Ch.16 Simulation hw
Tyler Holmquist

1.

```
tholmquist@tholmquist:~/Downloads/OS_ch16_hw$ python segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0
ARG seed 0
ARG address space size 128
ARG phys mem size 512

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 20

  Segment 1 base  (grows negative) : 0x00000200 (decimal 512)
  Segment 1 limit                  : 20

Virtual Address Trace
  VA  0: 0x0000006c (decimal:  108) --> PA or segmentation violation?
  VA  1: 0x00000061 (decimal:   97) --> PA or segmentation violation?
  VA  2: 0x00000035 (decimal:   53) --> PA or segmentation violation?
  VA  3: 0x00000021 (decimal:   33) --> PA or segmentation violation?
  VA  4: 0x00000041 (decimal:   65) --> 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.
```

Max segment bit = 2^6 = 64

108 - 108 = 1101100, seg = 1, offset = 44-64 = -20, |-20|<=20, **valid in segment 1: 492**

97 - 97 = 1100001, seg = 1, offset = 33-64 = -31, |-31|>20, **Segmentation Fault**

53 - 53 = 0110101, seg = 0, offset = 53, 53>20 , **segmentation fault**

33 - 33= 0100001, seg = 0, offset = 33, 33>20, **segmentation fault**

65 - 65 = 1000001, seg = 1, offset = 1-64 = -63, |-63|>20, **segmentation fault**

```
tholmquist@tholmquist:~/Downloads/OS_ch16_hw$ python segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1
ARG seed 1
ARG address space size 128
ARG phys mem size 512

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 20

  Segment 1 base  (grows negative) : 0x00000200 (decimal 512)
  Segment 1 limit                  : 20

Virtual Address Trace
  VA  0: 0x00000011 (decimal:   17) --> PA or segmentation violation?
  VA  1: 0x0000006c (decimal:  108) --> PA or segmentation violation?
  VA  2: 0x00000061 (decimal:   97) --> PA or segmentation violation?
  VA  3: 0x00000020 (decimal:   32) --> PA or segmentation violation?
  VA  4: 0x0000003f (decimal:   63) --> 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.

tholmquist@tholmquist:~/Downloads/OS_ch16_hw$
```

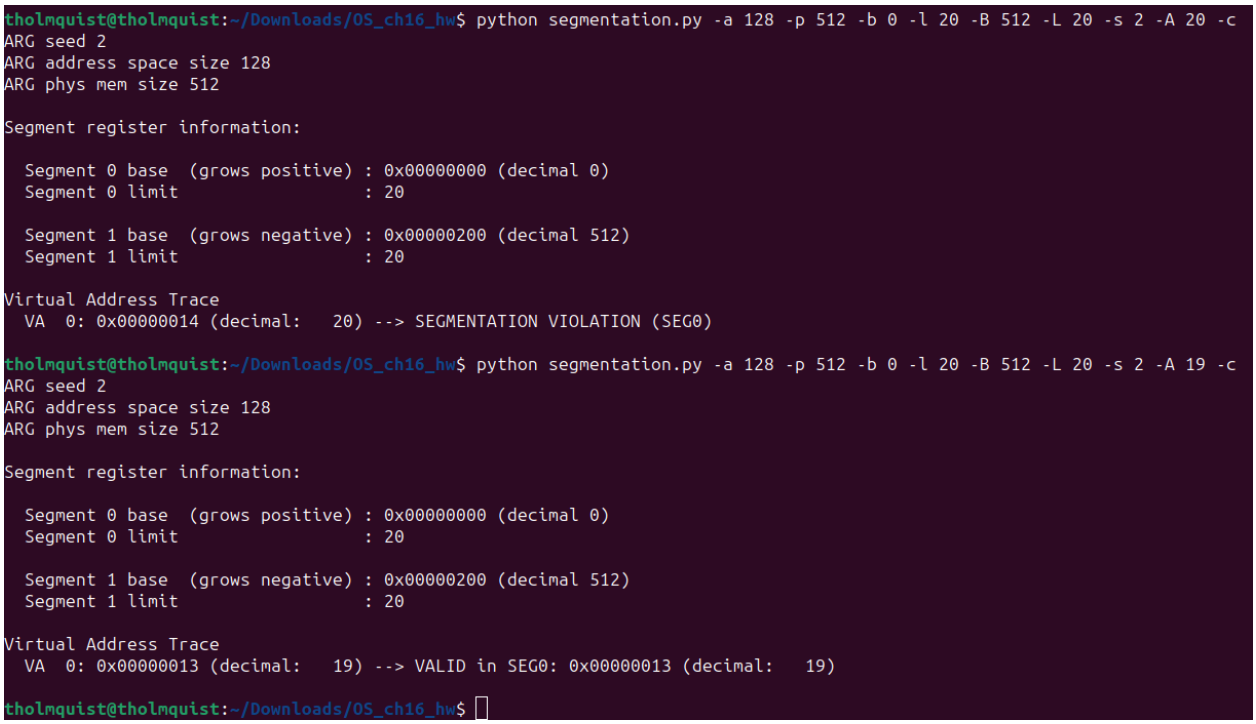
17 - 17=0010001, seg = 0, offset = 17, |17|<=20, **Valid in seg 0, 17**
108 - 108 = 1101100, seg = 1, offset = 44-64 = -20, |-20|<=20, **valid in segment 1: 492**
97 - 97 = 1100001, seg = 1, offset = 33-64 = -31, |-31|>20, **Segmentation Fault**
32 - 32 = 0100000, seg = 0, offset = 32, 32>20, **Segmentation Fault**
63 - 63 = 0111111, seg = 0, offset = 63, 63>20, **Segmentation Fault**

```
tholmquist@tholmquist:~/Downloads/OS_ch16_hw$ python segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2
ARG seed 2
ARG address space size 128
ARG phys mem size 512

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 20

  Segment 1 base  (grows negative) : 0x00000200 (decimal 512)
  Segment 1 limit                  : 20

Virtual Address Trace
  VA  0: 0x0000007a (decimal:  122) --> PA or segmentation violation?
  VA  1: 0x00000079 (decimal:  121) --> PA or segmentation violation?
  VA  2: 0x00000007 (decimal:    7) --> PA or segmentation violation?
  VA  3: 0x0000000a (decimal:   10) --> PA or segmentation violation?
  VA  4: 0x0000006a (decimal:  106) --> 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.

tholmquist@tholmquist:~/Downloads/OS_ch16_hw$
```

122 - 122=1111010, seg=1, offset= 58-64 = -6, |-6| <20, 512-6= 506, **Valid in seg 1, 506**
121 - 121=1111001`, seg=1, offset = 57-64= -7, |-7|<20, 512-7 = 505, **Valid in seg 1, 507**
7 - 7=0000111, seg=0, offset = 7, 7<20, **Valid in seg 0, 7**
10 - 10 = 0001010, seg=0, offset=10, 10<20, **Valid in seg 0, 10**

106 - 106=1101010, seg=1, offset=42-64=-22, |-22|>20, **Segmentation Fault**

2. The highest legal virtual address in segment 0 would be 19 since the limit is 20 and the base is 0 and the lowest legal virtual address in segment 1 would be 492 since the base is 512 and the limit is 20 going backwards.



You can use -c along with a specified VA trace using -A to find the highest and lowest legal virtual address since -c will tell you if they are a valid address or not and -A allows you to specify an address.

```
tholmquist@tholmquist:~/Downloads/OS_ch16_hw$ python segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2 -A 127 -c
ARG seed 2
ARG address space size 128
ARG phys mem size 512

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 20

  Segment 1 base  (grows negative) : 0x00000200 (decimal 512)
  Segment 1 limit                  : 20

Virtual Address Trace
  VA  0: 0x0000007f (decimal:  127) --> VALID in SEG1: 0x000001ff (decimal:  511)

tholmquist@tholmquist:~/Downloads/OS_ch16_hw$ python segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2 -A 128 -c
ARG seed 2
ARG address space size 128
ARG phys mem size 512

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 20

  Segment 1 base  (grows negative) : 0x00000200 (decimal 512)
  Segment 1 limit                  : 20

Virtual Address Trace
Error: virtual address 128 cannot be generated in an address space of size 128
```

3.

```
tholmquist@tholmquist:~/Downloads/OS_ch16_hw$ python segmentation.py -a 16 -p 128 -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 --b0 0 --l0
2 --b1 128 --l1 2 -c
ARG seed 0
ARG address space size 16
ARG phys mem size 128

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 2

  Segment 1 base  (grows negative) : 0x00000080 (decimal 128)
  Segment 1 limit                  : 2

Virtual Address Trace
  VA  0: 0x00000000 (decimal:    0) --> VALID in SEG0: 0x00000000 (decimal:    0)
  VA  1: 0x00000001 (decimal:    1) --> VALID in SEG0: 0x00000001 (decimal:    1)
  VA  2: 0x00000002 (decimal:    2) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x00000003 (decimal:    3) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x00000004 (decimal:    4) --> SEGMENTATION VIOLATION (SEG0)
  VA  5: 0x00000005 (decimal:    5) --> SEGMENTATION VIOLATION (SEG0)
  VA  6: 0x00000006 (decimal:    6) --> SEGMENTATION VIOLATION (SEG0)
  VA  7: 0x00000007 (decimal:    7) --> SEGMENTATION VIOLATION (SEG0)
  VA  8: 0x00000008 (decimal:    8) --> SEGMENTATION VIOLATION (SEG1)
  VA  9: 0x00000009 (decimal:    9) --> SEGMENTATION VIOLATION (SEG1)
  VA 10: 0x0000000a (decimal:   10) --> SEGMENTATION VIOLATION (SEG1)
  VA 11: 0x0000000b (decimal:   11) --> SEGMENTATION VIOLATION (SEG1)
  VA 12: 0x0000000c (decimal:   12) --> SEGMENTATION VIOLATION (SEG1)
  VA 13: 0x0000000d (decimal:   13) --> SEGMENTATION VIOLATION (SEG1)
  VA 14: 0x0000000e (decimal:   14) --> VALID in SEG1: 0x0000007e (decimal:  126)
  VA 15: 0x0000000f (decimal:   15) --> VALID in SEG1: 0x0000007f (decimal:  127)

tholmquist@tholmquist:~/Downloads/OS_ch16_hw$
```

If you set the base0 to 0 and the limit to 2 and then set the base1 to 128 and the limit to 2 that will only allow the first two and the last two segments to pass if the address space size is 16.

4. You would do this by setting the bounds to be 90% of the address space. For example,

```
tholmquist@tholmquist:~/Downloads/OS_ch16_hw$ python segmentation.py -a 20 -p 128 -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19
--b0 0 --l0 9 --b1 130 --l1 9 -c
ARG seed 0
ARG address space size 20
ARG phys mem size 128

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 9

  Segment 1 base  (grows negative) : 0x00000082 (decimal 130)
  Segment 1 limit                  : 9

Virtual Address Trace
  VA  0: 0x00000000 (decimal:    0) --> VALID in SEG0: 0x00000000 (decimal:    0)
  VA  1: 0x00000001 (decimal:    1) --> VALID in SEG0: 0x00000001 (decimal:    1)
  VA  2: 0x00000002 (decimal:    2) --> VALID in SEG0: 0x00000002 (decimal:    2)
  VA  3: 0x00000003 (decimal:    3) --> VALID in SEG0: 0x00000003 (decimal:    3)
  VA  4: 0x00000004 (decimal:    4) --> VALID in SEG0: 0x00000004 (decimal:    4)
  VA  5: 0x00000005 (decimal:    5) --> VALID in SEG0: 0x00000005 (decimal:    5)
  VA  6: 0x00000006 (decimal:    6) --> VALID in SEG0: 0x00000006 (decimal:    6)
  VA  7: 0x00000007 (decimal:    7) --> VALID in SEG0: 0x00000007 (decimal:    7)
  VA  8: 0x00000008 (decimal:    8) --> VALID in SEG0: 0x00000008 (decimal:    8)
  VA  9: 0x00000009 (decimal:    9) --> SEGMENTATION VIOLATION (SEG0)
  VA 10: 0x0000000a (decimal:   10) --> SEGMENTATION VIOLATION (SEG1)
  VA 11: 0x0000000b (decimal:   11) --> VALID in SEG1: 0x00000079 (decimal:  121)
  VA 12: 0x0000000c (decimal:   12) --> VALID in SEG1: 0x0000007a (decimal:  122)
  VA 13: 0x0000000d (decimal:   13) --> VALID in SEG1: 0x0000007b (decimal:  123)
  VA 14: 0x0000000e (decimal:   14) --> VALID in SEG1: 0x0000007c (decimal:  124)
  VA 15: 0x0000000f (decimal:   15) --> VALID in SEG1: 0x0000007d (decimal:  125)
  VA 16: 0x00000010 (decimal:   16) --> VALID in SEG1: 0x0000007e (decimal:  126)
  VA 17: 0x00000011 (decimal:   17) --> VALID in SEG1: 0x0000007f (decimal:  127)
  VA 18: 0x00000012 (decimal:   18) --> VALID in SEG1: 0x00000080 (decimal:  128)
  VA 19: 0x00000013 (decimal:   19) --> VALID in SEG1: 0x00000081 (decimal:  129)

tholmquist@tholmquist:~/Downloads/OS_ch16_hw$
```

In this screenshot I set the address space size to 20 and then set the limit to be 9 for segment 0 and nine for segment 1. This validates 18 virtual addresses which is 90% of 20. The most important parameters are the limit parameters to accomplish this goal.

5. You can set up the simulator so that no virtual addresses are valid by setting both segment 0 and segment 1's limits to be 0. This would make it so that no virtual addresses fall into the valid range and therefore are all segmentation violations.

```
tholmquist@tholmquist:~/Downloads/OS_ch16_hw$ python segmentation.py -a 20 -p 128 -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19
--b0 0 --l0 0 --b1 130 --l1 0 -c
ARG seed 0
ARG address space size 20
ARG phys mem size 128

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 0

  Segment 1 base  (grows negative) : 0x00000082 (decimal 130)
  Segment 1 limit                  : 0

Virtual Address Trace
  VA  0: 0x00000000 (decimal:    0) --> SEGMENTATION VIOLATION (SEG0)
  VA  1: 0x00000001 (decimal:    1) --> SEGMENTATION VIOLATION (SEG0)
  VA  2: 0x00000002 (decimal:    2) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x00000003 (decimal:    3) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x00000004 (decimal:    4) --> SEGMENTATION VIOLATION (SEG0)
  VA  5: 0x00000005 (decimal:    5) --> SEGMENTATION VIOLATION (SEG0)
  VA  6: 0x00000006 (decimal:    6) --> SEGMENTATION VIOLATION (SEG0)
  VA  7: 0x00000007 (decimal:    7) --> SEGMENTATION VIOLATION (SEG0)
  VA  8: 0x00000008 (decimal:    8) --> SEGMENTATION VIOLATION (SEG0)
  VA  9: 0x00000009 (decimal:    9) --> SEGMENTATION VIOLATION (SEG0)
  VA 10: 0x0000000a (decimal:   10) --> SEGMENTATION VIOLATION (SEG1)
  VA 11: 0x0000000b (decimal:   11) --> SEGMENTATION VIOLATION (SEG1)
  VA 12: 0x0000000c (decimal:   12) --> SEGMENTATION VIOLATION (SEG1)
  VA 13: 0x0000000d (decimal:   13) --> SEGMENTATION VIOLATION (SEG1)
  VA 14: 0x0000000e (decimal:   14) --> SEGMENTATION VIOLATION (SEG1)
  VA 15: 0x0000000f (decimal:   15) --> SEGMENTATION VIOLATION (SEG1)
  VA 16: 0x00000010 (decimal:   16) --> SEGMENTATION VIOLATION (SEG1)
  VA 17: 0x00000011 (decimal:   17) --> SEGMENTATION VIOLATION (SEG1)
  VA 18: 0x00000012 (decimal:   18) --> SEGMENTATION VIOLATION (SEG1)
  VA 19: 0x00000013 (decimal:   19) --> SEGMENTATION VIOLATION (SEG1)

tholmquist@tholmquist:~/Downloads/OS_ch16_hw$
```