Operating Systems Assignment 7

-210010032, Nithin Sabu

1.1 Run with seeds 1, 2 and 3:

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 relocation.py -s 1 -c

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

Base-and-Bounds register information:

  Base   : 0x0000363c (decimal 13884)
  Limit  : 290

Virtual Address Trace
  VA  0: 0x0000030e (decimal:  782) --> SEGMENTATION VIOLATION
  VA  1: 0x00000105 (decimal:  261) --> VALID: 0x00003741 (decimal: 14145)
  VA  2: 0x000001fb (decimal:  507) --> SEGMENTATION VIOLATION
  VA  3: 0x000001cc (decimal:  460) --> SEGMENTATION VIOLATION
  VA  4: 0x0000029b (decimal:  667) --> SEGMENTATION VIOLATION
```

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 relocation.py -s 2 -c

ARG seed 2
ARG address space size 1k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x00003ca9 (decimal 15529)
  Limit  : 500

Virtual Address Trace
  VA  0: 0x00000039 (decimal:   57) --> VALID: 0x00003ce2 (decimal: 15586)
  VA  1: 0x00000056 (decimal:   86) --> VALID: 0x00003cff (decimal: 15615)
  VA  2: 0x00000357 (decimal:  855) --> SEGMENTATION VIOLATION
  VA  3: 0x000002f1 (decimal:  753) --> SEGMENTATION VIOLATION
  VA  4: 0x000002ad (decimal:  685) --> SEGMENTATION VIOLATION
```

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 relocation.py -s 3 -c

ARG seed 3
ARG address space size 1k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x000022d4 (decimal 8916)
  Limit  : 316

Virtual Address Trace
  VA  0: 0x0000017a (decimal:  378) --> SEGMENTATION VIOLATION
  VA  1: 0x0000026a (decimal:  618) --> SEGMENTATION VIOLATION
  VA  2: 0x00000280 (decimal:  640) --> SEGMENTATION VIOLATION
  VA  3: 0x00000043 (decimal:   67) --> VALID: 0x00002317 (decimal: 8983)
  VA  4: 0x0000000d (decimal:   13) --> VALID: 0x000022e1 (decimal: 8929)
```

1.2 Run with flags -s 0 and -n 10

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 relocation.py -s 0 -n 10 -c

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

Base-and-Bounds register information:

  Base   : 0x00003082 (decimal 12418)
  Limit  : 472

Virtual Address Trace
  VA  0: 0x000001ae (decimal:  430) --> VALID: 0x00003230 (decimal: 12848)
  VA  1: 0x00000109 (decimal:  265) --> VALID: 0x0000318b (decimal: 12683)
  VA  2: 0x0000020b (decimal:  523) --> SEGMENTATION VIOLATION
  VA  3: 0x0000019e (decimal:  414) --> VALID: 0x00003220 (decimal: 12832)
  VA  4: 0x00000322 (decimal:  802) --> SEGMENTATION VIOLATION
  VA  5: 0x00000136 (decimal:  310) --> VALID: 0x000031b8 (decimal: 12728)
  VA  6: 0x000001e8 (decimal:  488) --> SEGMENTATION VIOLATION
  VA  7: 0x00000255 (decimal:  597) --> SEGMENTATION VIOLATION
  VA  8: 0x000003a1 (decimal:  929) --> SEGMENTATION VIOLATION
  VA  9: 0x00000204 (decimal:  516) --> SEGMENTATION VIOLATION
```

To get all virtual addresses within bounds, the ideal value of bounds register would be equal to the address space = 1k here. But since the seed generates the same values every time, we can say for **this specific case**, bound register must be greater than 929.

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 relocation.py -s 0 -n 10 -c -l 930

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

Base-and-Bounds register information:

  Base   : 0x0000360b (decimal 13835)
  Limit  : 930

Virtual Address Trace
  VA  0: 0x00000308 (decimal:  776) --> VALID: 0x00003913 (decimal: 14611)
  VA  1: 0x000001ae (decimal:  430) --> VALID: 0x000037b9 (decimal: 14265)
  VA  2: 0x00000109 (decimal:  265) --> VALID: 0x00003714 (decimal: 14100)
  VA  3: 0x0000020b (decimal:  523) --> VALID: 0x00003816 (decimal: 14358)
  VA  4: 0x0000019e (decimal:  414) --> VALID: 0x000037a9 (decimal: 14249)
  VA  5: 0x00000322 (decimal:  802) --> VALID: 0x0000392d (decimal: 14637)
  VA  6: 0x00000136 (decimal:  310) --> VALID: 0x00003741 (decimal: 14145)
  VA  7: 0x000001e8 (decimal:  488) --> VALID: 0x000037f3 (decimal: 14323)
  VA  8: 0x00000255 (decimal:  597) --> VALID: 0x00003860 (decimal: 14432)
  VA  9: 0x000003a1 (decimal:  929) --> VALID: 0x000039ac (decimal: 14764)
```

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 relocation.py -s 0 -n 10 -c -l 1k

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

Base-and-Bounds register information:

  Base   : 0x0000360b (decimal 13835)
  Limit  : 1024

Virtual Address Trace
  VA  0: 0x00000308 (decimal:  776) --> VALID: 0x00003913 (decimal: 14611)
  VA  1: 0x000001ae (decimal:  430) --> VALID: 0x000037b9 (decimal: 14265)
  VA  2: 0x00000109 (decimal:  265) --> VALID: 0x00003714 (decimal: 14100)
  VA  3: 0x0000020b (decimal:  523) --> VALID: 0x00003816 (decimal: 14358)
  VA  4: 0x0000019e (decimal:  414) --> VALID: 0x000037a9 (decimal: 14249)
  VA  5: 0x00000322 (decimal:  802) --> VALID: 0x0000392d (decimal: 14637)
  VA  6: 0x00000136 (decimal:  310) --> VALID: 0x00003741 (decimal: 14145)
  VA  7: 0x000001e8 (decimal:  488) --> VALID: 0x000037f3 (decimal: 14323)
  VA  8: 0x00000255 (decimal:  597) --> VALID: 0x00003860 (decimal: 14432)
  VA  9: 0x000003a1 (decimal:  929) --> VALID: 0x000039ac (decimal: 14764)
```

1.3 The maximum value base can be set to would be 16k-100 = 16284.

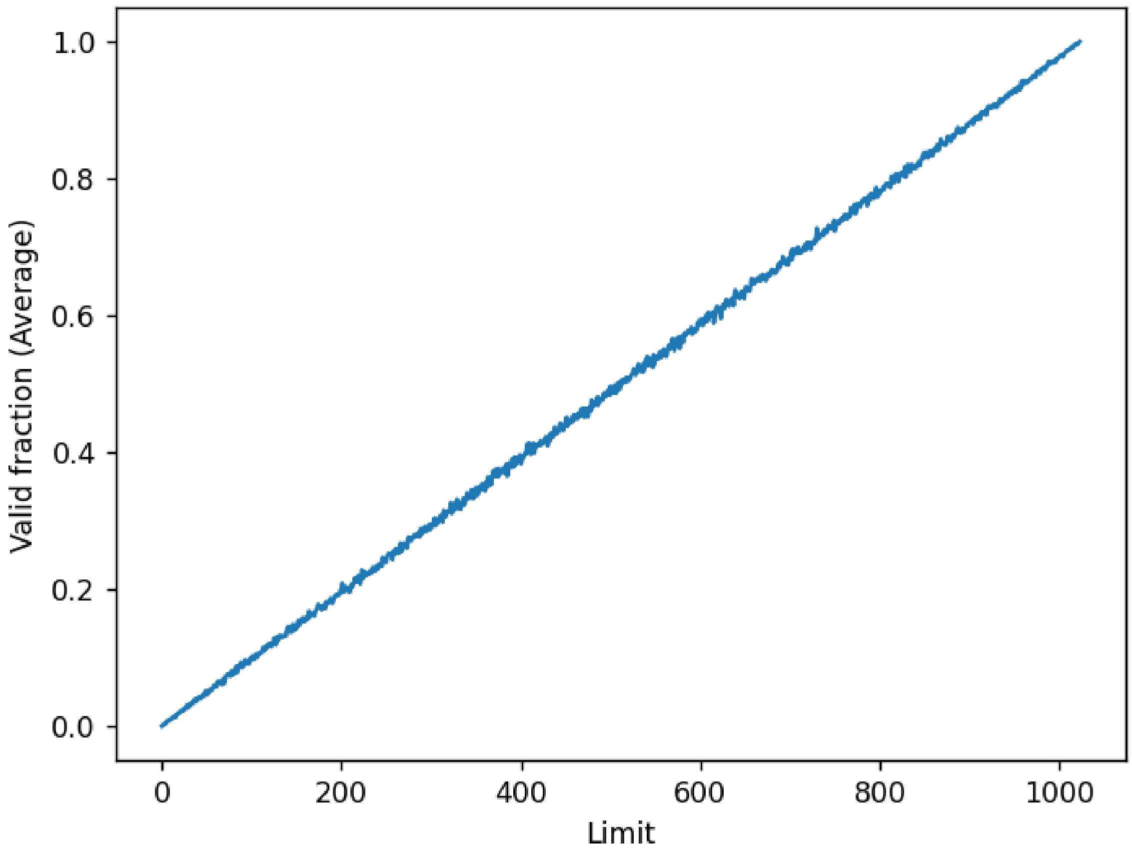
1.4 Running Q2 with -a 16k -p 64k

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 relocation.py -s 0 -n 10 -c -a 16k -p 64k

ARG seed 0
ARG address space size 16k
ARG phys mem size 64k

Base-and-Bounds register information:

  Base   : 0x0000c209 (decimal 49673)
  Limit  : 7554

Virtual Address Trace
  VA  0: 0x00001aea (decimal: 6890) --> VALID: 0x0000dcf3 (decimal: 56563)
  VA  1: 0x00001092 (decimal: 4242) --> VALID: 0x0000d29b (decimal: 53915)
  VA  2: 0x000020b8 (decimal: 8376) --> SEGMENTATION VIOLATION
  VA  3: 0x000019ea (decimal: 6634) --> VALID: 0x0000dbf3 (decimal: 56307)
  VA  4: 0x00003229 (decimal: 12841) --> SEGMENTATION VIOLATION
  VA  5: 0x00001369 (decimal: 4969) --> VALID: 0x0000d572 (decimal: 54642)
  VA  6: 0x00001e80 (decimal: 7808) --> SEGMENTATION VIOLATION
  VA  7: 0x00002556 (decimal: 9558) --> SEGMENTATION VIOLATION
  VA  8: 0x00003a1e (decimal: 14878) --> SEGMENTATION VIOLATION
  VA  9: 0x0000204c (decimal: 8268) --> SEGMENTATION VIOLATION
```

Running Q3 with -a 16k -p 64k

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 relocation.py -s 1 -n 10 -c -l 100 -a 16k -p 64k

ARG seed 1
ARG address space size 16k
ARG phys mem size 64k

Base-and-Bounds register information:

  Base   : 0x00002265 (decimal 8805)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x0000363c (decimal: 13884) --> SEGMENTATION VIOLATION
  VA  1: 0x000030e1 (decimal: 12513) --> SEGMENTATION VIOLATION
  VA  2: 0x00001053 (decimal: 4179) --> SEGMENTATION VIOLATION
  VA  3: 0x00001fb5 (decimal: 8117) --> SEGMENTATION VIOLATION
  VA  4: 0x00001cc4 (decimal: 7364) --> SEGMENTATION VIOLATION
  VA  5: 0x000029b3 (decimal: 10675) --> SEGMENTATION VIOLATION
  VA  6: 0x0000327a (decimal: 12922) --> SEGMENTATION VIOLATION
  VA  7: 0x00000601 (decimal: 1537) --> SEGMENTATION VIOLATION
  VA  8: 0x000001d0 (decimal:  464) --> SEGMENTATION VIOLATION
  VA  9: 0x0000357d (decimal: 13693) --> SEGMENTATION VIOLATION
```

1.5 Ran with address space = 1k and physical memory size = 16k, for 10000 iterations.



2.1

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 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


  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)
```

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 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)
```

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2 -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: 0x0000007a (decimal:  122) --> VALID in SEG1: 0x000001fa (decimal:  506)
  VA  1: 0x00000079 (decimal:  121) --> VALID in SEG1: 0x000001f9 (decimal:  505)
  VA  2: 0x00000007 (decimal:    7) --> VALID in SEG0: 0x00000007 (decimal:    7)
  VA  3: 0x0000000a (decimal:   10) --> VALID in SEG0: 0x0000000a (decimal:   10)
  VA  4: 0x0000006a (decimal:  106) --> SEGMENTATION VIOLATION (SEG1)
```

2.2 Highest legal virtual address of Segment 0 is 19 (0+19) and lowest legal virtual address of Segment 1 is 108 (127-19). Highest illegal virtual address is 107, lowest illegal virtual address is 20.
We pass flag as -A 19,20,107,108 to verify:

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 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)
```

2.3 Values for b = 0, B =128, l = 2, L = 2 will output the required result. This is because virtual addresses between 2 and 14 (inclusive) are all illegal.

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 segmentation.py -a 16 -p 128 b 0 -l 2 -B 128 -L 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) : 0x0000006c (decimal 108)
  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: 0x0000006c (decimal:  108)
  VA  1: 0x00000001 (decimal:    1) --> VALID in SEG0: 0x0000006d (decimal:  109)
  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)
```

2.4 For roughly 90% of valid virtual addresses, we keep both limit registers as approximately virtual address space*0.45. In the case of below image, 1024*0.45 ≈ 461.

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 segmentation.py -a 1k -p 16k b 0 -l 461 -B 16k -L 461 -n 20 -c
ARG seed 0
ARG address space size 1k
ARG phys mem size 16k

Segment register information:

  Segment 0 base  (grows positive) : 0x0000360b (decimal 13835)
  Segment 0 limit                  : 461

  Segment 1 base  (grows negative) : 0x00004000 (decimal 16384)
  Segment 1 limit                  : 461

Virtual Address Trace
  VA  0: 0x00000308 (decimal:  776) --> VALID in SEG1: 0x00003f08 (decimal: 16136)
  VA  1: 0x000001ae (decimal:  430) --> VALID in SEG0: 0x000037b9 (decimal: 14265)
  VA  2: 0x00000109 (decimal:  265) --> VALID in SEG0: 0x00003714 (decimal: 14100)
  VA  3: 0x0000020b (decimal:  523) --> SEGMENTATION VIOLATION (SEG1)
  VA  4: 0x0000019e (decimal:  414) --> VALID in SEG0: 0x000037a9 (decimal: 14249)
  VA  5: 0x00000322 (decimal:  802) --> VALID in SEG1: 0x00003f22 (decimal: 16162)
  VA  6: 0x00000136 (decimal:  310) --> VALID in SEG0: 0x00003741 (decimal: 14145)
  VA  7: 0x000001e8 (decimal:  488) --> SEGMENTATION VIOLATION (SEG0)
  VA  8: 0x00000255 (decimal:  597) --> VALID in SEG1: 0x00003e55 (decimal: 15957)
  VA  9: 0x000003a1 (decimal:  929) --> VALID in SEG1: 0x00003fa1 (decimal: 16289)
  VA 10: 0x00000204 (decimal:  516) --> SEGMENTATION VIOLATION (SEG1)
  VA 11: 0x00000120 (decimal:  288) --> VALID in SEG0: 0x0000372b (decimal: 14123)
  VA 12: 0x00000305 (decimal:  773) --> VALID in SEG1: 0x00003f05 (decimal: 16133)
  VA 13: 0x00000279 (decimal:  633) --> VALID in SEG1: 0x00003e79 (decimal: 15993)
  VA 14: 0x00000100 (decimal:  256) --> VALID in SEG0: 0x0000370b (decimal: 14091)
  VA 15: 0x000003a3 (decimal:  931) --> VALID in SEG1: 0x00003fa3 (decimal: 16291)
  VA 16: 0x000003ee (decimal: 1006) --> VALID in SEG1: 0x00003fee (decimal: 16366)
  VA 17: 0x0000033d (decimal:  829) --> VALID in SEG1: 0x00003f3d (decimal: 16189)
  VA 18: 0x0000039b (decimal:  923) --> VALID in SEG1: 0x00003f9b (decimal: 16283)
  VA 19: 0x0000013d (decimal:  317) --> VALID in SEG0: 0x00003748 (decimal: 14152)
```

2.5 Set both limit registers to 0.

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 segmentation.py -a 128 -p 512 -b 0 -l 0 -B 512 -L 0 -c -n 10
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                  : 0

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

Virtual Address Trace
  VA  0: 0x0000006c (decimal:  108) --> SEGMENTATION VIOLATION (SEG1)
  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)
  VA  5: 0x00000033 (decimal:   51) --> SEGMENTATION VIOLATION (SEG0)
  VA  6: 0x00000064 (decimal:  100) --> SEGMENTATION VIOLATION (SEG1)
  VA  7: 0x00000026 (decimal:   38) --> SEGMENTATION VIOLATION (SEG0)
  VA  8: 0x0000003d (decimal:   61) --> SEGMENTATION VIOLATION (SEG0)
  VA  9: 0x0000004a (decimal:   74) --> SEGMENTATION VIOLATION (SEG1)
```

Q3.

Note here that :

$$\text{page table size} = 2^{(\text{VPN bits})} \times (\text{Page table entry size})$$

1. Running by varying virtual address bytes:
   We observe that each bit increase in the virtual address size increases the page table size twice. In the below examples, increasing the address size by 10 bytes increased the page table size by a factor of 2^{10}.

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 paging-linear-size.py -c -v 20
ARG bits in virtual address 20
ARG page size 4k
ARG pte size 4

Recall that an address has two components:
[ Virtual Page Number (VPN) | Offset ]

The number of bits in the virtual address: 20
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 8
Thus, a virtual address looks like this:

V V V V V V V V | O O O O O O O O O O O O

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 256.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  1024 bytes
  in KB: 1.0
  in MB: 0.0009765625
```

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 paging-linear-size.py -c -v 30
ARG bits in virtual address 30
ARG page size 4k
ARG pte size 4

Recall that an address has two components:
[ Virtual Page Number (VPN) | Offset ]

The number of bits in the virtual address: 30
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 18
Thus, a virtual address looks like this:

V V V V V V V V V V V V V V V V V V | O O O O O O O O O O O O

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 262144.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  1048576 bytes
  in KB: 1024.0
  in MB: 1.0
```

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 paging-linear-size.py -c -v 40
ARG bits in virtual address 40
ARG page size 4k
ARG pte size 4

Recall that an address has two components:
[ Virtual Page Number (VPN) | Offset ]

The number of bits in the virtual address: 40
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 28
Thus, a virtual address looks like this:

V V V V V V V V V V V V V V V V V V V V V V V V V V V V | O O O O O O O O O O O O

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 268435456.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  1073741824 bytes
  in KB: 1048576.0
  in MB: 1024.0
```

2. Running by varying the page size: doubling the page size halves the page table size. This is because there is more offset bits on increasing page size, hence lesser virtual page number bits. (Page table size = $2^{(\text{VPN bits})}$)

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 paging-linear-size.py -c -p 2k
ARG bits in virtual address 32
ARG page size 2k
ARG pte size 4

Recall that an address has two components:
[ Virtual Page Number (VPN) | Offset ]

The number of bits in the virtual address: 32
The page size: 2048 bytes
Thus, the number of bits needed in the offset: 11
Which leaves this many bits for the VPN: 21
Thus, a virtual address looks like this:

V V V V V V V V V V V V V V V V V V V V V | O O O O O O O O O O O

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 2097152.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  8388608 bytes
  in KB: 8192.0
  in MB: 8.0
```

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 paging-linear-size.py -c -p 4k
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 4

Recall that an address has two components:
[ Virtual Page Number (VPN) | Offset ]

The number of bits in the virtual address: 32
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 20
Thus, a virtual address looks like this:

V V V V V V V V V V V V V V V V V V V V | O O O O O O O O O O O O

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 1048576.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  4194304 bytes
  in KB: 4096.0
  in MB: 4.0
```

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 paging-linear-size.py -c -p 8k
ARG bits in virtual address 32
ARG page size 8k
ARG pte size 4

Recall that an address has two components:
[ Virtual Page Number (VPN) | Offset ]

The number of bits in the virtual address: 32
The page size: 8192 bytes
Thus, the number of bits needed in the offset: 13
Which leaves this many bits for the VPN: 19
Thus, a virtual address looks like this:

V V V V V V V V V V V V V V V V V V V | O O O O O O O O O O O O O

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 524288.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  2097152 bytes
  in KB: 2048.0
  in MB: 2.0
```

3. Varying page table entry size:
   We observe page table size doubling on doubling page table entry size. This is because page table size Is directly proportional to the page table entry size.

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 paging-linear-size.py -c -e 4
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 4

Recall that an address has two components:
[ Virtual Page Number (VPN) | Offset ]

The number of bits in the virtual address: 32
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 20
Thus, a virtual address looks like this:

V V V V V V V V V V V V V V V V V V V V | O O O O O O O O O O O O

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 1048576.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  4194304 bytes
  in KB: 4096.0
  in MB: 4.0
```

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 paging-linear-size.py -c -e 8
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 8

Recall that an address has two components:
[ Virtual Page Number (VPN) | Offset ]

The number of bits in the virtual address: 32
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 20
Thus, a virtual address looks like this:

V V V V V V V V V V V V V V V V V V V V | O O O O O O O O O O O O

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 1048576.0
- The size of each page table entry, which is: 8
And then multiply them together. The final result:
  8388608 bytes
  in KB: 8192.0
  in MB: 8.0
```

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 paging-linear-size.py -c -e 16
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 16

Recall that an address has two components:
[ Virtual Page Number (VPN) | Offset ]

The number of bits in the virtual address: 32
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 20
Thus, a virtual address looks like this:

V V V V V V V V V V V V V V V V V V V V | O O O O O O O O O O O O

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 1048576.0
- The size of each page table entry, which is: 16
And then multiply them together. The final result:
  16777216 bytes
  in KB: 16384.0
  in MB: 16.0
```

4.1 Increasing the address space increases the page table size proportionately.

-a 1m :

```
  [     1017]   0x00000000
  [     1018]   0x00000000
  [     1019]   0x8002e9c9
  [     1020]   0x00000000
  [     1021]   0x00000000
  [     1022]   0x00000000
  [     1023]   0x00000000

Virtual Address Trace
```

-a 2m :

```
  [     2040]   0x80038ed5
  [     2041]   0x00000000
  [     2042]   0x00000000
  [     2043]   0x00000000
  [     2044]   0x00000000
  [     2045]   0x00000000
  [     2046]   0x8000eedd
  [     2047]   0x00000000

Virtual Address Trace
```

-a 4m :

```
  [     4089]   0x80078d9a
  [     4090]   0x8006ca8e
  [     4091]   0x800160f8
  [     4092]   0x80015abc
  [     4093]   0x8001483a
  [     4094]   0x00000000
  [     4095]   0x8002e298

Virtual Address Trace
```

Increasing page size decreases the page table size proportionately:

-P 1k :

```
[     1015]   0x00000000
[     1016]   0x00000000
[     1017]   0x00000000
[     1018]   0x00000000
[     1019]   0x8002e9c9
[     1020]   0x00000000
[     1021]   0x00000000
[     1022]   0x00000000
[     1023]   0x00000000

Virtual Address Trace
```

-P 2k :

```
[      503]   0x8003ea63
[      504]   0x00000000
[      505]   0x00000000
[      506]   0x00000000
[      507]   0x00000000
[      508]   0x8001a7f2
[      509]   0x8001c337
[      510]   0x00000000
[      511]   0x00000000

Virtual Address Trace
```

-P 4k :

```
[      249]   0x00000000
[      250]   0x00000000
[      251]   0x8001efec
[      252]   0x8001cd5b
[      253]   0x800125d2
[      254]   0x80019c37
[      255]   0x8001fb27

Virtual Address Trace
```

Larger page size decreases the page table size, but has a trade off, such as increases fragmentation, and more memory overhead.

4.2. As u is increased the number of allocated pages increases.

-u 0 : No entries are allocated.

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 0
ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 1k
ARG verbose True
ARG addresses -1
  [       0]   0x00000000
  [       1]   0x00000000
  [       2]   0x00000000
  [       3]   0x00000000
  [       4]   0x00000000
  [       5]   0x00000000
  [       6]   0x00000000
  [       7]   0x00000000
  [       8]   0x00000000
  [       9]   0x00000000
  [      10]   0x00000000
  [      11]   0x00000000
  [      12]   0x00000000
  [      13]   0x00000000
  [      14]   0x00000000
  [      15]   0x00000000
```

-u 25 : 6 entries are allocated.

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 25
ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 1k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]   0x80000018
  [       1]   0x00000000
  [       2]   0x00000000
  [       3]   0x00000000
  [       4]   0x00000000
  [       5]   0x80000009
  [       6]   0x00000000
  [       7]   0x00000000
  [       8]   0x80000010
  [       9]   0x00000000
  [      10]   0x80000013
  [      11]   0x00000000
  [      12]   0x8000001f
  [      13]   0x8000001c
  [      14]   0x00000000
  [      15]   0x00000000
```

-u 50 : 9 entries are allocated.

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 50
ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 1k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]   0x80000018
  [       1]   0x00000000
  [       2]   0x00000000
  [       3]   0x8000000c
  [       4]   0x80000009
  [       5]   0x00000000
  [       6]   0x8000001d
  [       7]   0x80000013
  [       8]   0x00000000
  [       9]   0x8000001f
  [      10]   0x8000001c
  [      11]   0x00000000
  [      12]   0x8000000f
  [      13]   0x00000000
  [      14]   0x00000000
  [      15]   0x80000008
```

-u 75 : all entries are allocated.

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 75
ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 1k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]   0x80000018
  [       1]   0x80000008
  [       2]   0x8000000c
  [       3]   0x80000009
  [       4]   0x80000012
  [       5]   0x80000010
  [       6]   0x8000001f
  [       7]   0x8000001c
  [       8]   0x80000017
  [       9]   0x80000015
  [      10]   0x80000003
  [      11]   0x80000013
  [      12]   0x8000001e
  [      13]   0x8000001b
  [      14]   0x80000019
  [      15]   0x80000000
```

-u 100 : all entries are allocated.

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 100
ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 1k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]   0x80000018
  [       1]   0x80000008
  [       2]   0x8000000c
  [       3]   0x80000009
  [       4]   0x80000012
  [       5]   0x80000010
  [       6]   0x8000001f
  [       7]   0x8000001c
  [       8]   0x80000017
  [       9]   0x80000015
  [      10]   0x80000003
  [      11]   0x80000013
  [      12]   0x8000001e
  [      13]   0x8000001b
  [      14]   0x80000019
  [      15]   0x80000000
```

4.3 All three cases seem to be unrealistic:

Case 1: because each page is allotted only 8B, and at most 4 pages can be present in the address space

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 paging-linear-translate.py -P 8 -a 32 -p 1024 -v -s 1 -c
ARG seed 1
ARG address space size 32
ARG phys mem size 1024
ARG page size 8
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]   0x00000000
  [       1]   0x80000061
  [       2]   0x00000000
  [       3]   0x00000000

Virtual Address Trace
  VA 0x0000000e (decimal:       14) --> 0000030e (decimal      782) [VPN 1]
  VA 0x00000014 (decimal:       20) -->  Invalid (VPN 2 not valid)
  VA 0x00000019 (decimal:       25) -->  Invalid (VPN 3 not valid)
  VA 0x00000003 (decimal:        3) -->  Invalid (VPN 0 not valid)
  VA 0x00000000 (decimal:        0) -->  Invalid (VPN 0 not valid)
```

Case 2: because at most 4 pages in the address space.

```
nithinsabu@DESKTOP-QM7QAJ9:/mnt/d/Operating Systems/Lab 7$ python2 paging-linear-translate.py -P 8k -a 32k -p 1m -v -s 2 -c
ARG seed 2
ARG address space size 32k
ARG phys mem size 1m
ARG page size 8k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]   0x80000079
  [       1]   0x00000000
  [       2]   0x00000000
  [       3]   0x8000005e

Virtual Address Trace
  VA 0x000055b9 (decimal:    21945) -->  Invalid (VPN 2 not valid)
  VA 0x00002771 (decimal:    10097) -->  Invalid (VPN 1 not valid)
  VA 0x00004d8f (decimal:    19855) -->  Invalid (VPN 2 not valid)
  VA 0x00004dab (decimal:    19883) -->  Invalid (VPN 2 not valid)
  VA 0x00004a64 (decimal:    19044) -->  Invalid (VPN 2 not valid)
```

Case 3: Page size is 1MB which is wastage of memory, hence unrealistic.

```
  [     249]   0x00000000
  [     250]   0x800001eb
  [     251]   0x00000000
  [     252]   0x00000000
  [     253]   0x00000000
  [     254]   0x80000159
  [     255]   0x00000000

Virtual Address Trace
  VA 0x0308b24d (decimal: 50901581) --> 1f68b24d (decimal 526955085) [VPN 48]
  VA 0x042351e6 (decimal: 69423590) -->  Invalid (VPN 66 not valid)
  VA 0x02feb67b (decimal: 50247291) --> 0a9eb67b (decimal 178173563) [VPN 47]
  VA 0x0b46977d (decimal: 189175677) -->  Invalid (VPN 180 not valid)
  VA 0x0dbcceb4 (decimal: 230477492) --> 1f2cceb4 (decimal 523030196) [VPN 219]
```

4.4 The limitations of the code are:

1. address space, physical memory, page size etc. must all be positive.

2. physical memory and address space must be multiple of page size.

3. page size should be a multiple of 2.

4. page size should be smaller than the physical memory, but larger than address space size.