Operating Systems Assignment 5 (CLO 1, PLO1, Total Marks 25)

This program allows you to see how address translations are performed in a system with segmentation. See the README for details.

Questions

1. [8] 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? You are required to translate "by hand". Infer the segment and the offset, and compute the physical address using the inferred values. You are required to show your work.

   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

2. [1+1+2+3] 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?

3. [4] 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 ?

4. [3] Assume we want to generate a problem where roughly 90% of the randomly-generated virtual addresses are valid (not segmentation violations). How should you configure the simulator to do so? Which parameters are important to getting this outcome?

5. [3] Can you run the simulator such that no virtual addresses are valid? How?

(Solution)

(1)

Terminal Command:
./segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0

Terminal Output:

```
(base) sneaky@sneaky-Lenovo-ideapad-520-15IKB:~/Documents/c/010_HA5$ ./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.
```

Manual Solution:
Address Space Bits = 7 bits

VA 0: 0x0000006C = 0b110_1100 —2's—> 0b001_0100 = 0d20 offset, which is within the limit of 20.
Hence, no violation in segment 1. **Physical address** is **0x000001EC**
VA 1: 0x00000061 = 0b110_0001 —2's—> 0b001_1111 = 0d31 offset, which is out of the limit of 20.
Hence, violation in segment 1. **Physical address** is **0x000001E1**
VA 2: 0x00000035 = 0b011_0101 = 0d53 offset, which is out of the limit of 20.
Hence, violation in segment 0. **Physical address** is **0x00000035**
VA 3: 0x00000021 = 0b010_0001 = 0d33 offset, which is out of the limit of 20.
Hence, violation in segment 0. **Physical address** is **0x00000021**
VA 4: 0x00000041 = 0b100_0001 —2's—> 0b011_1100 = 0d60 offset, which is out of the limit of 20.
Hence, violation in segment 1. **Physical address** is **0x000001C4**

Terminal Output:

```
(base) sneaky@sneaky-Lenovo-ideapad-520-15IKB:~/Documents/c/010_HA5$ ./segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0 -c
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) --> VALID in SEG1: 0x000001ec (decimal:  492)
  VA  1: 0x00000061 (decimal:   97) --> SEGMENTATION VIOLATION (SEG1)
  VA  2: 0x00000035 (decimal:   53) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x00000021 (decimal:   33) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x00000041 (decimal:   65) --> SEGMENTATION VIOLATION (SEG1)
```

Terminal Command:
./segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1

Terminal Output:

```
(base) sneaky@sneaky-Lenovo-ideapad-520-15IKB:~/Documents/c/010_HA5$ ./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.
```

Manual Solution:
Address Space Bits = 7 bits

VA 0: 0x00000011 = 0b001_0001 = 0d17 offset, which is within the limit of 20.
Hence, no violation in segment 0. **Physical address** is **0x00000011**
VA 1: 0x0000006C = 0b110_1100 —2's—> 0b001_0100 = 0d20 offset, which is within the limit of 20.
Hence, no violation in segment 1. **Physical address** is **0x000001EC**
VA 2: 0x00000061 = 0b110_0001 —2's—> 0b001_1111 = 0d31 offset, which is out of the limit of 20.
Hence, violation in segment 1. **Physical address** is **0x000001E1**
VA 3: 0x00000020 = 0b010_0001 = 0d32 offset, which is out of the limit of 20.
Hence, violation in segment 0. **Physical address** is **0x00000020**
VA 4: 0x0000003F = 0b011_1111 = 0d63 offset, which is out of the limit of 20.
Hence, violation in segment 0. **Physical address** is **0x0000003F**

Terminal Output:

```
(base) sneaky@sneaky-Lenovo-ideapad-520-15IKB:~/Documents/c/010_HA5$ ./segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1 -c
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) --> VALID in SEG0: 0x00000011 (decimal:   17)
  VA  1: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492)
  VA  2: 0x00000061 (decimal:   97) --> SEGMENTATION VIOLATION (SEG1)
  VA  3: 0x00000020 (decimal:   32) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x0000003f (decimal:   63) --> SEGMENTATION VIOLATION (SEG0)
```

(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?

Highest legal virtual address in segment 0: 0x00000000 + 0d20 - 1 = 0x00000013 = 0d19
Lowest legal virtual address in segment 1: 0x00000080 - 0d20 = 0x0000006C = 0d108
Highest legal virtual address in the entire address space: 0x00000015 = 0d20
Lowest legal virtual address in the entire address space: 0x000001ED = 0d107

Terminal Command: ./segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -A 19,20,107,108 -c
Terminal Output:

```
(base) sneaky@sneaky-Lenovo-ideapad-520-15IKB:~/Documents/c/010_HA5$  ./segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -A 19,20,107,108 -c
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: 0x00000013 (decimal:   19) --> VALID in SEG0: 0x00000013 (decimal:   19)
  VA  1: 0x00000014 (decimal:   20) --> SEGMENTATION VIOLATION (SEG0)
  VA  2: 0x0000006b (decimal:  107) --> SEGMENTATION VIOLATION (SEG1)
  VA  3: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492)
```

(3)

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 ?

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

Terminal Output:

```
(base) sneaky@sneaky-Lenovo-ideapad-520-15IKB:~/Documents/c/010_HA5$ ./segmentation.py -a 16 -p 128 --b0 0 --l0 2 --b1 14 --l1 2 -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 -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) : 0x0000000e (decimal 14)
  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: 0x0000000c (decimal:   12)
  VA 15: 0x0000000f (decimal:   15) --> VALID in SEG1: 0x0000000d (decimal:   13)
```

(4)

To solve this problem, I wrote a script with the help of chatgpt that takes the same parameters as segmentation.py that helps my script to compute valid and invalid virtual address boundaries. Then there are 3 more command arguments -n tells the number of addresses you want to generate, -P tells the probability of the address to be valid and -g tells whether you want to generate addresses within segment 0(assuming positively growing) or segment 1(negatively growing).

(Case 50% Valid)

Terminal Command:

python address_generator.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -g 1 -n 10 **-P 50** > virtualAddressGenerated.txt

cat virtualAddressGenerated.txt
./segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -c -A "$(cat virtualAddressGenerated.txt)"

Terminal Output:

```
(base) sneaky@sneaky-Lenovo-ideapad-520-15IKB:~/Documents/c/010_HA5$ python address_generator.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -g 1 -n 10 -P 50 > virtualAddressGenerated.txt
(base) sneaky@sneaky-Lenovo-ideapad-520-15IKB:~/Documents/c/010_HA5$ cat virtualAddressGenerated.txt
116,124,71,106,115,26,114,30,28,121
(base) sneaky@sneaky-Lenovo-ideapad-520-15IKB:~/Documents/c/010_HA5$ ./segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -c -A "$(cat virtualAddressGenerated.txt)"
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: 0x00000074 (decimal:  116) --> VALID in SEG1: 0x000001f4 (decimal:  500)
  VA  1: 0x0000007c (decimal:  124) --> VALID in SEG1: 0x000001fc (decimal:  508)
  VA  2: 0x00000047 (decimal:   71) --> SEGMENTATION VIOLATION (SEG1)
  VA  3: 0x0000006a (decimal:  106) --> SEGMENTATION VIOLATION (SEG1)
  VA  4: 0x00000073 (decimal:  115) --> VALID in SEG1: 0x000001f3 (decimal:  499)
  VA  5: 0x0000001a (decimal:   26) --> SEGMENTATION VIOLATION (SEG0)
  VA  6: 0x00000072 (decimal:  114) --> VALID in SEG1: 0x000001f2 (decimal:  498)
  VA  7: 0x0000001e (decimal:   30) --> SEGMENTATION VIOLATION (SEG0)
  VA  8: 0x0000001c (decimal:   28) --> SEGMENTATION VIOLATION (SEG0)
  VA  9: 0x00000079 (decimal:  121) --> VALID in SEG1: 0x000001f9 (decimal:  505)
```

We can see from running the script that 50% of the virtual addresses are not giving segmentation violations.

(Case 90% Valid)

Terminal Command:

python address_generator.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -g 0 -n 10 **-P 90** > virtualAddressGenerated.txt

cat virtualAddressGenerated.txt
./segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -c -A "$(cat virtualAddressGenerated.txt)"

Terminal Output:

```
(base) sneaky@sneaky-Lenovo-ideapad-520-15IKB:~/Documents/c/010_HA5$ python address_generator.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -g 0 -n 10 -P 90 > virtualAddressGenerated.txt
(base) sneaky@sneaky-Lenovo-ideapad-520-15IKB:~/Documents/c/010_HA5$ cat virtualAddressGenerated.txt
19,7,28,18,14,5,8,5,3,11
(base) sneaky@sneaky-Lenovo-ideapad-520-15IKB:~/Documents/c/010_HA5$ ./segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -c -A "$(cat virtualAddressGenerated.txt)"
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: 0x00000013 (decimal:   19) --> VALID in SEG0: 0x00000013 (decimal:   19)
  VA  1: 0x00000007 (decimal:    7) --> VALID in SEG0: 0x00000007 (decimal:    7)
  VA  2: 0x0000001c (decimal:   28) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x00000012 (decimal:   18) --> VALID in SEG0: 0x00000012 (decimal:   18)
  VA  4: 0x0000000e (decimal:   14) --> VALID in SEG0: 0x0000000e (decimal:   14)
  VA  5: 0x00000005 (decimal:    5) --> VALID in SEG0: 0x00000005 (decimal:    5)
  VA  6: 0x00000008 (decimal:    8) --> VALID in SEG0: 0x00000008 (decimal:    8)
  VA  7: 0x00000005 (decimal:    5) --> VALID in SEG0: 0x00000005 (decimal:    5)
  VA  8: 0x00000003 (decimal:    3) --> VALID in SEG0: 0x00000003 (decimal:    3)
  VA  9: 0x0000000b (decimal:   11) --> VALID in SEG0: 0x0000000b (decimal:   11)
```

We can see from running the script that 90% of the virtual addresses are not giving segmentation violations.

(5)

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

Terminal Command:

python address_generator.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -g 0 -n 10 **-P 0** > virtualAddressGenerated.txt

cat virtualAddressGenerated.txt
./segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -c -A "$(cat virtualAddressGenerated.txt)"

Terminal Output:

```
(base) sneaky@sneaky-Lenovo-ideapad-520-15IKB:~/Documents/c/010_HA5$ python address_generator.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -g 0 -n 10 -P 0 > virtualAddressGenerated.txt
(base) sneaky@sneaky-Lenovo-ideapad-520-15IKB:~/Documents/c/010_HA5$ cat virtualAddressGenerated.txt
91,81,48,40,89,85,27,34,69,92
(base) sneaky@sneaky-Lenovo-ideapad-520-15IKB:~/Documents/c/010_HA5$ ./segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -c -A "$(cat virtualAddressGenerated.txt)"
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: 0x0000005b (decimal:   91) --> SEGMENTATION VIOLATION (SEG1)
  VA  1: 0x00000051 (decimal:   81) --> SEGMENTATION VIOLATION (SEG1)
  VA  2: 0x00000030 (decimal:   48) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x00000028 (decimal:   40) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x00000059 (decimal:   89) --> SEGMENTATION VIOLATION (SEG1)
  VA  5: 0x00000055 (decimal:   85) --> SEGMENTATION VIOLATION (SEG1)
  VA  6: 0x0000001b (decimal:   27) --> SEGMENTATION VIOLATION (SEG0)
  VA  7: 0x00000022 (decimal:   34) --> SEGMENTATION VIOLATION (SEG0)
  VA  8: 0x00000045 (decimal:   69) --> SEGMENTATION VIOLATION (SEG1)
  VA  9: 0x0000005c (decimal:   92) --> SEGMENTATION VIOLATION (SEG1)
```