. The stack has been relocated to physical address 28KB in the diagram above, but with one important difference: *it grows backwards* (in physical memory, it starts at 28KB and grows backwards to 26KB, which correspond to virtual addresses 16KB to 14KB); translation has to proceed differently.

The first thing we need is a little extra hardware support. Instead of just base and bounds values, 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). Our updated view of what the hardware tracks (Table 15.2):

Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0

Table 15.2: 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. Let's take an example stack virtual address and translate it to understand the process.

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 - 4KB which equals -1KB. We simply 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 the segment's size.

15.4   Support for Sharing

As support for segmentation grew, system designers soon realized that they could realize new types of efficiencies with a little more hardware support. Specifically, 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 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; while 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, and thus the illusion is preserved.

An example of the additional information tracked by the hardware (and OS) is shown in Figure 15.3. As you can see, the code segment is set to read and execute, and thus the same physical segment in memory could be mapped into multiple virtual address spaces.

Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

Table 15.3: Segment Register Values (with Protection)

With protection bits, the hardware algorithm described earlier would also have to change. 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 page, or execute from a non-executable page, the hardware should raise an exception, and thus let the OS deal with the offending process.

15.5   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 (e.g., Multics [CV65,DD68]) were much more flexible and allowed for address spaces to be comprised of a large number smaller segments; we refer to this as **fine-grained** segmentation.

Supporting many segments 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, and thus enable a system to use segments in more flexible ways than we have thus far discussed. For example, early machines like the Burroughs B5000 had support for thousands of segments, and expected a compiler to chop code and data into separate segments which the OS and hardware would then support [RK68]. The thinking at the time was that by having fine-grained segments, the OS could better learn about which segments are in use and which are not and thus utilize main memory more effectively.

15.6 OS Support

You now should have a basic idea as to how segmentation works. Pieces of the address space are relocated into physical memory as the system runs, and thus a huge savings of physical memory is achieved relative to our simpler approach with just a single base/bounds pair for the entire address space. Specifically, all the unused space between the stack and the heap need not be allocated in physical memory, allowing us to fit more address spaces into physical memory.

However, segmentation does raise a number of new issues. We'll first describe the new OS issues that must be addressed. The first is an old one: what should the OS do on a context switch? You should have a good guess by now: the segment registers must be saved and restored. Clearly, each process has its own virtual address space, and thus the OS must make sure to set up these registers correctly before letting the process run again.

The second, and more important, issue is managing free space in physical memory. When a new address space is created, the OS has to be able to find space in physical memory for its segments. Previously, we assumed that each address space was the same size, and thus physical memory could be thought of as a bunch of slots where processes would fit in. Now, we have a number of segments per process, and each segment might be a different size.
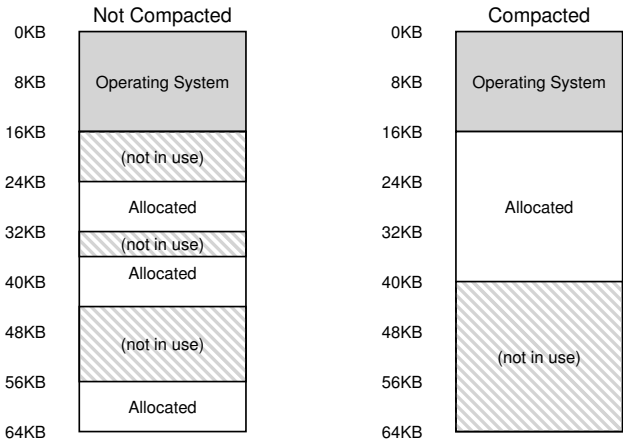
Figure 15.3: Non-compacted and Compacted Memory.

The general problem that arises is that physical memory quickly becomes full of little holes of free space, making it difficult to allocate new segments, or to grow existing ones. We call this problem **external fragmentation** [R69]; see Figure 15.3 (left).

In the example, a process comes along and wishes to allocate a 20KB segment. In that example, there is 24KB free, but not in one contiguous segment (rather, in three non-contiguous chunks). Thus, the OS cannot satisfy the 20KB request.

One solution to this problem would be to **compact** physical memory by rearranging the existing segments. For example, the OS could stop whichever processes are running, copy their data to one contiguous region of memory, change their segment register values to point to the new physical locations, and thus have a large free extent of memory with which to work. By doing so, the OS enables the new allocation request to succeed. However, compaction is expensive, as copying segments is memory-intensive and thus would use a fair amount of processor time. See Figure 15.3 (right) for what a compacted physical memory would look like.

A simpler approach is to use a free-list management algorithm

> TIP: IF 1000 SOLUTIONS EXIST, NO GREAT ONE DOES
> The fact that so many different algorithms exist to try to minimize
> external fragmentation is indicative of a stronger underlying truth:
> there is no one "best" way to solve the problem. Thus, we settle
> for something reasonable and hope it is good enough. The only real
> solution (as we will see in forthcoming chapters) is to avoid the prob-
> lem altogether, by never allocating memory in variable-sized chunks.

that tries to keep large extents of memory available for allocation.
There are literally hundreds of approaches that people have taken,
including classic algorithms like **best-fit** (which keeps a list of free
spaces and returns the one closest in size that satisfies the desired al-
location to the requester), **worst-fit**, **first-fit**, and more complex schemes
like **buddy algorithm** [K68]. An excellent survey by Wilson et al. is a
good place to start if you want to learn more about such algorithms
[W+95], or you can wait until we cover some of the basics ourselves
in a later chapter. Unfortunately, though, no matter how smart the
algorithm, external fragmentation will still exist; thus, a good algo-
rithm simply attempts to minimize it.

15.7 Summary

Segmentation solves a number of problems, and helps us build
a more effective virtualization of memory. Beyond just dynamic re-
location, segmentation can better support sparse address spaces, by
avoiding the huge potential waste of memory between logical seg-
ments of the address space. It is also fast, as doing the arithmetic seg-
mentation requires in hardware 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 seg-
ment 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, as discussed above, is external fragmentation. Because seg-
ments 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 [W+95] 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. We thus need to find some new solutions. Ready to find them?

References

[CV65] "Introduction and Overview of the Multics System"
F. J. Corbato and V. A. Vyssotsky
Fall Joint Computer Conference, 1965
One of five papers presented on Multics at the Fall Joint Computer Conference; oh to be a fly on the wall in that room that day!

[DD68] "Virtual Memory, Processes, and Sharing in Multics"
Robert C. Daley and Jack B. Dennis
Communications of the ACM, Volume 11, Issue 5, May 1968
An early paper on how to perform dynamic linking in Multics, which was way ahead of its time. Dynamic linking finally found its way back into systems about 20 years later, as the large X-windows libraries demanded it. Some say that these large X11 libraries were MIT's revenge for removing support for dynamic linking in early versions of UNIX!

[G62] "Fact Segmentation"
M. N. Greenfield
Proceedings of the SJCC, Volume 21, May 1962
Another early paper on segmentation; so early that it has no references to other work.

[H61] "Program Organization and Record Keeping for Dynamic Storage"
A. W. Holt
Communications of the ACM, Volume 4, Issue 10, October 1961
An incredibly early and difficult to read paper about segmentation and some of its uses.

[I09] "Intel 64 and IA-32 Architectures Software Developer's Manuals"
Intel, 2009
Available: http://www.intel.com/products/processor/manuals
Try reading about segmentation in here (Chapter 3 in Volume 3a); it'll hurt your head, at least a little bit.

[K68] "The Art of Computer Programming: Volume I"
Donald Knuth
Addison-Wesley, 1968
Knuth is famous not only for his early books on the Art of Computer Programming but for his typesetting system TeX which is still a powerhouse typesetting tool used by professionals today, and indeed to typeset this very book. His tomes on algorithms are a great early reference to many of the algorithms that underly computing systems today.

[L83] "Hints for Computer Systems Design"
Butler Lampson
ACM Operating Systems Review, 15:5, October 1983
A treasure-trove of sage advice on how to build systems. Hard to read in one sitting; take it in a little at a time, like a fine wine, or a reference manual.

[LL82] "Virtual Memory Management in the VAX/VMS Operating System"
Henry M. Levy and Peter H. Lipman
IEEE Computer, Volume 15, Number 3 (March 1982)
A classic memory management system, with lots of common sense in its design. We'll study it in more detail in a later chapter.

[RK68] "Dynamic Storage Allocation Systems"
B. Randell and C.J. Kuehner
Communications of the ACM
Volume 11(5), pages 297-306, May 1968
A nice overview of the differences between paging and segmentation, with some historical discussion of various machines.

[R69] "A note on storage fragmentation and program segmentation"
Brian Randell.
Communications of the ACM
Volume 12(7), pages 365-372, July 1969
One of the earliest papers to discuss fragmentation.

[W+95] "Dynamic Storage Allocation: A Survey and Critical Review"
Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles.
In International Workshop on Memory Management
Scotland, United Kingdom, September 1995
A great survey paper on memory allocators.

Homework

This program allows you to see how address translations are performed in a system with segmentation. The segmentation that this system uses is pretty simple: an address space has just *two* segments; further, the top bit of the virtual address generated by the process determines which segment the address is in: 0 for segment 0 (where, say, code and the heap would reside) and 1 for segment 1 (where the stack lives). Segment 0 grows in a positive direction (towards higher addresses), whereas segment 1 grows in the negative direction.

With segmentation, as you might recall, there is a base/limit pair of registers per segment. Thus, in this problem, there are two base/limit pairs. The segment-0 base tells which physical address the *top* of segment 0 has been placed in physical memory and the limit tells how big the segment is; the segment-1 base tells where the *bottom* of segment 1 has been placed in physical memory and the corresponding limit also tells us how big the segment is (or how far it grows in the negative direction).

As before, there are two steps to running the program to test out your understanding of segmentation. First, run without the -c flag to generate a set of translations and see if you can correctly perform the address translations yourself. Then, when done, run with the -c flag to check your answers.

To run with the default flags, type segmentation.py at the command line[1]. The result should be something like 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
```

---

[1]If this doesn't work, try ./segmentation.py, or possibly try python ./segmentation.py. If these don't work, email us!

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

Then, after you have computed the translations in the virtual address trace, run the program again with the −c flag. You will see the following (not including the redundant information):

```
Virtual Address Trace

VA  0: 0x0000020b (decimal: 523) --> SEG VIOLATION (SEG1)
VA  1: 0x0000019e (decimal: 414) --> VALID SEG0: 0x00001c88
VA  2: 0x00000322 (decimal: 802) --> VALID SEG1: 0x00001176
VA  3: 0x00000136 (decimal: 310) --> VALID SEG0: 0x00001c20
VA  4: 0x000001e8 (decimal: 488) --> SEG VIOLATION (SEG0)
```

As you can see, with −c, the program translates the addresses for you, and hence you can check if you understand how a system using segmentation translates addresses.

Of course, there are some parameters you can use to give yourself different problems. One particularly important parameter is the −s or −seed parameter, which lets you generate different problems by passing in a different random seed. Of course, make sure to use the same random seed when you are generating a problem and then solving it.

There are also some parameters you can use to play with different-sized address spaces and physical memories. For example, to experiment with segmentation in a tiny system, you might type:

```
segmentation.py −s 100 −a 16 −p 32
```

which will yield:

```
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 violation?
VA 1: 0x00000008 (decimal:  8) --> PA or violation?
VA 2: 0x00000001 (decimal:  1) --> PA or violation?
VA 3: 0x00000007 (decimal:  7) --> PA or violation?
VA 4: 0x00000000 (decimal:  0) --> PA or violation?
```

The parameters tell the program to use a random seed of 100, and to generate virtual addresses for a 16-byte address space (−a 16) placed somewhere in a 32-byte physical memory (−p 32). As you can see, the resulting virtual addresses are tiny (12, 8, 1, 7, and 0). As you can also see, the program picks tiny base register and limit values, as appropriate. Run with −c to see the answers.

This example should also show you exactly what each base pair means. For example, segment 0's base is set to a physical address of 24 (decimal) and is of size 4 bytes. Thus, *virtual* addresses 0, 1, 2, and 3 are in segment 0 and valid, and map to physical addresses 24, 25, 26, and 27, respectively.

Slightly more tricky is the negative-direction-growing segment 1. In the tiny example above, segment 1's base register is set to physical address 18, with a size of 5 bytes. That means that the *last* five bytes of the virtual address space, in this case 11, 12, 13, 14, and 15, are valid virtual addresses, and that they map to physical addresses 13, 14, 15, 16, and 17, respectively.

Note you can specify bigger values by tacking a k, m, or even g onto the values you pass in with the −a or −p flags, as in "kilobytes", "megabytes", and "gigabytes", respectively. Thus, if you wanted to do some translations with a 1-MB address space set in a 32-MB physical memory, you might type:

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

ARPACI-DUSSEAU

If you want to get even more specific, you can set the base register and limit register values yourself, with the `--b0`, `--l0`, `--b1`, and `--l1` flags.

There is also one more interesting flag: `-A` (or `--addresses`). This flag lets you pass in a comma-separated list of virtual addresses (in decimal) for the program to translate, which can be useful to test your knowledge in different ways. For example, running:

```
segmentation.py -A 0,1,2 -c
```

would run the simulation for virtual addresses 0, 1, and 2. Finally, you can always run:

```
segmentation.py -h
```

Doing so gets you a complete list of flags and options, as usual.

Questions

- First let's use a tiny address space to translate some addresses. Here's a simple set of parameters with a few different 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
```

- Now, let's see if we understand this tiny address space we've constructed (using the parameters from the question above). What is the highest legal virtual address in segment 0? What about the lowest legal virtual address in segment 1? What are the lowest and highest *illegal* addresses in this entire address space? Finally, how would you run segmentation.py with the -A flag to test if you are right?

- Let's say we have a tiny 16-byte address space in a 128-byte physical memory. What base and bounds would you set up so as to get the simulator to generate the following translation results for the specified address stream: valid, valid, violation, ..., violation, valid, valid? Assume the following parameters:

```
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 ?
```

- Assuming we want to generate a problem where roughly 90% of the randomly-generated virtual addresses are valid (i.e., not segmentation violations). How should you configure the simulator to do so? Which parameters are important?

- Can you run the simulator such that no virtual addresses are valid? How?