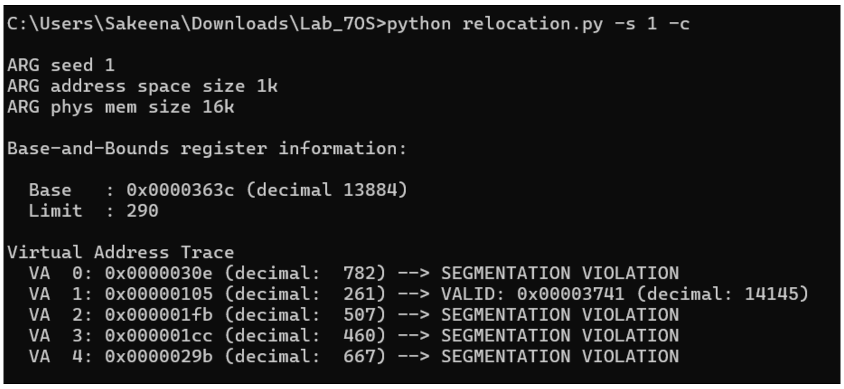
CS-314 OS LAB ASSIGNMENT - 7

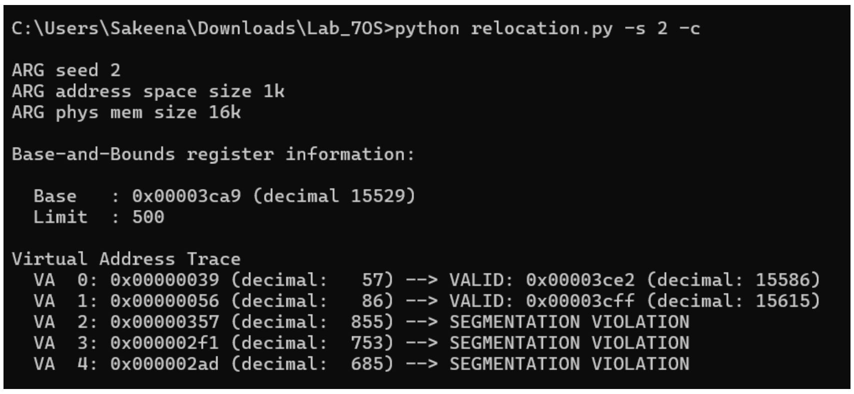
D.S.Sadhika- 210010062

Q1)

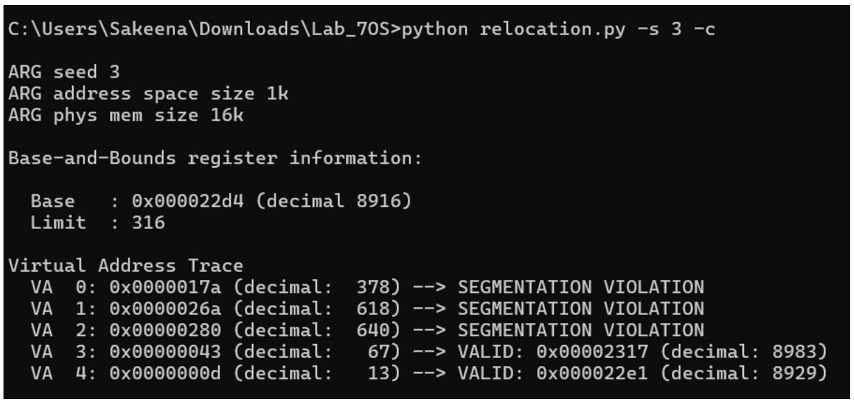
1. To execute relocation.py, I had run the seed instructions -s 1, -s 2, and -s 3. I added the command -c at the end to compute the translation. VA 1 was translated for the first two seeds because it fell within the bounds. The remaining VAs exceeded the limits. The following snaps are evidence of it . Because the seed values are consistent, the random number generator will produce the same results each time the same seed is used



```
C:\Users\Sakeena\Downloads\Lab_7OS>python 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
```



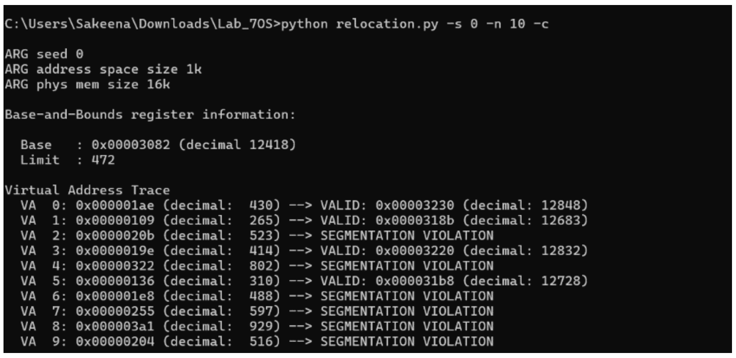
```
C:\Users\Sakeena\Downloads\Lab_7OS>python relocation.py -s 2 -c

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
```



```
C:\Users\Sakeena\Downloads\Lab_7OS>python 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)
```

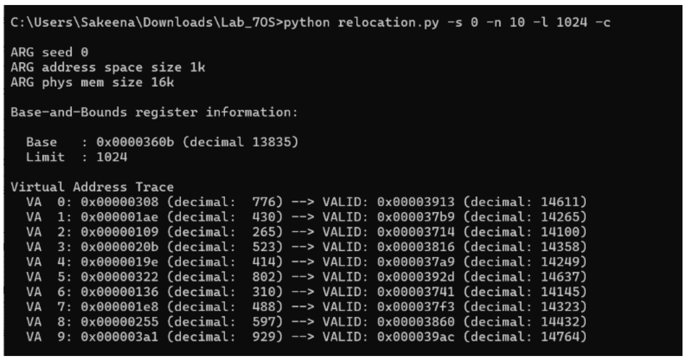
The key takeaway is that the randomness introduced by the seed values leads to variations in the placement of the address space within the physical memory and consequently affects the outcomes of the virtual address trace. Each run provides a different scenario based on the specific seed used.

2. First I had run with the flags : -s 0 -n 10 and computed then the virtual addresses traced were all not valid



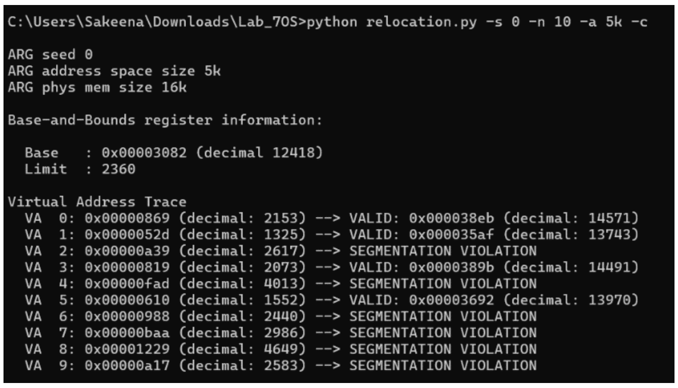
```
C:\Users\Sakeena\Downloads\Lab_7OS>python relocation.py -s 0 -n 10 -c

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
```

Now the bounds register -l is set to the maximum value of the virtual address trace from the above snapshot which is -l as 1024(1K) then we got all the valid addresses.



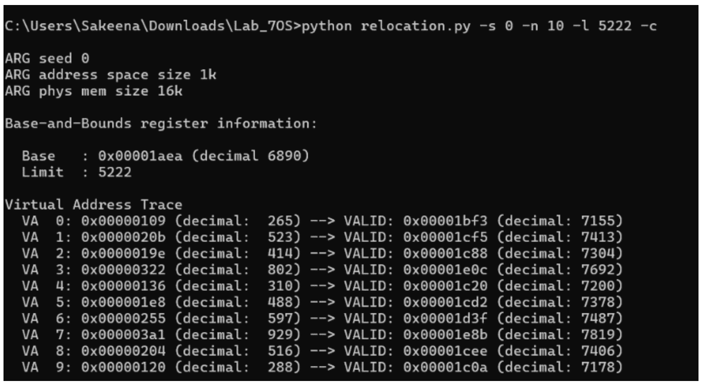
```
C:\Users\Sakeena\Downloads\Lab_7OS>python relocation.py -s 0 -n 10 -l 1024 -c

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)
```

But when we changed the address space size to 5K and run then we got some segmentation violations.



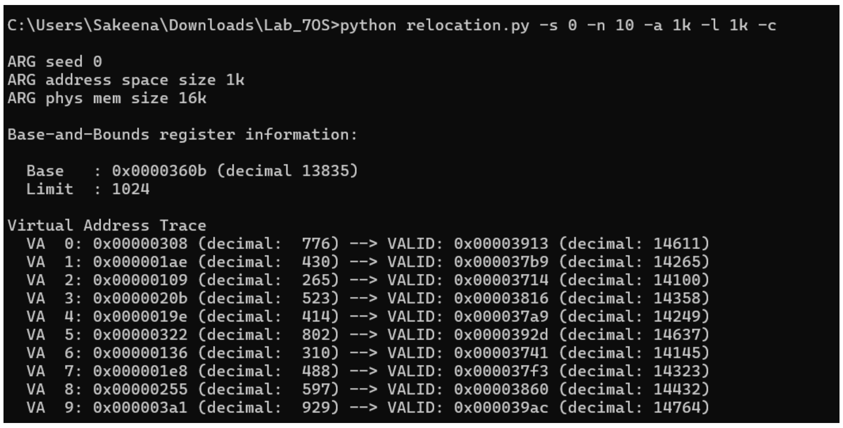
```
C:\Users\Sakeena\Downloads\Lab_7OS>python relocation.py -s 0 -n 10 -a 5k -c

ARG seed 0
ARG address space size 5k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x00003082 (decimal 12418)
  Limit  : 2360

Virtual Address Trace
  VA  0: 0x00000869 (decimal: 2153) --> VALID: 0x000038eb (decimal: 14571)
  VA  1: 0x0000052d (decimal: 1325) --> VALID: 0x000035af (decimal: 13743)
  VA  2: 0x00000a39 (decimal: 2617) --> SEGMENTATION VIOLATION
  VA  3: 0x00000819 (decimal: 2073) --> VALID: 0x0000389b (decimal: 14491)
  VA  4: 0x00000fad (decimal: 4013) --> SEGMENTATION VIOLATION
  VA  5: 0x00000610 (decimal: 1552) --> VALID: 0x00003692 (decimal: 13970)
  VA  6: 0x00000988 (decimal: 2440) --> SEGMENTATION VIOLATION
  VA  7: 0x00000baa (decimal: 2986) --> SEGMENTATION VIOLATION
  VA  8: 0x00001229 (decimal: 4649) --> SEGMENTATION VIOLATION
  VA  9: 0x00000a17 (decimal: 2583) --> SEGMENTATION VIOLATION
```

Whereas when the bounds register that -l is set to 5K and address space size was 1k then we got all Valid Virtual address trace.



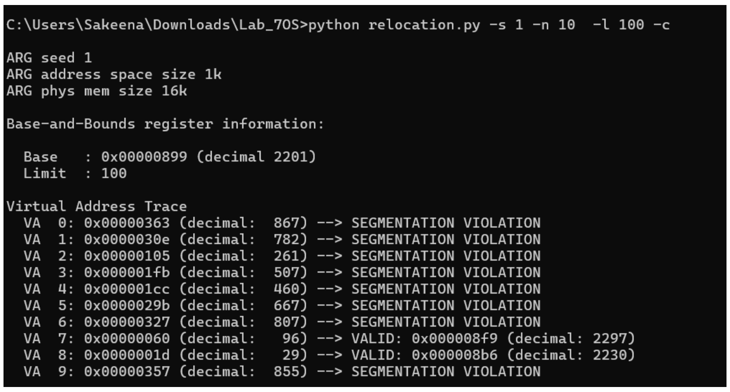
```
C:\Users\Sakeena\Downloads\Lab_7OS>python relocation.py -s 0 -n 10 -l 5222 -c

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

Base-and-Bounds register information:

  Base   : 0x00001aea (decimal 6890)
  Limit  : 5222

Virtual Address Trace
  VA  0: 0x00000109 (decimal:  265) --> VALID: 0x00001bf3 (decimal: 7155)
  VA  1: 0x0000020b (decimal:  523) --> VALID: 0x00001cf5 (decimal: 7413)
  VA  2: 0x0000019e (decimal:  414) --> VALID: 0x00001c88 (decimal: 7304)
  VA  3: 0x00000322 (decimal:  802) --> VALID: 0x00001e0c (decimal: 7692)
  VA  4: 0x00000136 (decimal:  310) --> VALID: 0x00001c20 (decimal: 7200)
  VA  5: 0x000001e8 (decimal:  488) --> VALID: 0x00001cd2 (decimal: 7378)
  VA  6: 0x00000255 (decimal:  597) --> VALID: 0x00001d3f (decimal: 7487)
  VA  7: 0x000003a1 (decimal:  929) --> VALID: 0x00001e8b (decimal: 7819)
  VA  8: 0x00000204 (decimal:  516) --> VALID: 0x00001cee (decimal: 7406)
  VA  9: 0x00000120 (decimal:  288) --> VALID: 0x00001c0a (decimal: 7178)
```

This shows that The address space has an impact on how virtual addresses (decimal) are created, as shown in the figures. Of course, since they must be in the address area. To guarantee that all of the address spaces generated are valid, we must ensure **that limit=address space**. All virtual addresses generated within the address space will only become valid after the limit is applied to them. **address space!= maximum limit.**
Hence, for address space = limit = 1K, all the generated addresses are valid.



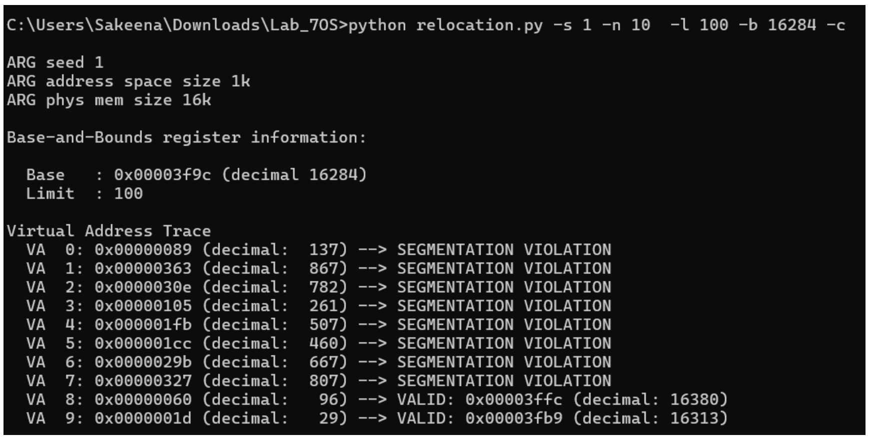
```
C:\Users\Sakeena\Downloads\Lab_7OS>python relocation.py -s 0 -n 10 -a 1k -l 1k -c

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)
```

3. Default b is set to -1

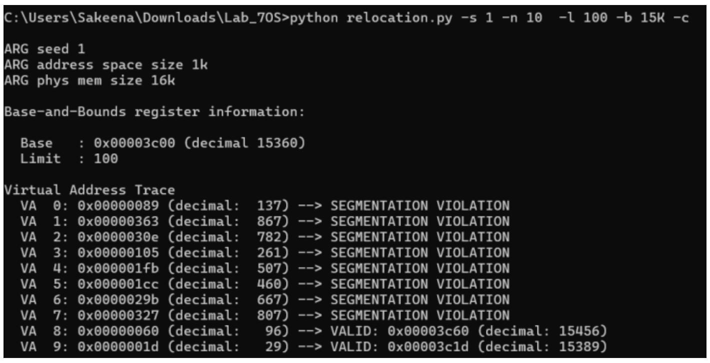


```
C:\Users\Sakeena\Downloads\Lab_7OS>python relocation.py -s 1 -n 10  -l 100 -c

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

Base-and-Bounds register information:

  Base   : 0x00000899 (decimal 2201)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x00000363 (decimal:  867) --> SEGMENTATION VIOLATION
  VA  1: 0x0000030e (decimal:  782) --> SEGMENTATION VIOLATION
  VA  2: 0x00000105 (decimal:  261) --> SEGMENTATION VIOLATION
  VA  3: 0x000001fb (decimal:  507) --> SEGMENTATION VIOLATION
  VA  4: 0x000001cc (decimal:  460) --> SEGMENTATION VIOLATION
  VA  5: 0x0000029b (decimal:  667) --> SEGMENTATION VIOLATION
  VA  6: 0x00000327 (decimal:  807) --> SEGMENTATION VIOLATION
  VA  7: 0x00000060 (decimal:   96) --> VALID: 0x000008f9 (decimal: 2297)
  VA  8: 0x0000001d (decimal:   29) --> VALID: 0x000008b6 (decimal: 2230)
  VA  9: 0x00000357 (decimal:  855) --> SEGMENTATION VIOLATION
```

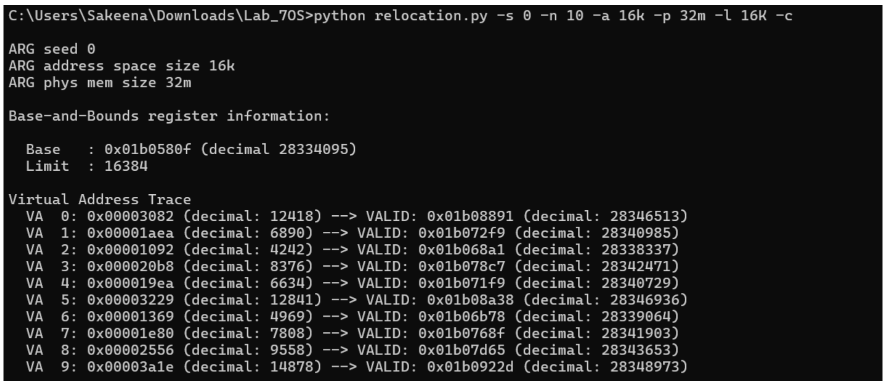
The available address space is 1K. The entire address space, which is 100 for us, should reside in the physical memory. As a result, up to 100 addresses beginning at the base are permitted; each address must fit inside of 16K. That implies that the base can begin with **16K-100, or 16*1024-100 (because 1K =1024), or 16284 .(base= psize-limit)**
The condition **base + limit <= psize** ensures that the sum of the base register and the limit register does not exceed the size of the physical memory. If it did exceed, it would mean that the entire address space cannot fit into the available physical memory, which could lead to out-of-bounds memory access or other issues. The below figure shows it:


```
C:\Users\Sakeena\Downloads\Lab_7OS>python relocation.py -s 1 -n 10  -l 100 -b 16284 -c

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

Base-and-Bounds register information:

  Base   : 0x00003f9c (decimal 16284)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x00000089 (decimal:  137) --> SEGMENTATION VIOLATION
  VA  1: 0x00000363 (decimal:  867) --> SEGMENTATION VIOLATION
  VA  2: 0x0000030e (decimal:  782) --> SEGMENTATION VIOLATION
  VA  3: 0x00000105 (decimal:  261) --> SEGMENTATION VIOLATION
  VA  4: 0x000001fb (decimal:  507) --> SEGMENTATION VIOLATION
  VA  5: 0x000001cc (decimal:  460) --> SEGMENTATION VIOLATION
  VA  6: 0x0000029b (decimal:  667) --> SEGMENTATION VIOLATION
  VA  7: 0x00000327 (decimal:  807) --> SEGMENTATION VIOLATION
  VA  8: 0x00000060 (decimal:   96) --> VALID: 0x00003ffc (decimal: 16380)
  VA  9: 0x0000001d (decimal:   29) --> VALID: 0x00003fb9 (decimal: 16313)
```

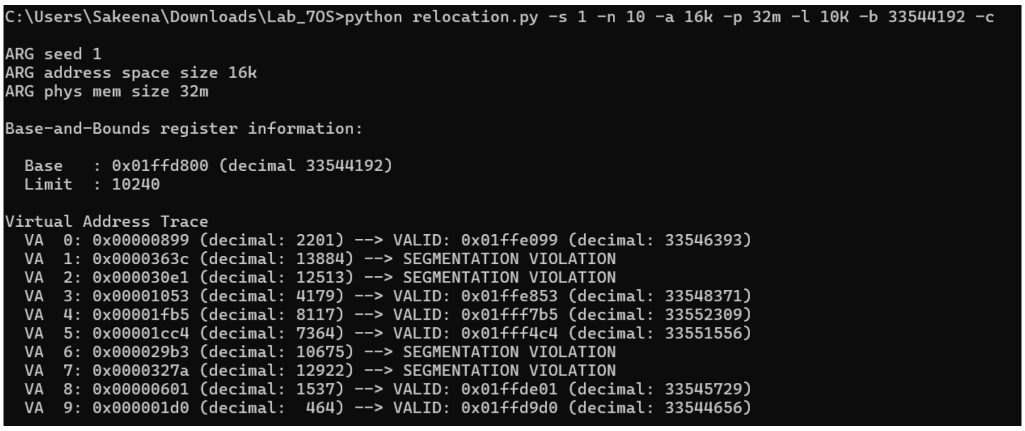
ALTERNATIVELY,the following approach can also be seen:
Everything from base up to 1K should be in physical memory if we want the full address space, including beyond the limit, to be in that storage medium. Base memory in this case should start at 16K-1K=15K.


```
C:\Users\Sakeena\Downloads\Lab_7OS>python relocation.py -s 1 -n 10  -l 100 -b 15K -c

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

Base-and-Bounds register information:

  Base   : 0x00003c00 (decimal 15360)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x00000089 (decimal:  137) --> SEGMENTATION VIOLATION
  VA  1: 0x00000363 (decimal:  867) --> SEGMENTATION VIOLATION
  VA  2: 0x0000030e (decimal:  782) --> SEGMENTATION VIOLATION
  VA  3: 0x00000105 (decimal:  261) --> SEGMENTATION VIOLATION
  VA  4: 0x000001fb (decimal:  507) --> SEGMENTATION VIOLATION
  VA  5: 0x000001cc (decimal:  460) --> SEGMENTATION VIOLATION
  VA  6: 0x0000029b (decimal:  667) --> SEGMENTATION VIOLATION
  VA  7: 0x00000327 (decimal:  807) --> SEGMENTATION VIOLATION
  VA  8: 0x00000060 (decimal:   96) --> VALID: 0x00003c60 (decimal: 15456)
  VA  9: 0x0000001d (decimal:   29) --> VALID: 0x00003c1d (decimal: 15389)
```

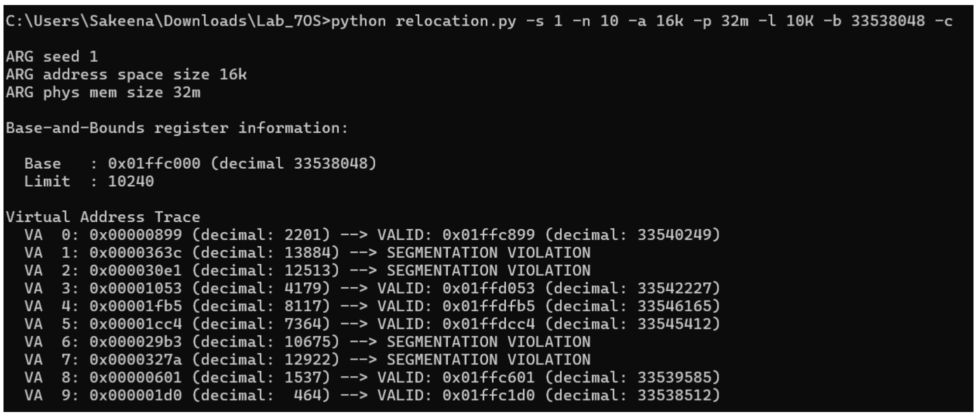
4. Taking **-a 16K**, **and -p 32m**. Run question 2 once more, to **use -l 16K** to ensure that all addresses created are **valid**.


```
C:\Users\Sakeena\Downloads\Lab_7OS>python relocation.py -s 0 -n 10 -a 16k -p 32m -l 16K -c

ARG seed 0
ARG address space size 16k
ARG phys mem size 32m

Base-and-Bounds register information:

  Base   : 0x01b0580f (decimal 28334095)
  Limit  : 16384

Virtual Address Trace
  VA  0: 0x00003082 (decimal: 12418) --> VALID: 0x01b08891 (decimal: 28346513)
  VA  1: 0x00001aea (decimal: 6890) --> VALID: 0x01b072f9 (decimal: 28340985)
  VA  2: 0x00001092 (decimal: 4242) --> VALID: 0x01b068a1 (decimal: 28338337)
  VA  3: 0x000020b8 (decimal: 8376) --> VALID: 0x01b078c7 (decimal: 28342471)
  VA  4: 0x000019ea (decimal: 6634) --> VALID: 0x01b071f9 (decimal: 28340729)
  VA  5: 0x00003229 (decimal: 12841) --> VALID: 0x01b08a38 (decimal: 28346936)
  VA  6: 0x00001369 (decimal: 4969) --> VALID: 0x01b06b78 (decimal: 28339064)
  VA  7: 0x00001e80 (decimal: 7808) --> VALID: 0x01b0768f (decimal: 28341903)
  VA  8: 0x00002556 (decimal: 9558) --> VALID: 0x01b07d65 (decimal: 28343653)
  VA  9: 0x00003a1e (decimal: 14878) --> VALID: 0x01b0922d (decimal: 28348973)
```

With -a 16K, -p 32m, and -l as 10K, I ran question3.
The maximum value of b that can be specified is therefore **either -b = 32m-10K =** 32*1024*1024 - 10*1024 = 33544192. **Alternatively -b = 32m-16K = 32*1024*1024 - 16*1024 = 33538048**

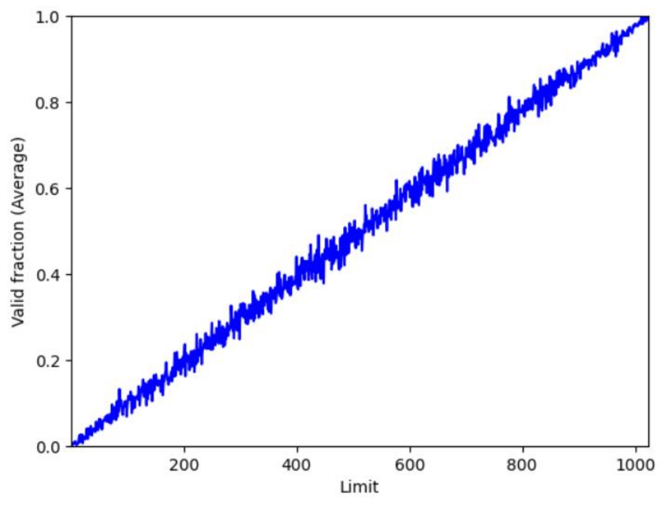


```
C:\Users\Sakeena\Downloads\Lab_7OS>python relocation.py -s 1 -n 10 -a 16k -p 32m -l 10K -b 33544192 -c

ARG seed 1
ARG address space size 16k
ARG phys mem size 32m

Base-and-Bounds register information:

  Base   : 0x01ffd800 (decimal 33544192)
  Limit  : 10240

Virtual Address Trace
  VA  0: 0x00000899 (decimal: 2201) --> VALID: 0x01ffe099 (decimal: 33546393)
  VA  1: 0x0000363c (decimal: 13884) --> SEGMENTATION VIOLATION
  VA  2: 0x000030e1 (decimal: 12513) --> SEGMENTATION VIOLATION
  VA  3: 0x00001053 (decimal: 4179) --> VALID: 0x01ffe853 (decimal: 33548371)
  VA  4: 0x00001fb5 (decimal: 8117) --> VALID: 0x01fff7b5 (decimal: 33552309)
  VA  5: 0x00001cc4 (decimal: 7364) --> VALID: 0x01fff4c4 (decimal: 33551556)
  VA  6: 0x000029b3 (decimal: 10675) --> SEGMENTATION VIOLATION
  VA  7: 0x0000327a (decimal: 12922) --> SEGMENTATION VIOLATION
  VA  8: 0x00000601 (decimal: 1537) --> VALID: 0x01ffde01 (decimal: 33545729)
  VA  9: 0x000001d0 (decimal:  464) --> VALID: 0x01ffd9d0 (decimal: 33544656)
```

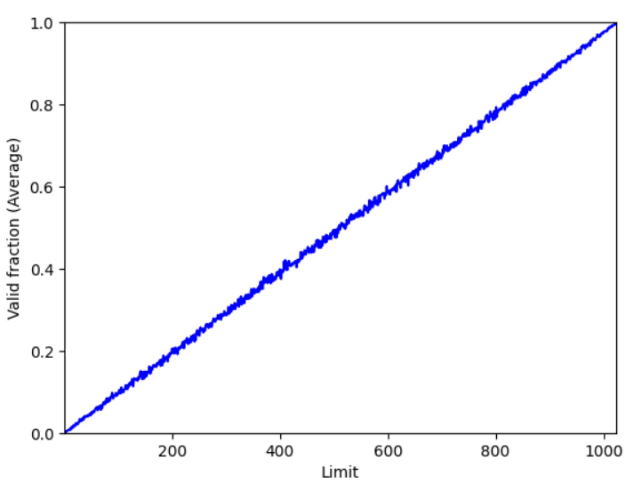


```
C:\Users\Sakeena\Downloads\Lab_7OS>python relocation.py -s 1 -n 10 -a 16k -p 32m -l 10K -b 33538048 -c

ARG seed 1
ARG address space size 16k
ARG phys mem size 32m

Base-and-Bounds register information:

  Base   : 0x01ffc000 (decimal 33538048)
  Limit  : 10240

Virtual Address Trace
  VA  0: 0x00000899 (decimal: 2201) --> VALID: 0x01ffc899 (decimal: 33540249)
  VA  1: 0x0000363c (decimal: 13884) --> SEGMENTATION VIOLATION
  VA  2: 0x000030e1 (decimal: 12513) --> SEGMENTATION VIOLATION
  VA  3: 0x00001053 (decimal: 4179) --> VALID: 0x01ffd053 (decimal: 33542227)
  VA  4: 0x00001fb5 (decimal: 8117) --> VALID: 0x01ffdfb5 (decimal: 33546165)
  VA  5: 0x00001cc4 (decimal: 7364) --> VALID: 0x01ffdcc4 (decimal: 33545412)
  VA  6: 0x000029b3 (decimal: 10675) --> SEGMENTATION VIOLATION
  VA  7: 0x0000327a (decimal: 12922) --> SEGMENTATION VIOLATION
  VA  8: 0x00000601 (decimal: 1537) --> VALID: 0x01ffc601 (decimal: 33539585)
  VA  9: 0x000001d0 (decimal:  464) --> VALID: 0x01ffc1d0 (decimal: 33538512)
```

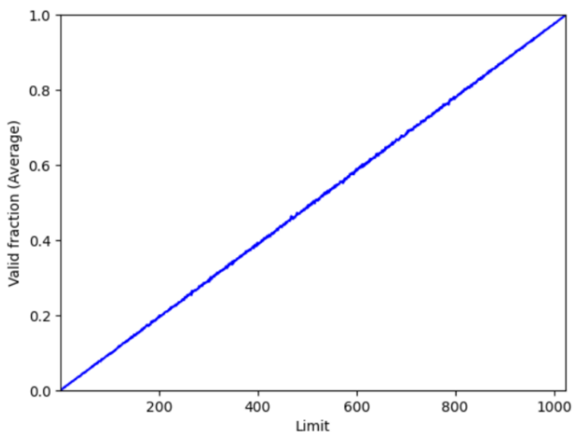
5. Here the limit values range from 0 to address space with a 1024-bit address space of my choice. The percentage of virtual addresses that were valid for each limit value was then calculated after a large number of potential virtual addresses were generated at random using a large number of seed values.
   X-axis = limit values Y-axis = valid fraction (1 = 100%)
   For 500 different seed values



For 5000 different seed values



For 50000 different seed values



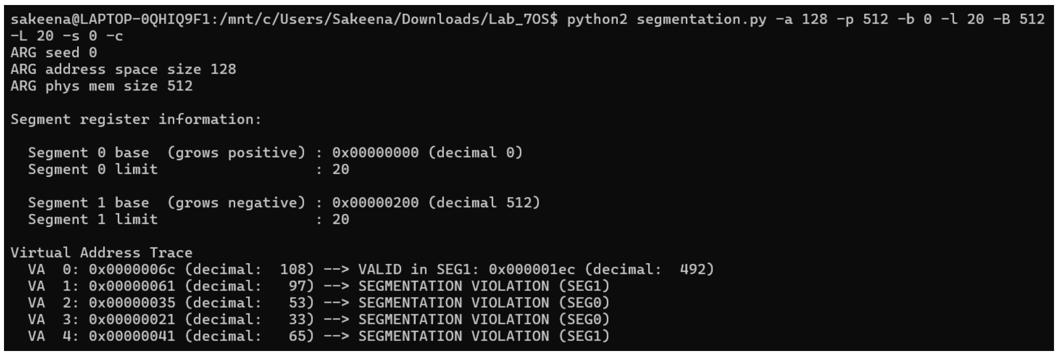
Given that the random virtual addresses are generated based on address space size, it

seems sense that the validity percentage approaches 100% as the limit increases.

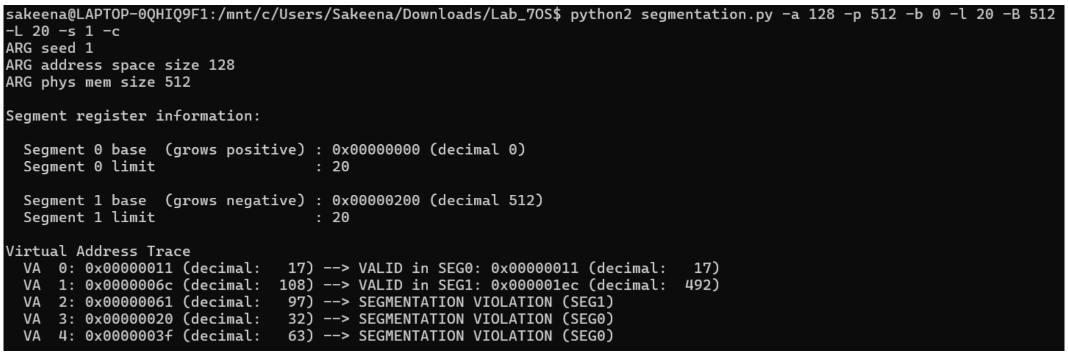
Obviously, a truer image is produced by using more seed values.

Q2)

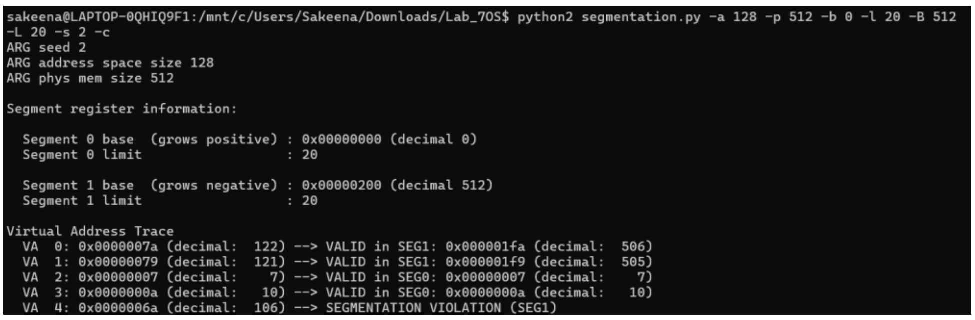
1. When we use the command -c at the end of the full command line, it converts any valid addresses and reports a segmentation fault for any invalid ones.



```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 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)
```



```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ 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)
```



```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ 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)
```

The above snaps show that the base values of segments 0 and 1 were used to translate valid addresses. The translation shows whether each virtual address is valid within its corresponding segment or if it results in a segmentation violation, along with the corresponding physical address when applicable.

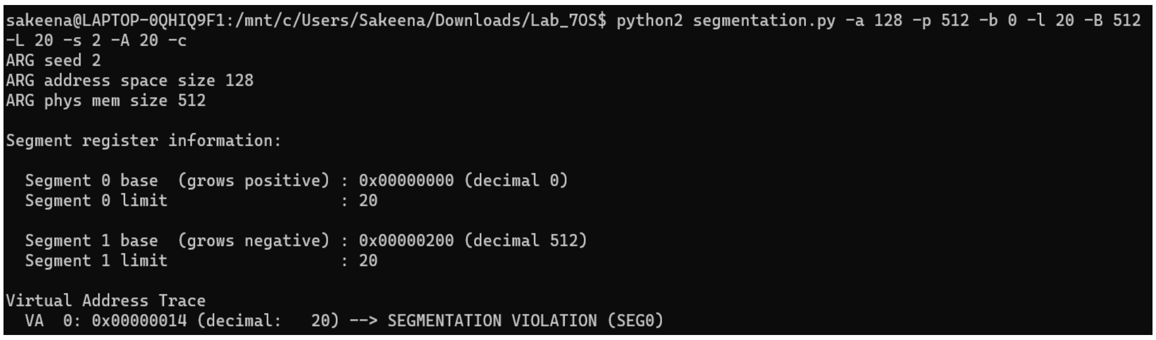
2. the legality of virtual addresses in segments is determined by the base and limit registers for each segment. The base address of each segment serves as its origin and the limit address as its depth limit. While segment 1 expands negatively or decreases, segment 0 expands positively or increases.

In Segment 0 :
Highest legal virtual address: sum of the base and limit registers, here it is **base+limit -1**
((When we add **base** and **limit**, we are essentially calculating the address of the last byte in the segment. Subtracting 1 ensures that we get the highest valid address within the segment. This is because addresses are typically zero-indexed, and the last byte in the segment would be at the address calculated by **base + limit - 1**.))
Highest Legal Virtual Address = **0x00000000 + 20 - 1 = (decimal: 19)**

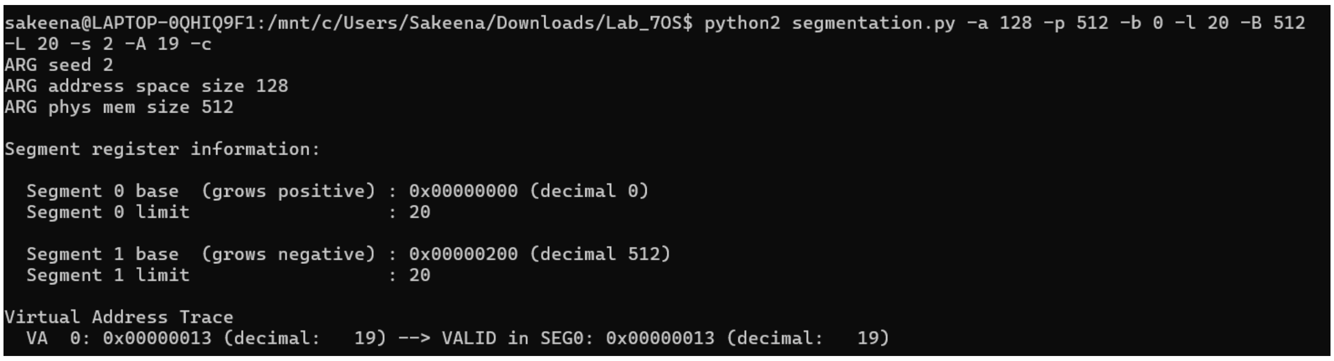
In Segment 1 :
Lowest legal virtual address: the **base register**, here it is 493.

Using Flag-A lets verify to check we r right or not !!

Segment 0's highest allowed virtual address is 19. (Base = 0 Limit = 20 thus, 0+20-1=19).



```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512
-L 20 -s 2 -A 20 -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: 0x00000014 (decimal:   20) --> SEGMENTATION VIOLATION (SEG0)
```

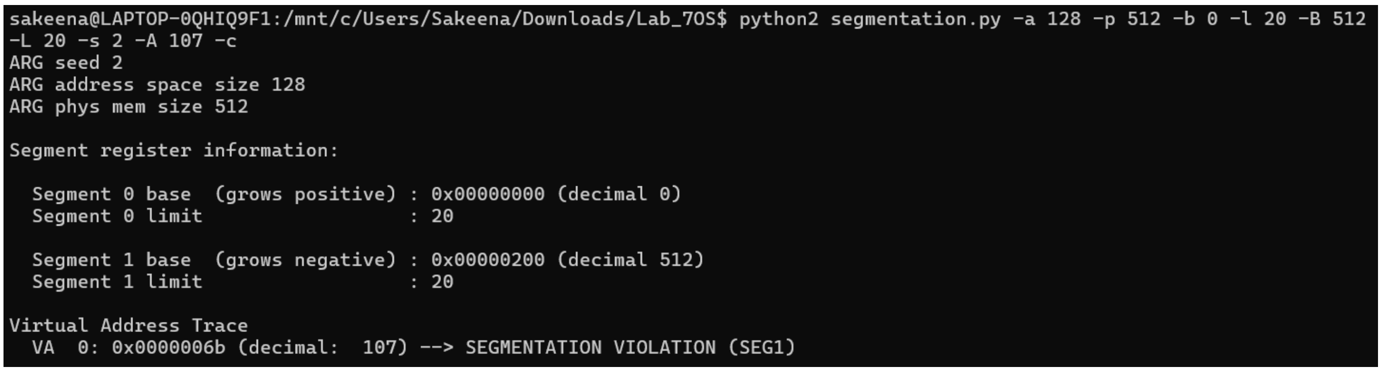


```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512
-L 20 -s 2 -A 19 -c
ARG seed 2
ARG address space size 128
ARG phys mem size 512

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: 0x00000013 (decimal:   19) --> VALID in SEG0: 0x00000013 (decimal:   19)
```

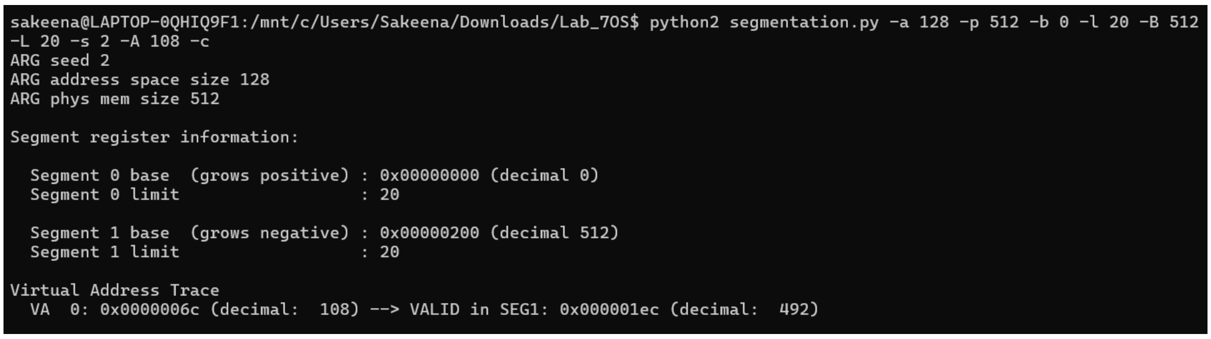
Segment 0: VALID For 19, INVALID for 20.

The segment in segment 1's lowest legal virtual address is 493.

(Base = 512,Max = 20. 128-20 = 108)



```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512
-L 20 -s 2 -A 107 -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: 0x0000006b (decimal:  107) --> SEGMENTATION VIOLATION (SEG1)
```


```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512
-L 20 -s 2 -A 108 -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: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492)
```

Segment 1: For 107- invalid, followed by 108-valid.

● The lowest and highest illegal addresses in the entire address space is:

○ 1. The lowest - 20

○ 2. The highest - 107

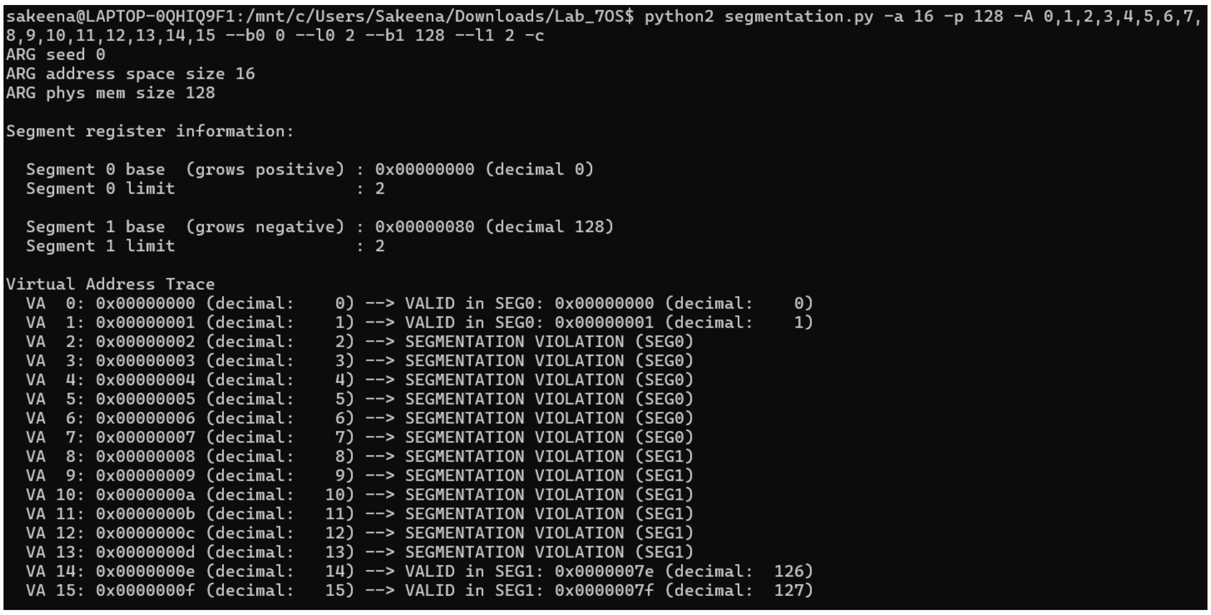
We are certain of the answer to the aforesaid Qs3 after examining the Qs1 and Qs2 above.

3. As we can see, we want all of the values to be false, including 0, 1, 14, 15, and the rest. Because of this, the base should be 0 and the maximum should be 2, as opposed to the base being 128 and the limit being 2.

   The only addresses that can be real in segments 0 and 1 are the first two (which expand

   negatively), and all other addresses must be false.

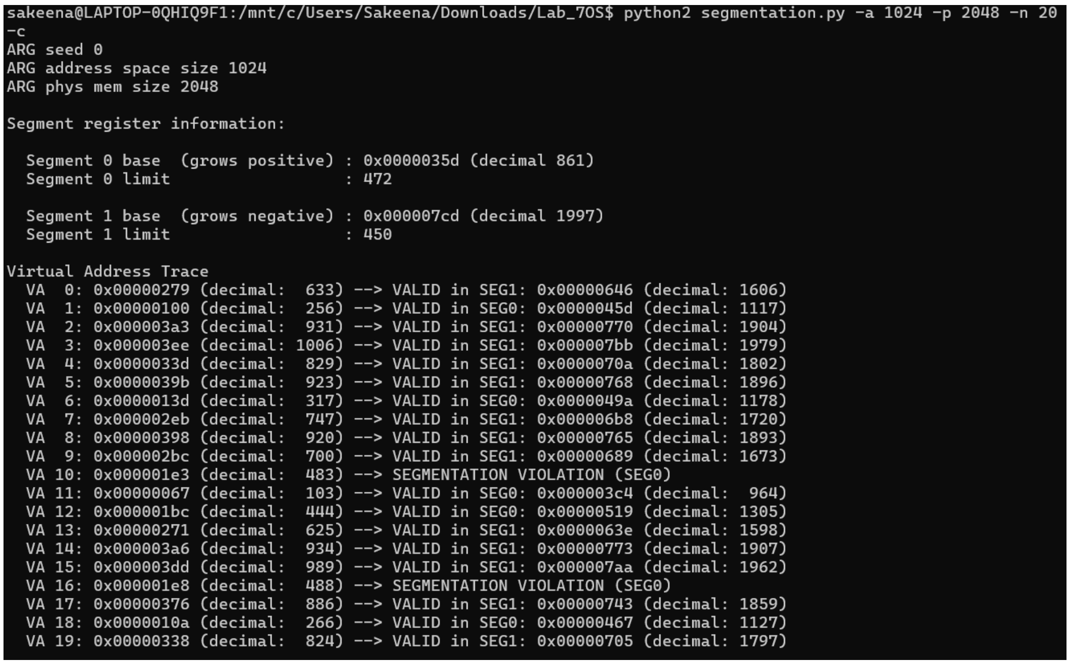
   Clearly, we can demonstrate it by:


```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 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)
```

4. The validity rate increases when the limit and address space size are lowered, as we have observed in numerous instances. Now, the limit should be roughly equivalent to 90% of the address space size if we want approximately 90% of the virtual addresses generated to be valid.
   Now picture a 1024-byte address space.

After that, segment 0: 0-511, segment 1: 511-1023 are used.
The overall limit should essentially cover 90% of 1024, then around 45% of 1024 for each segment taken separately,
45 percent of 1024 is almost 460 (460.8).
Decide that the size of the physical memory is 2048 bytes, then limit 1 = 460 and base 1 = 0, Base 2 is 2048, while Limit 2 is 460.

Let's check the 20 virtual addresses that were generated.



```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 segmentation.py -a 1024 -p 2048 -n 20
-c
ARG seed 0
ARG address space size 1024
ARG phys mem size 2048

Segment register information:

  Segment 0 base  (grows positive) : 0x0000035d (decimal 861)
  Segment 0 limit                  : 472

  Segment 1 base  (grows negative) : 0x000007cd (decimal 1997)
  Segment 1 limit                  : 450

Virtual Address Trace
  VA  0: 0x00000279 (decimal:  633) --> VALID in SEG1: 0x00000646 (decimal: 1606)
  VA  1: 0x00000100 (decimal:  256) --> VALID in SEG0: 0x0000045d (decimal: 1117)
  VA  2: 0x000003a3 (decimal:  931) --> VALID in SEG1: 0x00000770 (decimal: 1904)
  VA  3: 0x000003ee (decimal: 1006) --> VALID in SEG1: 0x000007bb (decimal: 1979)
  VA  4: 0x0000033d (decimal:  829) --> VALID in SEG1: 0x0000070a (decimal: 1802)
  VA  5: 0x0000039b (decimal:  923) --> VALID in SEG1: 0x00000768 (decimal: 1896)
  VA  6: 0x0000013d (decimal:  317) --> VALID in SEG0: 0x0000049a (decimal: 1178)
  VA  7: 0x000002eb (decimal:  747) --> VALID in SEG1: 0x000006b8 (decimal: 1720)
  VA  8: 0x00000398 (decimal:  920) --> VALID in SEG1: 0x00000765 (decimal: 1893)
  VA  9: 0x000002bc (decimal:  700) --> VALID in SEG1: 0x00000689 (decimal: 1673)
  VA 10: 0x000001e3 (decimal:  483) --> SEGMENTATION VIOLATION (SEG0)
  VA 11: 0x00000067 (decimal:  103) --> VALID in SEG0: 0x000003c4 (decimal:  964)
  VA 12: 0x000001bc (decimal:  444) --> VALID in SEG0: 0x00000519 (decimal: 1305)
  VA 13: 0x00000271 (decimal:  625) --> VALID in SEG1: 0x0000063e (decimal: 1598)
  VA 14: 0x000003a6 (decimal:  934) --> VALID in SEG1: 0x00000773 (decimal: 1907)
  VA 15: 0x000003dd (decimal:  989) --> VALID in SEG1: 0x000007aa (decimal: 1962)
  VA 16: 0x000001e8 (decimal:  488) --> SEGMENTATION VIOLATION (SEG0)
  VA 17: 0x00000376 (decimal:  886) --> VALID in SEG1: 0x00000743 (decimal: 1859)
  VA 18: 0x0000010a (decimal:  266) --> VALID in SEG0: 0x00000467 (decimal: 1127)
  VA 19: 0x00000338 (decimal:  824) --> VALID in SEG1: 0x00000705 (decimal: 1797)
```

Here, we see that out of 20, 18 are valid i.e. exactly 90% of addresses.

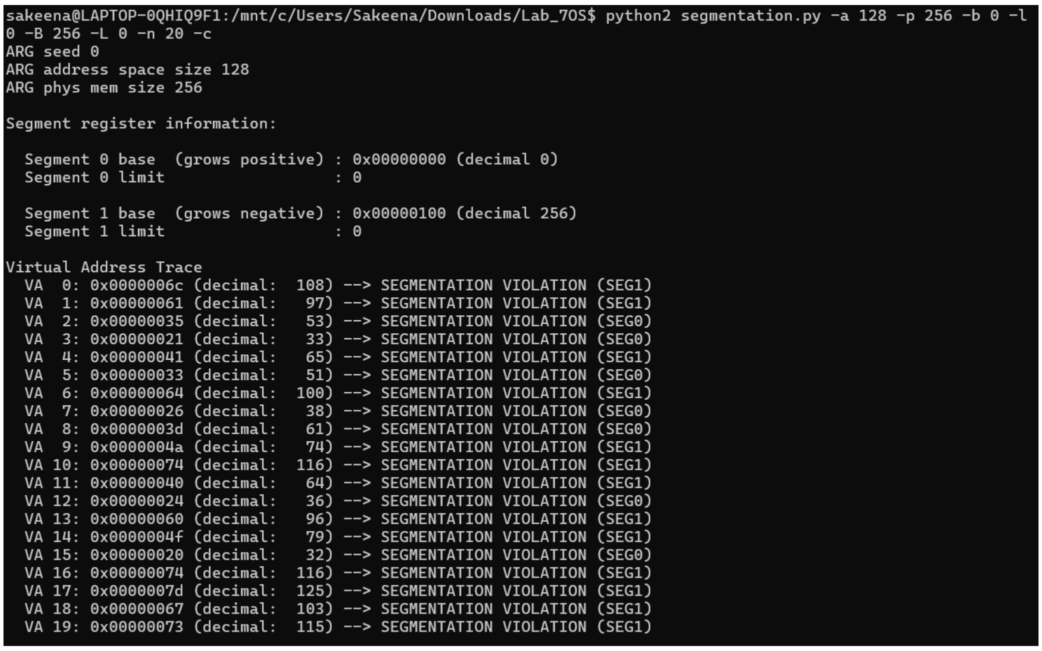
5. Well, in essence, we want to force valid addresses to either be negative or larger than the physical memory size. Nevertheless, since the programme will generate some arbitrary limit numbers, we are unable to specify the limit to be negative.

   The limit was therefore set to 0. Use -a 128 -p 256 -b 0 -l 0 -B 256 -L 0.

   Segment-0's lowest valid address is 0+0-1 = 1.

   Segment-1's lowest valid address is 256-0+1, or 257.

Hence, none of the addresses will work;



```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 segmentation.py -a 128 -p 256 -b 0 -l
0 -B 256 -L 0 -n 20 -c
ARG seed 0
ARG address space size 128
ARG phys mem size 256

Segment register information:

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

  Segment 1 base  (grows negative) : 0x00000100 (decimal 256)
  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)
  VA 10: 0x00000074 (decimal:  116) --> SEGMENTATION VIOLATION (SEG1)
  VA 11: 0x00000040 (decimal:   64) --> SEGMENTATION VIOLATION (SEG1)
  VA 12: 0x00000024 (decimal:   36) --> SEGMENTATION VIOLATION (SEG0)
  VA 13: 0x00000060 (decimal:   96) --> SEGMENTATION VIOLATION (SEG1)
  VA 14: 0x0000004f (decimal:   79) --> SEGMENTATION VIOLATION (SEG1)
  VA 15: 0x00000020 (decimal:   32) --> SEGMENTATION VIOLATION (SEG0)
  VA 16: 0x00000074 (decimal:  116) --> SEGMENTATION VIOLATION (SEG1)
  VA 17: 0x0000007d (decimal:  125) --> SEGMENTATION VIOLATION (SEG1)
  VA 18: 0x00000067 (decimal:  103) --> SEGMENTATION VIOLATION (SEG1)
  VA 19: 0x00000073 (decimal:  115) --> SEGMENTATION VIOLATION (SEG1)
```

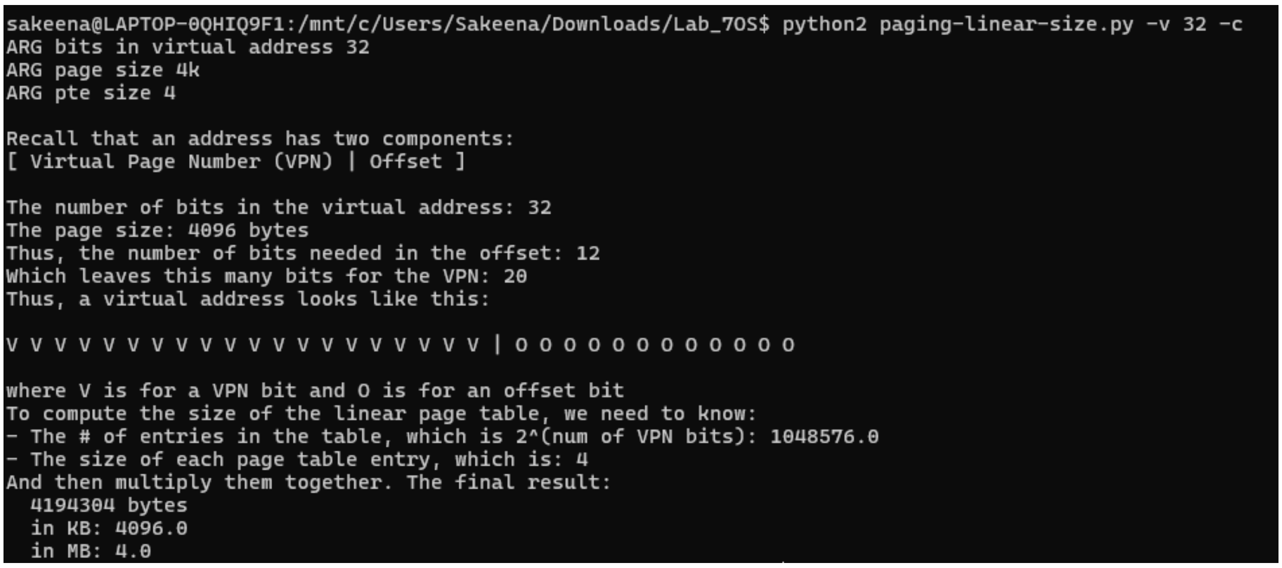
Q3) Number of bits in address' space **.**Page time doubles for every bit of address space that is added. As there are twice as many PTEs for every bit increase (Page table entries).

PTE count equals 2^(VPN bits). Since the number of offset bits remains constant, the number of VPN bits increases and the number of PTEs follows. When the bit count is 32, the page table size is 4 MB.

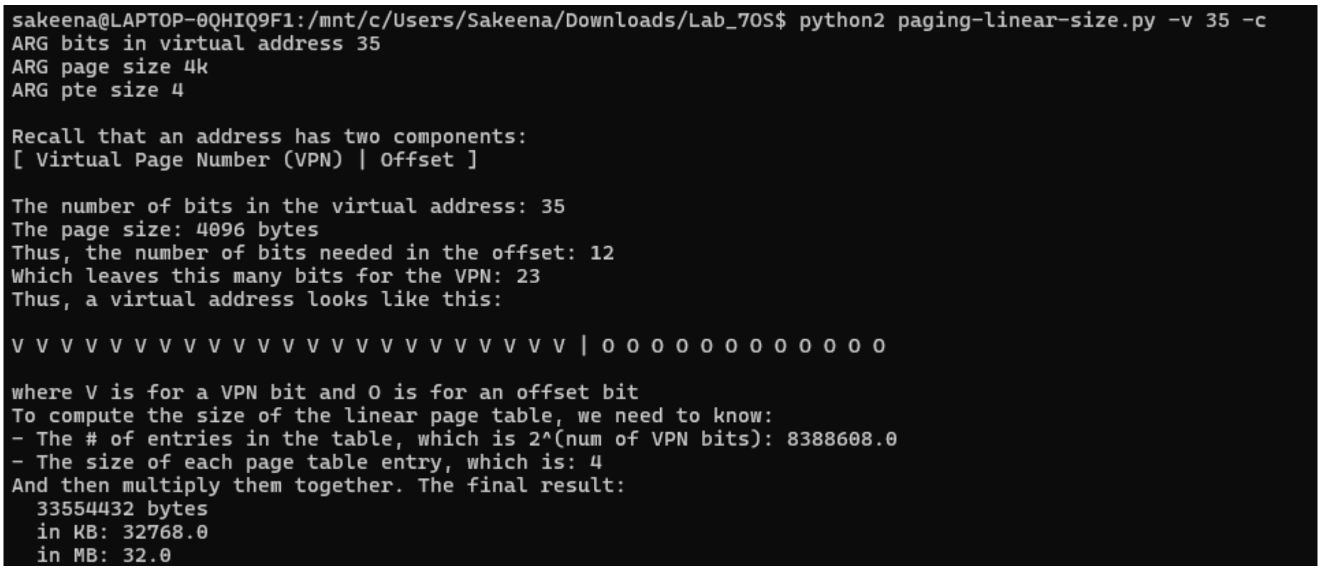
For bit count of 35, the page table size is 4MB*8 = 32MB.

For bit count of 64MB, the page table size is 4MB*16 = 64MB.

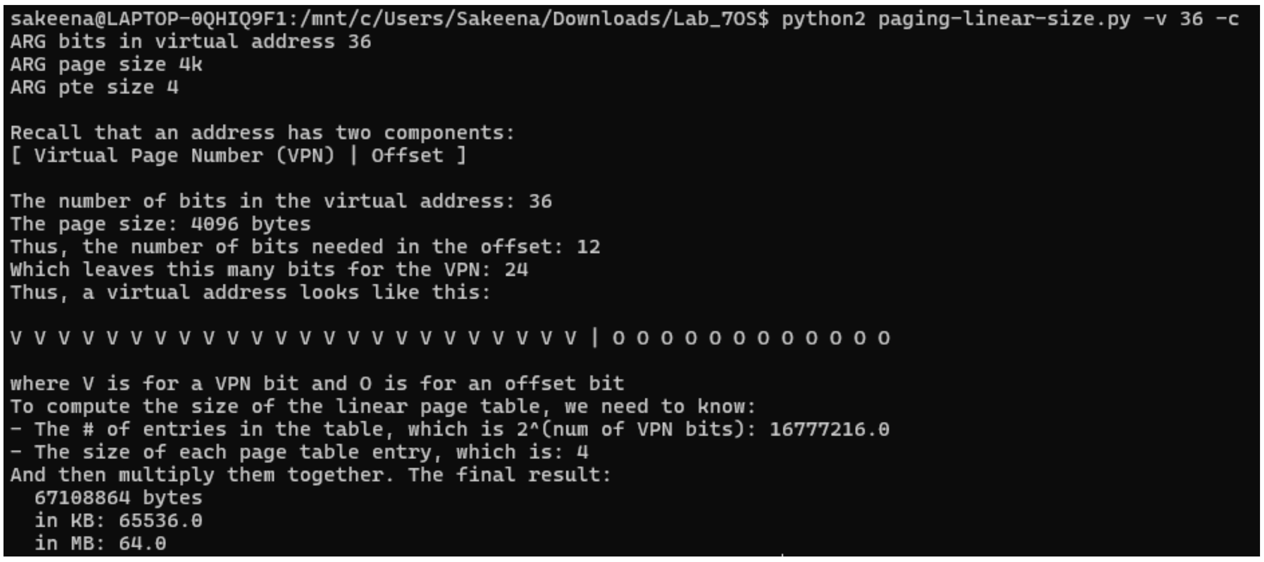
The size of the page table increases by =8 times from 32-35-3 bits, and so on.



```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 paging-linear-size.py -v 32 -c
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
```



```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 paging-linear-size.py -v 35 -c
ARG bits in virtual address 35
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: 35
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 23
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 | 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): 8388608.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  33554432 bytes
  in KB: 32768.0
  in MB: 32.0
```



```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 paging-linear-size.py -v 36 -c
ARG bits in virtual address 36
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: 36
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 24
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 | 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): 16777216.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  67108864 bytes
  in KB: 65536.0
  in MB: 64.0
```

Different Page size:

The quantity of PTEs decreases when the page size decreases but the address space remains constant.

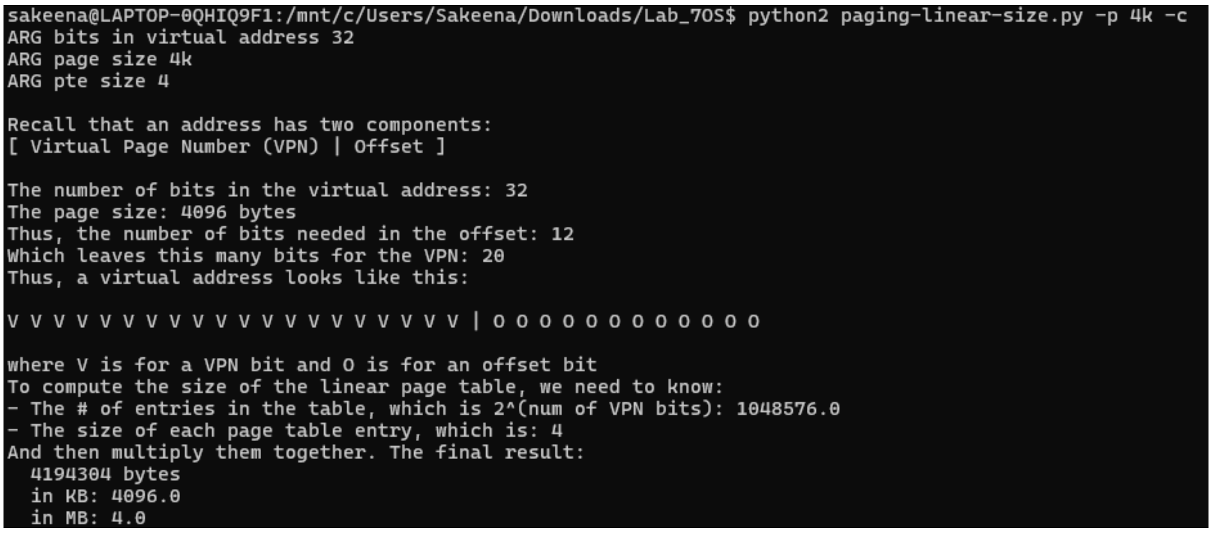
No. of PTEs = Size of Address Space/Size of Page

The page table size increases/decreases by two times for every two times decrease/increase in page size.

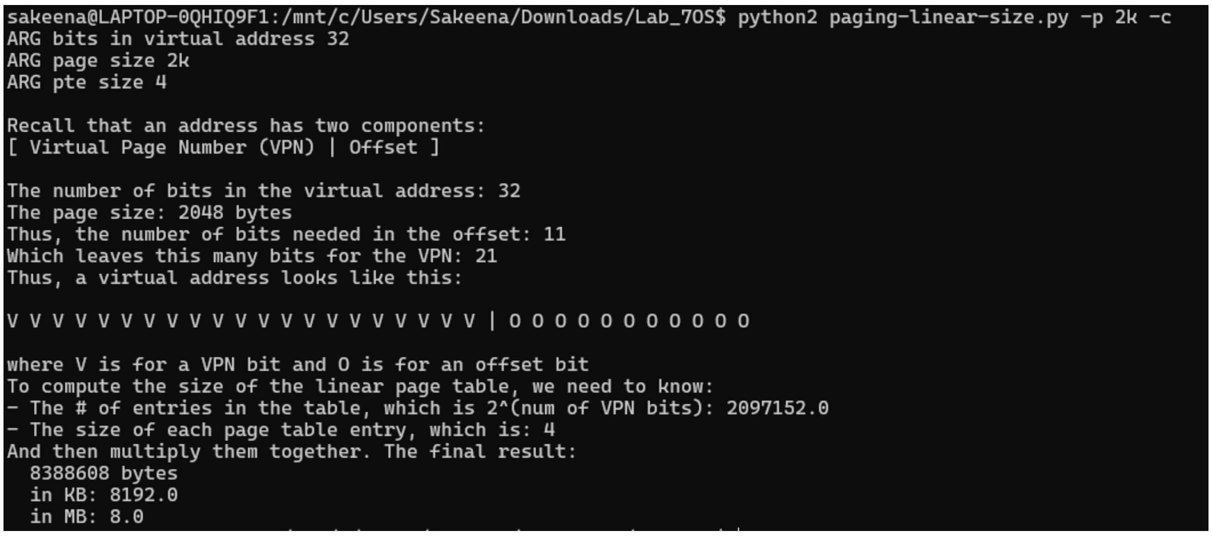
For page size = 1K, Page table size = 16 MB

For page size = 2k, Page table size = 8 MB

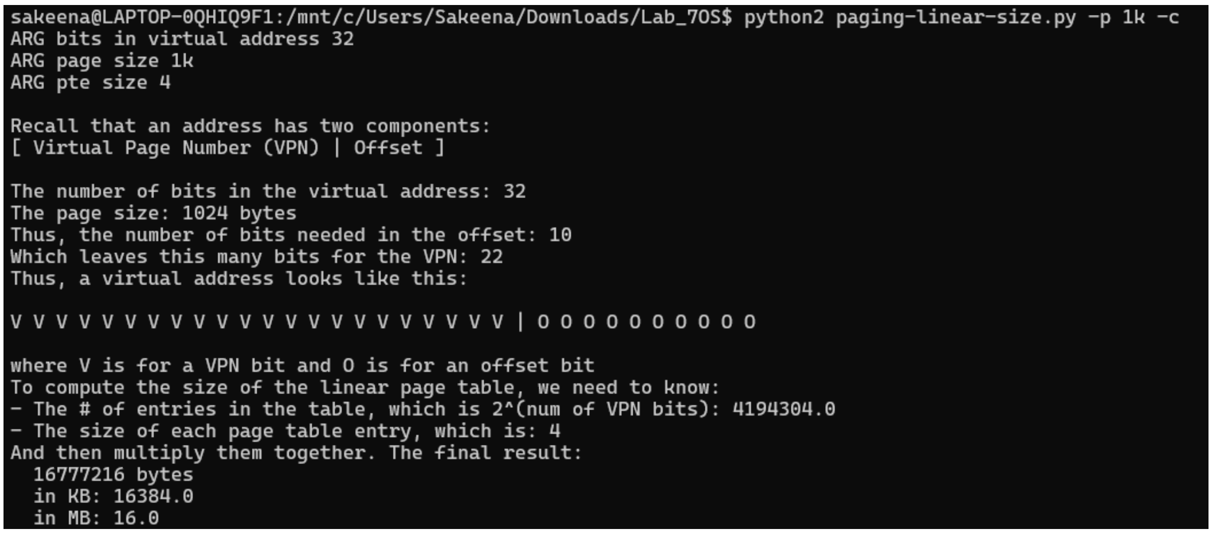
For page size = 4k, Page table size = 4 MB



```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 paging-linear-size.py -p 4k -c
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
```



```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 paging-linear-size.py -p 2k -c
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
```

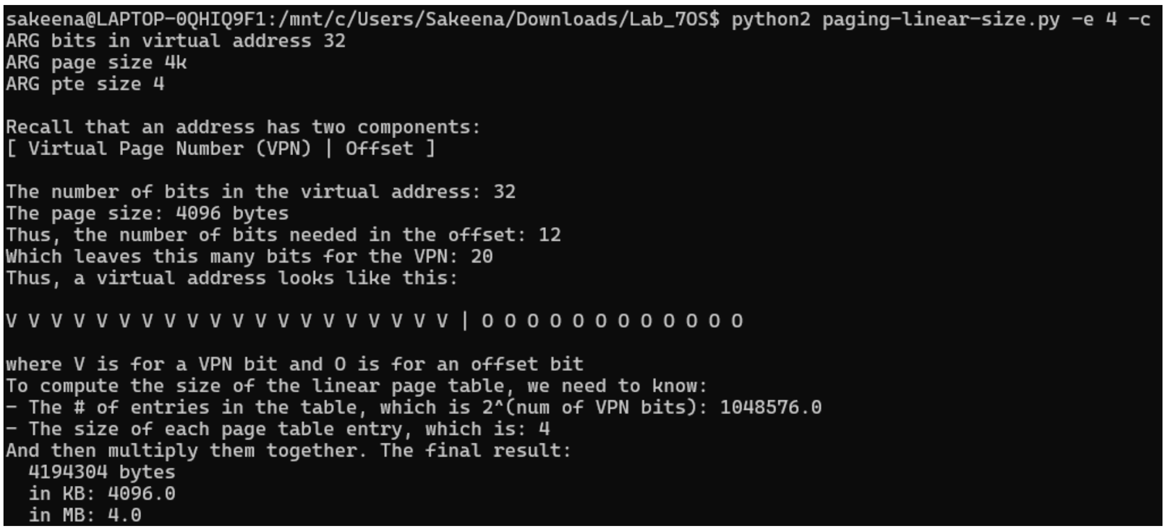


```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 paging-linear-size.py -p 1k -c
ARG bits in virtual address 32
ARG page size 1k
ARG pte size 4

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

The number of bits in the virtual address: 32
The page size: 1024 bytes
Thus, the number of bits needed in the offset: 10
Which leaves this many bits for the VPN: 22
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 | 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): 4194304.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  16777216 bytes
  in KB: 16384.0
  in MB: 16.0
```

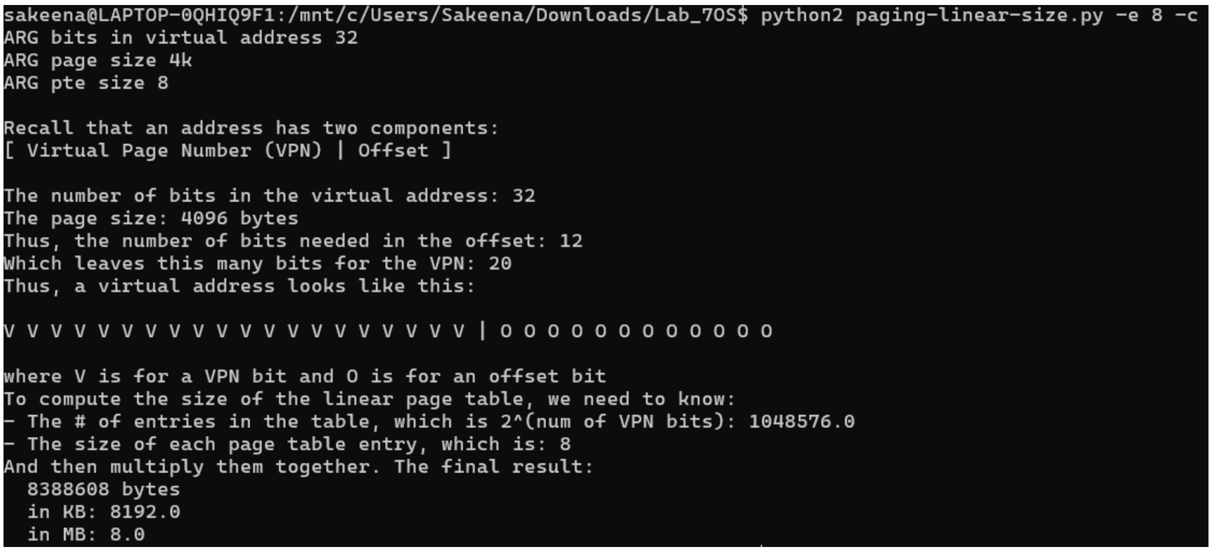
Different Page Entry Size

Page Table Size = Size of PTE * No. of PTEs

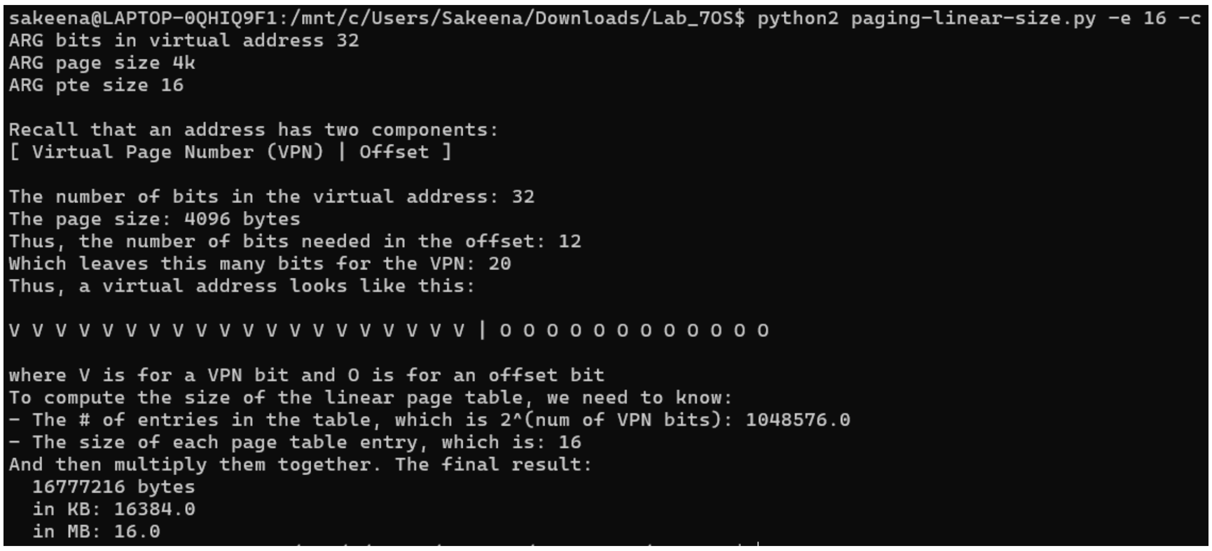
The size of the page table increases according to the amount of PTE.



```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 paging-linear-size.py -e 4 -c
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
```



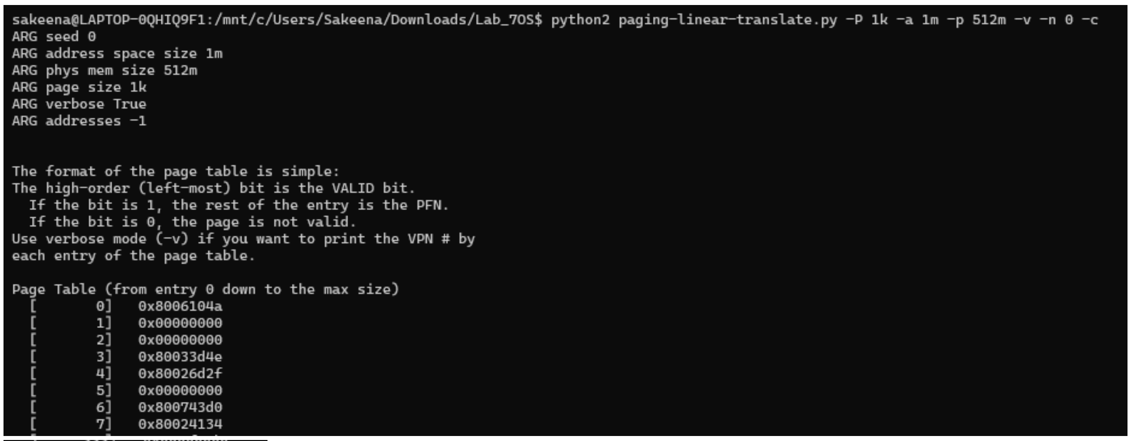
```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 paging-linear-size.py -e 8 -c
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 8

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
```



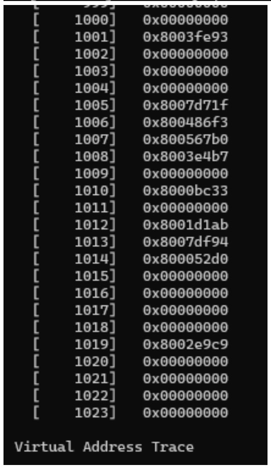
```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 paging-linear-size.py -e 16 -c
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 16

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
```

Q4)

1. The size of the Address space increases, Using -a 1m we have 1023 entries in the table.

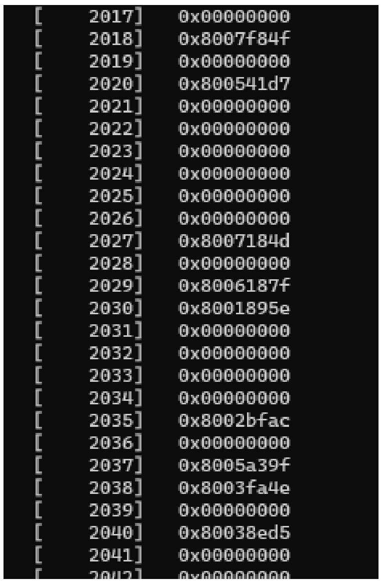


```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0 -c
ARG seed 0
ARG address space size 1m
ARG phys mem size 512m
ARG page size 1k
ARG verbose True
ARG addresses -1


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]   0x8006104a
  [       1]   0x00000000
  [       2]   0x00000000
  [       3]   0x80033d4e
  [       4]   0x80026d2f
  [       5]   0x00000000
  [       6]   0x800743d0
  [       7]   0x80024134
```

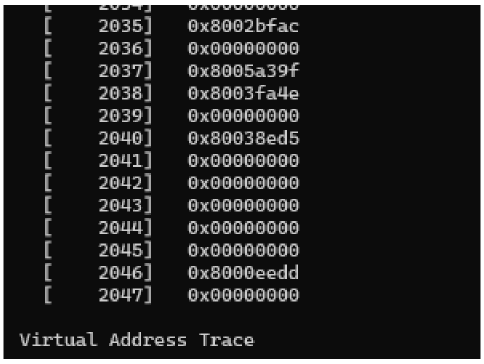


```
  [    1000]   0x00000000
  [    1001]   0x8003fe93
  [    1002]   0x00000000
  [    1003]   0x00000000
  [    1004]   0x00000000
  [    1005]   0x8007d71f
  [    1006]   0x800486f3
  [    1007]   0x800567b0
  [    1008]   0x8003e4b7
  [    1009]   0x00000000
  [    1010]   0x8000bc33
  [    1011]   0x00000000
  [    1012]   0x8001d1ab
  [    1013]   0x8007df94
  [    1014]   0x800052d0
  [    1015]   0x00000000
  [    1016]   0x00000000
  [    1017]   0x00000000
  [    1018]   0x00000000
  [    1019]   0x8002e9c9
  [    1020]   0x00000000
  [    1021]   0x00000000
  [    1022]   0x00000000
  [    1023]   0x00000000

 Virtual Address Trace
```

Using -a 2m we have 2047 entries in page table

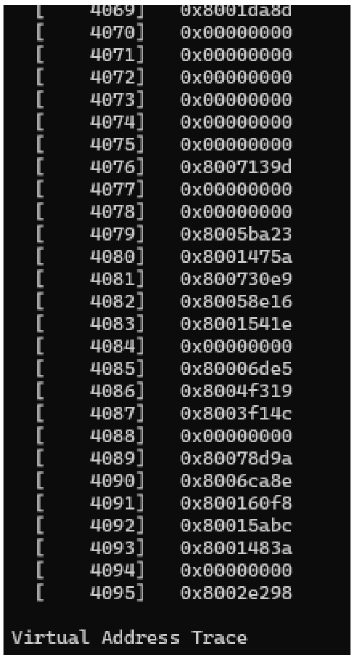


```
  [    2017]   0x00000000
  [    2018]   0x8007f84f
  [    2019]   0x00000000
  [    2020]   0x800541d7
  [    2021]   0x00000000
  [    2022]   0x00000000
  [    2023]   0x00000000
  [    2024]   0x00000000
  [    2025]   0x00000000
  [    2026]   0x00000000
  [    2027]   0x8007184d
  [    2028]   0x00000000
  [    2029]   0x8006187f
  [    2030]   0x8001895e
  [    2031]   0x00000000
  [    2032]   0x00000000
  [    2033]   0x00000000
  [    2034]   0x00000000
  [    2035]   0x8002bfac
  [    2036]   0x00000000
  [    2037]   0x8005a39f
  [    2038]   0x8003fa4e
  [    2039]   0x00000000
  [    2040]   0x80038ed5
  [    2041]   0x00000000
```



```
  [    2035]   0x8002bfac
  [    2036]   0x00000000
  [    2037]   0x8005a39f
  [    2038]   0x8003fa4e
  [    2039]   0x00000000
  [    2040]   0x80038ed5
  [    2041]   0x00000000
  [    2042]   0x00000000
  [    2043]   0x00000000
  [    2044]   0x00000000
  [    2045]   0x00000000
  [    2046]   0x8000eedd
  [    2047]   0x00000000

 Virtual Address Trace
```

Using -a 4m we have 4095 entries in table.



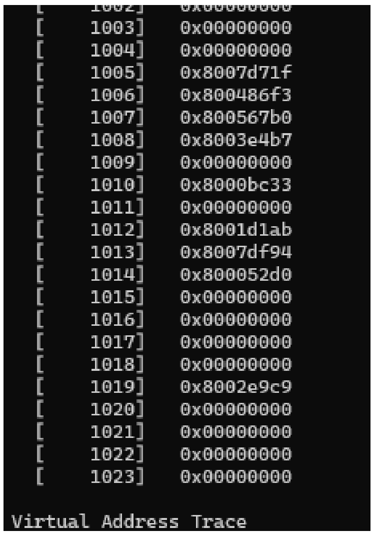
```
  [     4069]   0x8001da8d
  [     4070]   0x00000000
  [     4071]   0x00000000
  [     4072]   0x00000000
  [     4073]   0x00000000
  [     4074]   0x00000000
  [     4075]   0x00000000
  [     4076]   0x8007139d
  [     4077]   0x00000000
  [     4078]   0x00000000
  [     4079]   0x8005ba23
  [     4080]   0x8001475a
  [     4081]   0x800730e9
  [     4082]   0x80058e16
  [     4083]   0x8001541e
  [     4084]   0x00000000
  [     4085]   0x80006de5
  [     4086]   0x8004f319
  [     4087]   0x8003f14c
  [     4088]   0x00000000
  [     4089]   0x80078d9a
  [     4090]   0x8006ca8e
  [     4091]   0x800160f8
  [     4092]   0x80015abc
  [     4093]   0x8001483a
  [     4094]   0x00000000
  [     4095]   0x8002e298

Virtual Address Trace
```

Each entry in the page table is the same size, but as the address space gets bigger, the page table gets bigger as well.

Page Size Increases:

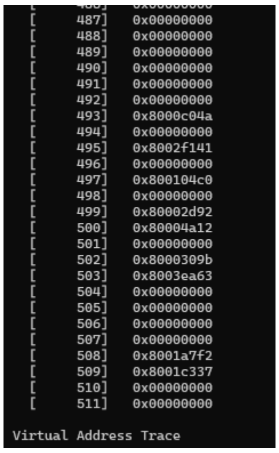
● The number of page table entries decreases and, consequently, the size of the page table as page size increases from 1k to 2k to 4k

1k:



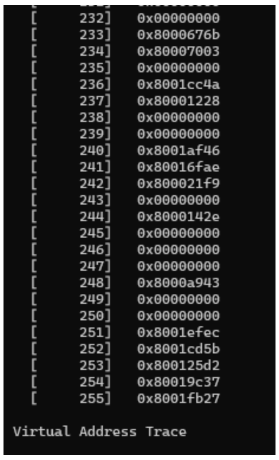
```
  [     1002]   0x00000000
  [     1003]   0x00000000
  [     1004]   0x00000000
  [     1005]   0x8007d71f
  [     1006]   0x800486f3
  [     1007]   0x800567b0
  [     1008]   0x8003e4b7
  [     1009]   0x00000000
  [     1010]   0x8000bc33
  [     1011]   0x00000000
  [     1012]   0x8001d1ab
  [     1013]   0x8007df94
  [     1014]   0x800052d0
  [     1015]   0x00000000
  [     1016]   0x00000000
  [     1017]   0x00000000
  [     1018]   0x00000000
  [     1019]   0x8002e9c9
  [     1020]   0x00000000
  [     1021]   0x00000000
  [     1022]   0x00000000
  [     1023]   0x00000000

Virtual Address Trace
```

2k:



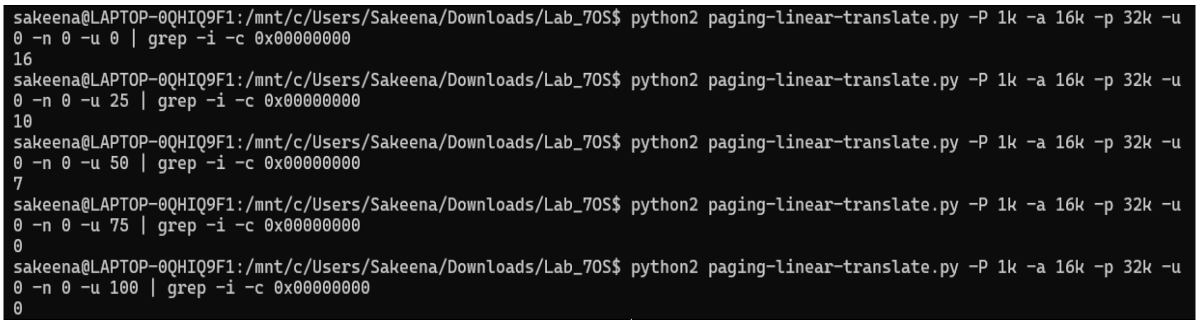
```
  [      487]   0x00000000
  [      488]   0x00000000
  [      489]   0x00000000
  [      490]   0x00000000
  [      491]   0x00000000
  [      492]   0x00000000
  [      493]   0x8000c04a
  [      494]   0x00000000
  [      495]   0x8002f141
  [      496]   0x00000000
  [      497]   0x800104c0
  [      498]   0x00000000
  [      499]   0x80002d92
  [      500]   0x80004a12
  [      501]   0x00000000
  [      502]   0x8000309b
  [      503]   0x8003ea63
  [      504]   0x00000000
  [      505]   0x00000000
  [      506]   0x00000000
  [      507]   0x00000000
  [      508]   0x8001a7f2
  [      509]   0x8001c337
  [      510]   0x00000000
  [      511]   0x00000000

Virtual Address Trace
```

4k:



```
  [      232]   0x00000000
  [      233]   0x8000676b
  [      234]   0x80007003
  [      235]   0x00000000
  [      236]   0x8001cc4a
  [      237]   0x80001228
  [      238]   0x00000000
  [      239]   0x00000000
  [      240]   0x8001af46
  [      241]   0x80016fae
  [      242]   0x800021f9
  [      243]   0x00000000
  [      244]   0x8000142e
  [      245]   0x00000000
  [      246]   0x00000000
  [      247]   0x00000000
  [      248]   0x8000a943
  [      249]   0x00000000
  [      250]   0x00000000
  [      251]   0x8001efec
  [      252]   0x8001cd5b
  [      253]   0x800125d2
  [      254]   0x80019c37
  [      255]   0x8001fb27

Virtual Address Trace
```

2. In essence, as u rises, more address space is being used.
   Now that **a=16k** and each page is **1k** in size. Therefore, there **are 16** entries total.
   In other words, when u increases from 0 to 100, more physical pages and, consequently, more page table entries, are allotted.
   Entries that are not assigned are indicated by the **number 0x00000000.**
   To determine the quantity of items in the unallocated page table, I used the grep tool



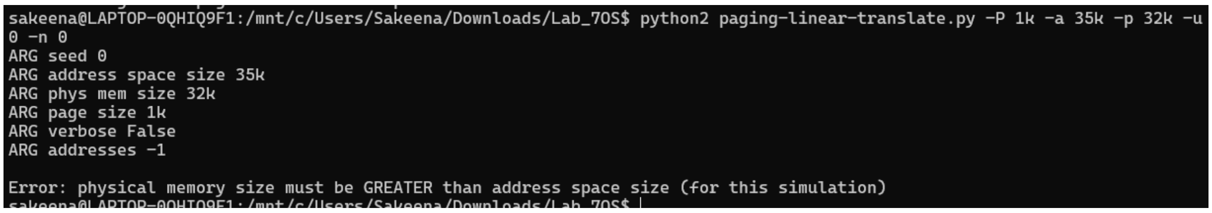
```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -u
0 -n 0 -u 0 | grep -i -c 0x00000000
16
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -u
0 -n 0 -u 25 | grep -i -c 0x00000000
10
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -u
0 -n 0 -u 50 | grep -i -c 0x00000000
7
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -u
0 -n 0 -u 75 | grep -i -c 0x00000000
0
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -u
0 -n 0 -u 100 | grep -i -c 0x00000000
0
```

   As we increase u, the number of page table entries that are unallocated falls.

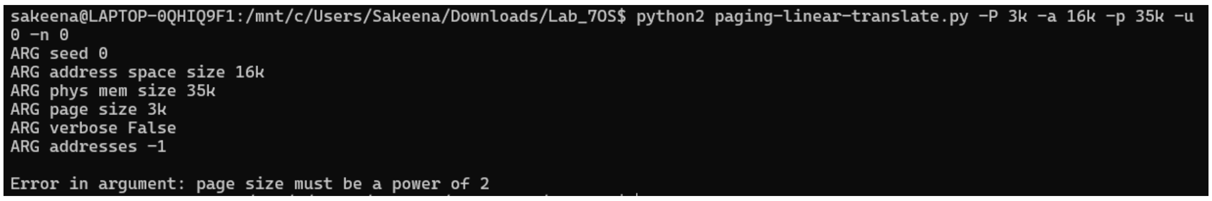
3. **First Parameter:** we can store 32 bytes for each process, we can run only 32 processes. (1024/32=32)
   Second Parameter: We can store 32 Kilo bytes for each process, we can run only 32 processes. (1m/32k=32)
   Third Parameter: Each process can store 256m of data. We can run only 2

processes (512/256=2)
Since each process can only consume 32 bytes of memory, at first it appears utterly irrational.

4.
The program's coding imposes the following restrictions on us:
● The size and bounds of the address space and physical memory must be positive.
● The size of a page, the address space, and the actual memory.
● The size of the address space and physical memory must double the size of the page.
● The minimum page size is 2.
The software will no longer function if any of the aforementioned conditions are not met.



```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 paging-linear-translate.py -P 1k -a 35k -p 32k -u
0 -n 0
ARG seed 0
ARG address space size 35k
ARG phys mem size 32k
ARG page size 1k
ARG verbose False
ARG addresses -1

Error: physical memory size must be GREATER than address space size (for this simulation)
```



```
sakeena@LAPTOP-0QHIQ9F1:/mnt/c/Users/Sakeena/Downloads/Lab_7OS$ python2 paging-linear-translate.py -P 3k -a 16k -p 35k -u
0 -n 0
ARG seed 0
ARG address space size 16k
ARG phys mem size 35k
ARG page size 3k
ARG verbose False
ARG addresses -1

Error in argument: page size must be a power of 2
```