CS314 - Laboratory 7

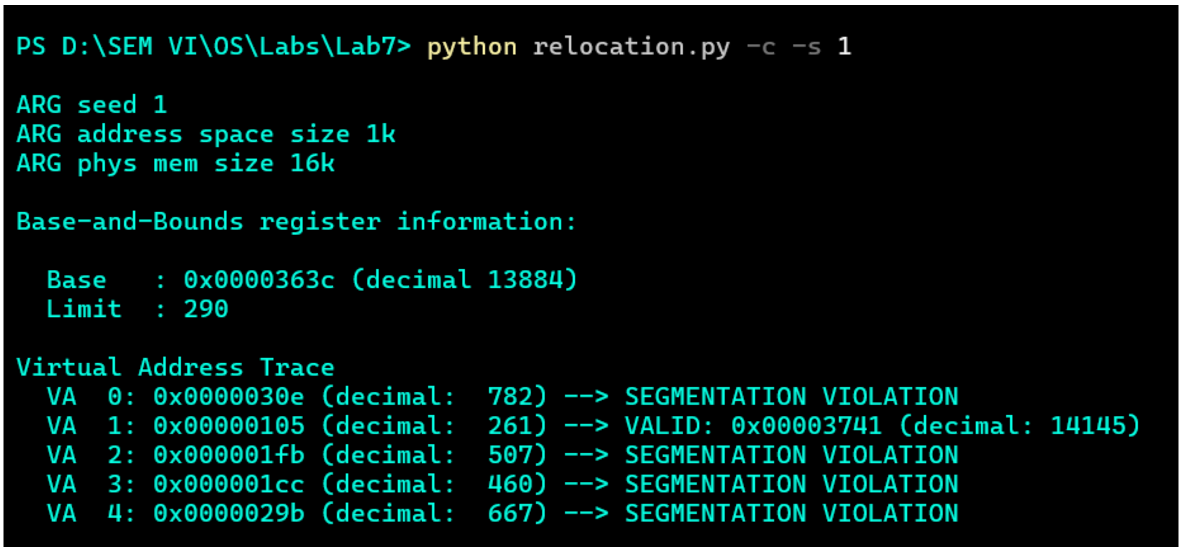
Hrishikesh Ravindra Karande

210010020

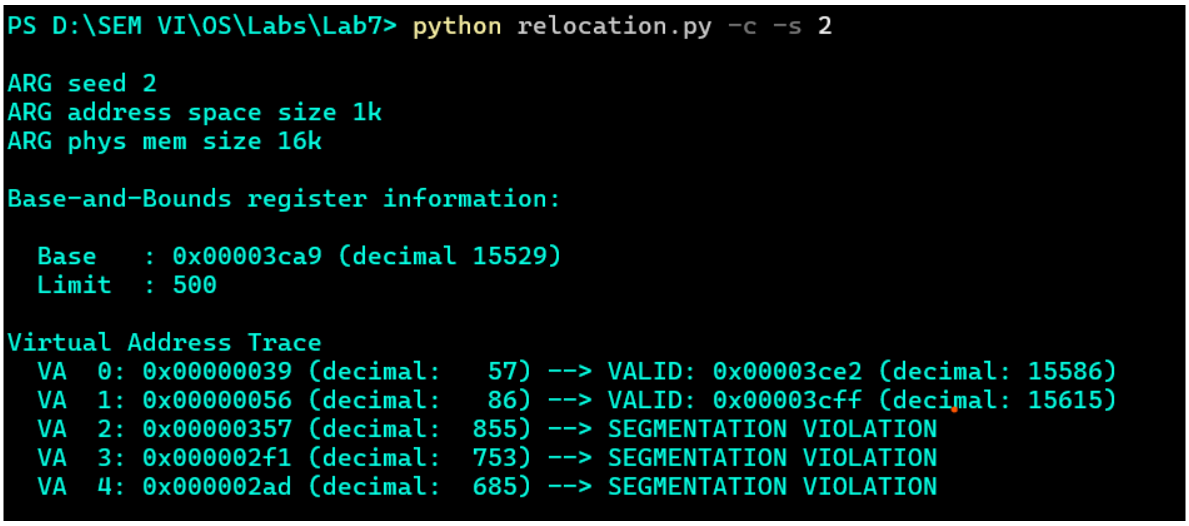
Part1: reloacation.py

Q1] Run with seeds 1, 2 and 3, and compute whether each virtual address generated by the process is in or out of bounds. If in bounds, compute the translation.

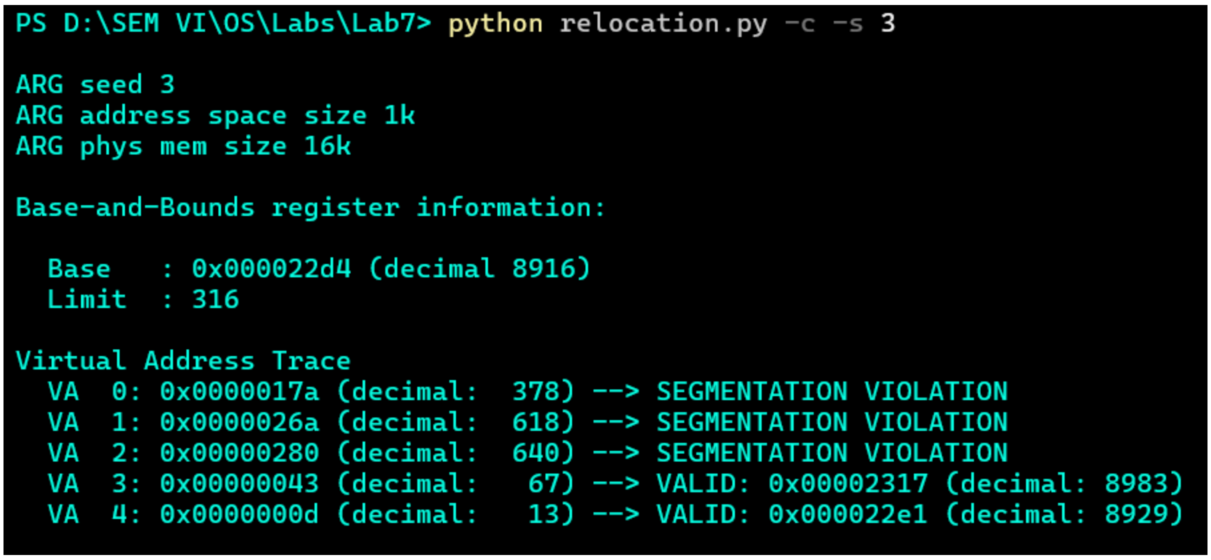
A]



```
PS D:\SEM VI\OS\Labs\Lab7> python relocation.py -c -s 1

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
```



```
PS D:\SEM VI\OS\Labs\Lab7> python relocation.py -c -s 2

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
```



```
PS D:\SEM VI\OS\Labs\Lab7> python relocation.py -c -s 3

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)
```

B&B Address Translation: Discussion

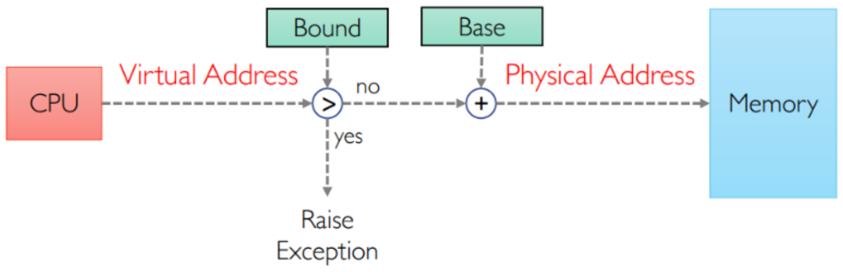
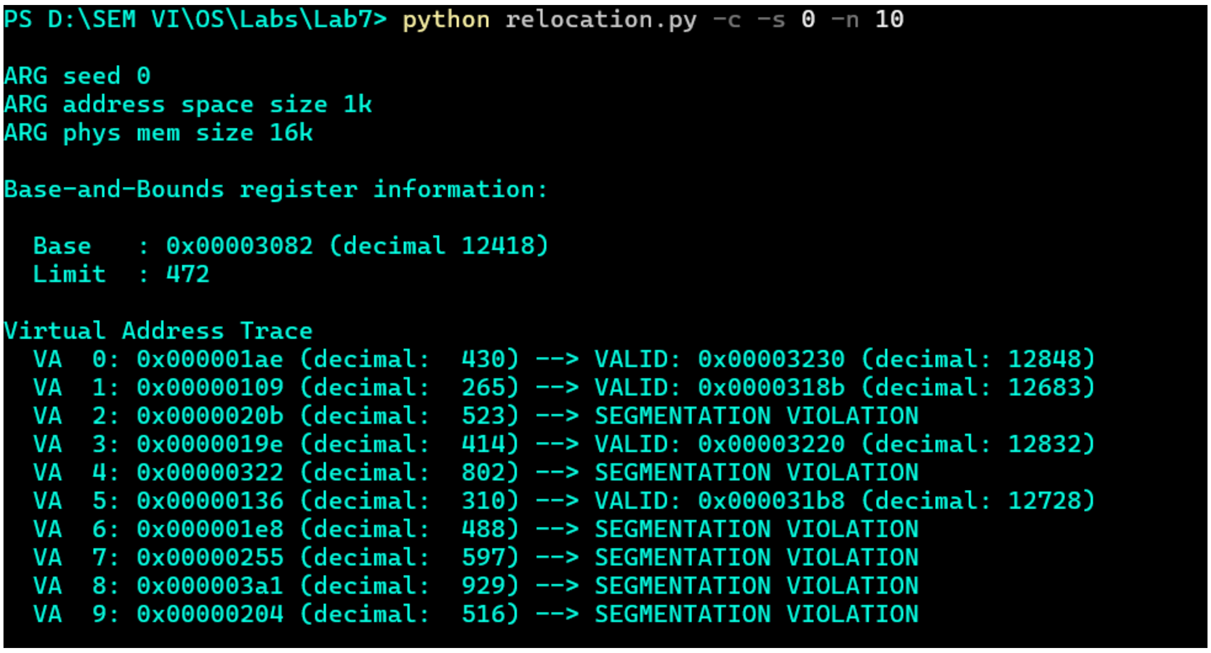


Fig1

Q2] Run with these flags: −s 0 −n 10. What value do you have set −l (the bounds register) to in order to ensure that all the generated virtual addresses are within bounds?


```
PS D:\SEM VI\OS\Labs\Lab7> python relocation.py -c -s 0 -n 10

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
```

From the image above, we observe that the maximum value of the virtual address allotted is 929. So setting the $\boxed{\text{limit value >= 930 (using −l 930)}}$ solves the problem of Segmentation violation for all addresses.

Q3] Run with these flags: −s 1 −n 10 −l 100. What is the maximum value that base can be set to, such that the address space still fits into physical memory in its entirety?

A] Total Physical Memory Size = 16k
Bound value = 100

To avoid any segmentation fault and valid address space

Base value + Bound value <= Total Physical Memory size

Therefore, Base value = Total Physical value – Bound Value

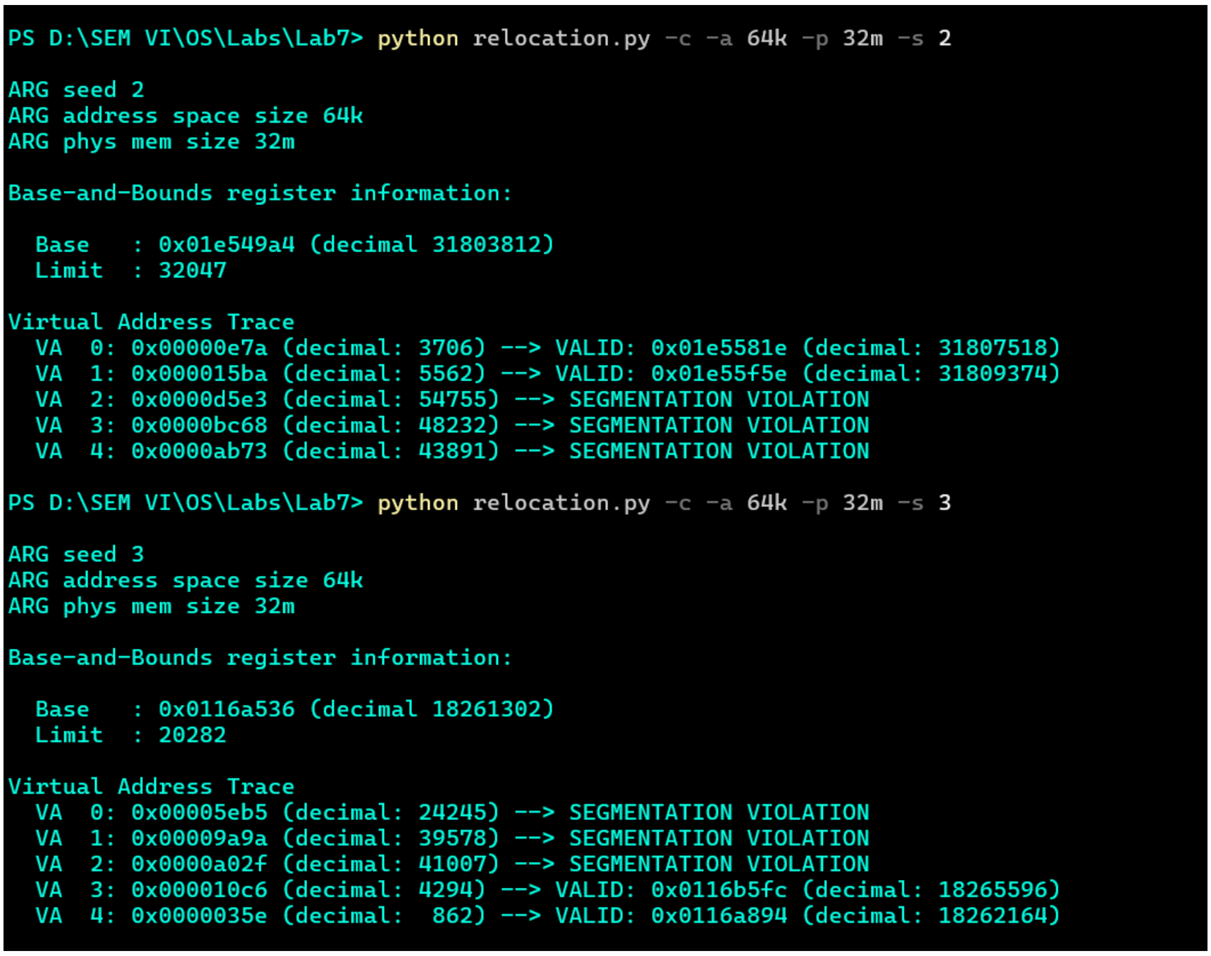
Base value = 16k – 100

Base Value = 16*(1024)-100

$\boxed{\text{Base Value = 16,284}}$

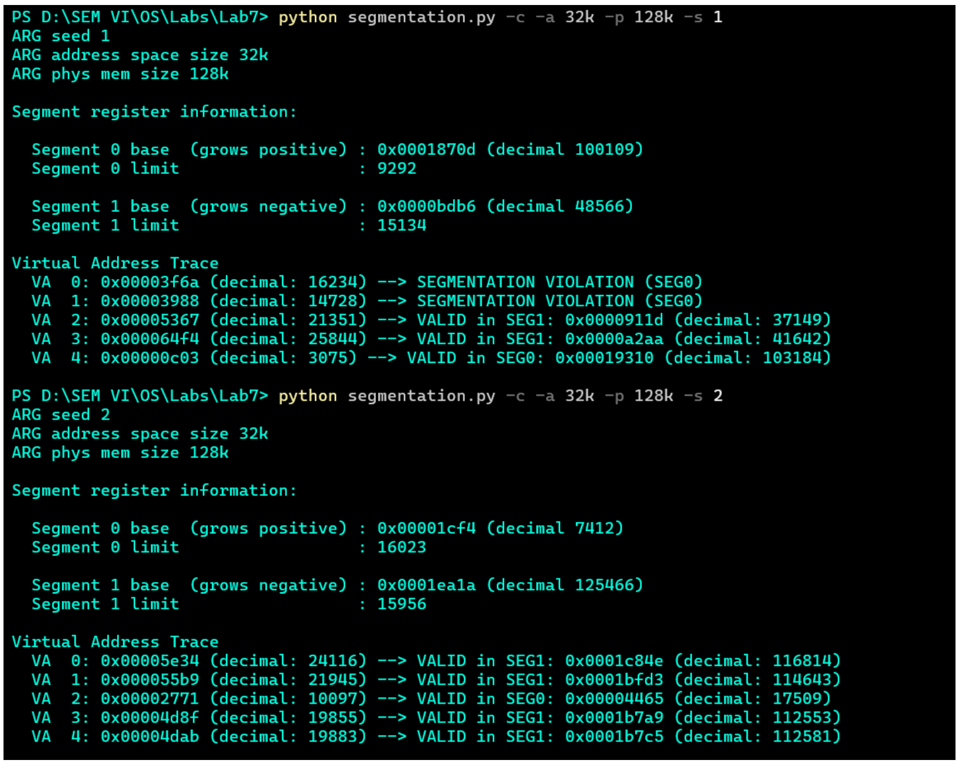
Q4] Run some of the same problems above, but with larger address spaces (−a) and physical memories (−p).

- Address space size = 64k and Physical Memory Size = 32m

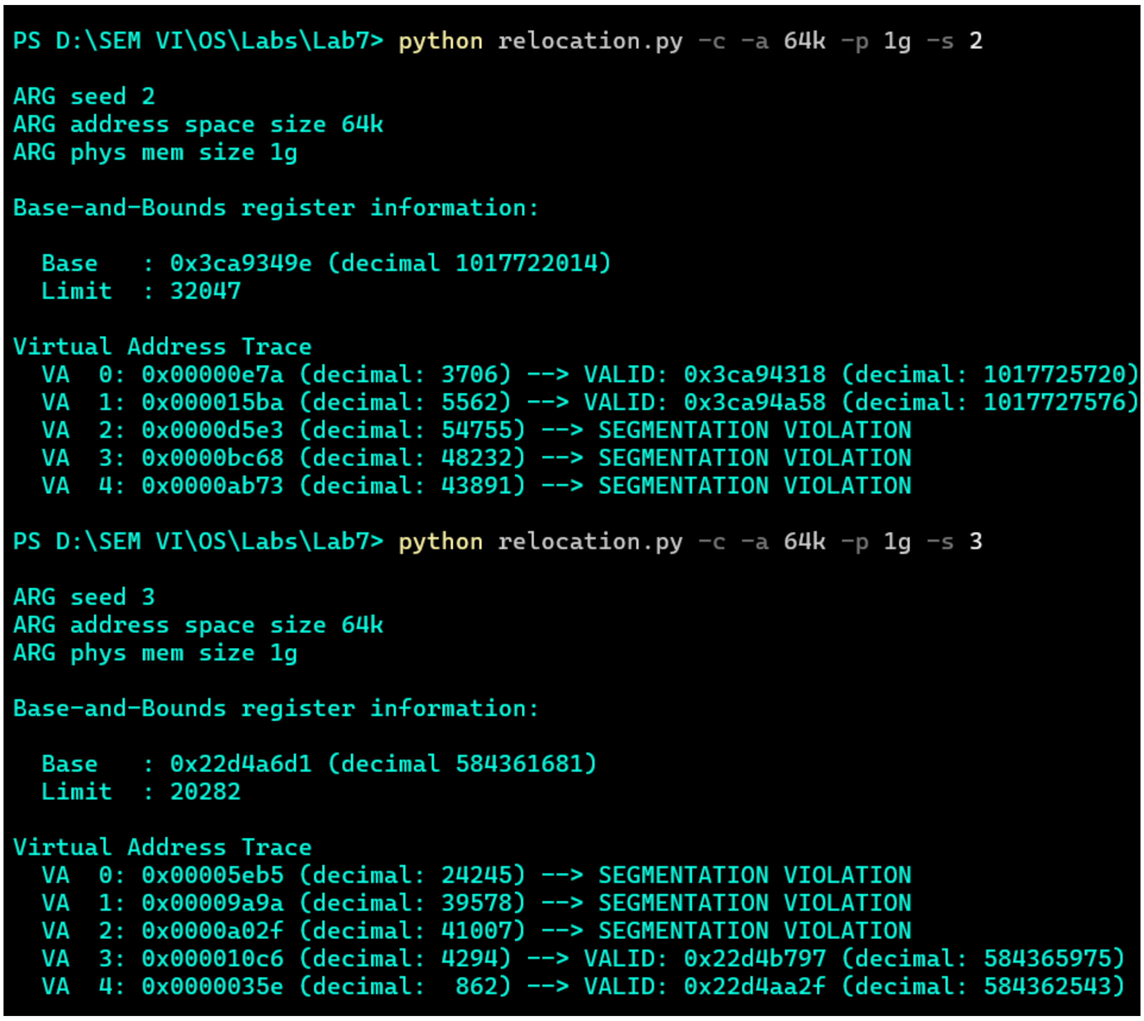


```
PS D:\SEM VI\OS\Labs\Lab7> python relocation.py -c -a 64k -p 32m -s 2

ARG seed 2
ARG address space size 64k
ARG phys mem size 32m

Base-and-Bounds register information:

  Base   : 0x01e549a4 (decimal 31803812)
  Limit  : 32047

Virtual Address Trace
  VA  0: 0x00000e7a (decimal: 3706) --> VALID: 0x01e5581e (decimal: 31807518)
  VA  1: 0x000015ba (decimal: 5562) --> VALID: 0x01e55f5e (decimal: 31809374)
  VA  2: 0x0000d5e3 (decimal: 54755) --> SEGMENTATION VIOLATION
  VA  3: 0x0000bc68 (decimal: 48232) --> SEGMENTATION VIOLATION
  VA  4: 0x0000ab73 (decimal: 43891) --> SEGMENTATION VIOLATION

PS D:\SEM VI\OS\Labs\Lab7> python relocation.py -c -a 64k -p 32m -s 3

ARG seed 3
ARG address space size 64k
ARG phys mem size 32m

Base-and-Bounds register information:

  Base   : 0x0116a536 (decimal 18261302)
  Limit  : 20282

Virtual Address Trace
  VA  0: 0x00005eb5 (decimal: 24245) --> SEGMENTATION VIOLATION
  VA  1: 0x00009a9a (decimal: 39578) --> SEGMENTATION VIOLATION
  VA  2: 0x0000a02f (decimal: 41007) --> SEGMENTATION VIOLATION
  VA  3: 0x000010c6 (decimal: 4294) --> VALID: 0x0116b5fc (decimal: 18265596)
  VA  4: 0x0000035e (decimal:  862) --> VALID: 0x0116a894 (decimal: 18262164)
```

- Address space size = 64k and Physical Memory Size = 1g



```
PS D:\SEM VI\OS\Labs\Lab7> python segmentation.py -c -a 32k -p 128k -s 1
ARG seed 1
ARG address space size 32k
ARG phys mem size 128k

Segment register information:

  Segment 0 base  (grows positive) : 0x0001870d (decimal 100109)
  Segment 0 limit                  : 9292

  Segment 1 base  (grows negative) : 0x0000bdb6 (decimal 48566)
  Segment 1 limit                  : 15134

Virtual Address Trace
  VA  0: 0x00003f6a (decimal: 16234) --> SEGMENTATION VIOLATION (SEG0)
  VA  1: 0x00003988 (decimal: 14728) --> SEGMENTATION VIOLATION (SEG0)
  VA  2: 0x00005367 (decimal: 21351) --> VALID in SEG1: 0x0000911d (decimal: 37149)
  VA  3: 0x000064f4 (decimal: 25844) --> VALID in SEG1: 0x0000a2aa (decimal: 41642)
  VA  4: 0x00000c03 (decimal: 3075) --> VALID in SEG0: 0x00019310 (decimal: 103184)

PS D:\SEM VI\OS\Labs\Lab7> python segmentation.py -c -a 32k -p 128k -s 2
ARG seed 2
ARG address space size 32k
ARG phys mem size 128k

Segment register information:

  Segment 0 base  (grows positive) : 0x00001cf4 (decimal 7412)
  Segment 0 limit                  : 16023

  Segment 1 base  (grows negative) : 0x0001ea1a (decimal 125466)
  Segment 1 limit                  : 15956

Virtual Address Trace
  VA  0: 0x00005e34 (decimal: 24116) --> VALID in SEG1: 0x0001c84e (decimal: 116814)
  VA  1: 0x000055b9 (decimal: 21945) --> VALID in SEG1: 0x0001bfd3 (decimal: 114643)
  VA  2: 0x00002771 (decimal: 10097) --> VALID in SEG0: 0x00004465 (decimal: 17509)
  VA  3: 0x00004d8f (decimal: 19855) --> VALID in SEG1: 0x0001b7a9 (decimal: 112553)
  VA  4: 0x00004dab (decimal: 19883) --> VALID in SEG1: 0x0001b7c5 (decimal: 112581)
```

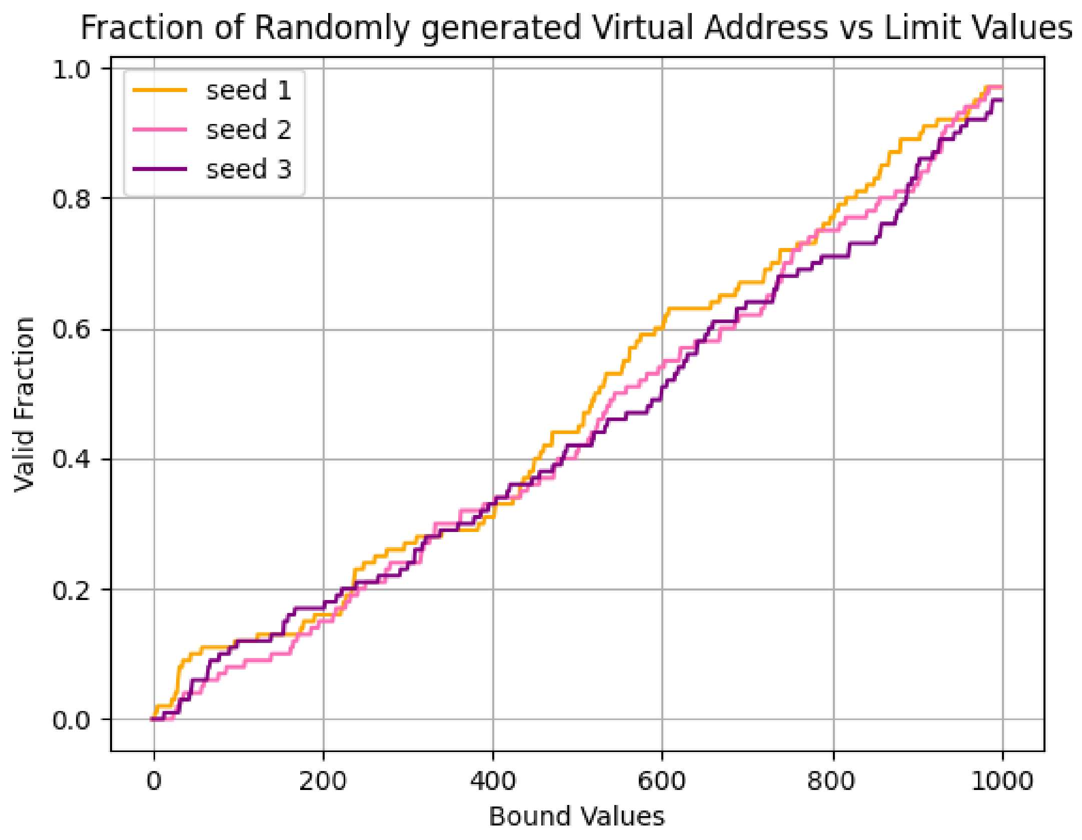


```
PS D:\SEM VI\OS\Labs\Lab7> python relocation.py -c -a 64k -p 1g -s 2

ARG seed 2
ARG address space size 64k
ARG phys mem size 1g

Base-and-Bounds register information:

  Base   : 0x3ca9349e (decimal 1017722014)
  Limit  : 32047

Virtual Address Trace
  VA  0: 0x00000e7a (decimal: 3706) --> VALID: 0x3ca94318 (decimal: 1017725720)
  VA  1: 0x000015ba (decimal: 5562) --> VALID: 0x3ca94a58 (decimal: 1017727576)
  VA  2: 0x0000d5e3 (decimal: 54755) --> SEGMENTATION VIOLATION
  VA  3: 0x0000bc68 (decimal: 48232) --> SEGMENTATION VIOLATION
  VA  4: 0x0000ab73 (decimal: 43891) --> SEGMENTATION VIOLATION

PS D:\SEM VI\OS\Labs\Lab7> python relocation.py -c -a 64k -p 1g -s 3

ARG seed 3
ARG address space size 64k
ARG phys mem size 1g

Base-and-Bounds register information:

  Base   : 0x22d4a6d1 (decimal 584361681)
  Limit  : 20282

Virtual Address Trace
  VA  0: 0x00005eb5 (decimal: 24245) --> SEGMENTATION VIOLATION
  VA  1: 0x00009a9a (decimal: 39578) --> SEGMENTATION VIOLATION
  VA  2: 0x0000a02f (decimal: 41007) --> SEGMENTATION VIOLATION
  VA  3: 0x000010c6 (decimal: 4294) --> VALID: 0x22d4b797 (decimal: 584365975)
  VA  4: 0x0000035e (decimal:  862) --> VALID: 0x22d4aa2f (decimal: 584362543)
```

Q5] What fraction of randomly-generated virtual addresses are valid, as a function of the value of the bounds register? Make a graph from running with different random seeds, with limit values ranging from 0 up to the maximum size of the address space.

A]

- Two files are attached in the zip file ./script.py and ./Q1_5_count.py, create an empty folder Data to create a dataset with different values of limits and seeds. For each seed we calculate the fraction for valid/(valid + segmentation).
- We observe that the relationship between the limit value and the % valid addresses is almost linear i.e %valid is directly proportional to limit value.
- The plot can be seen as below:



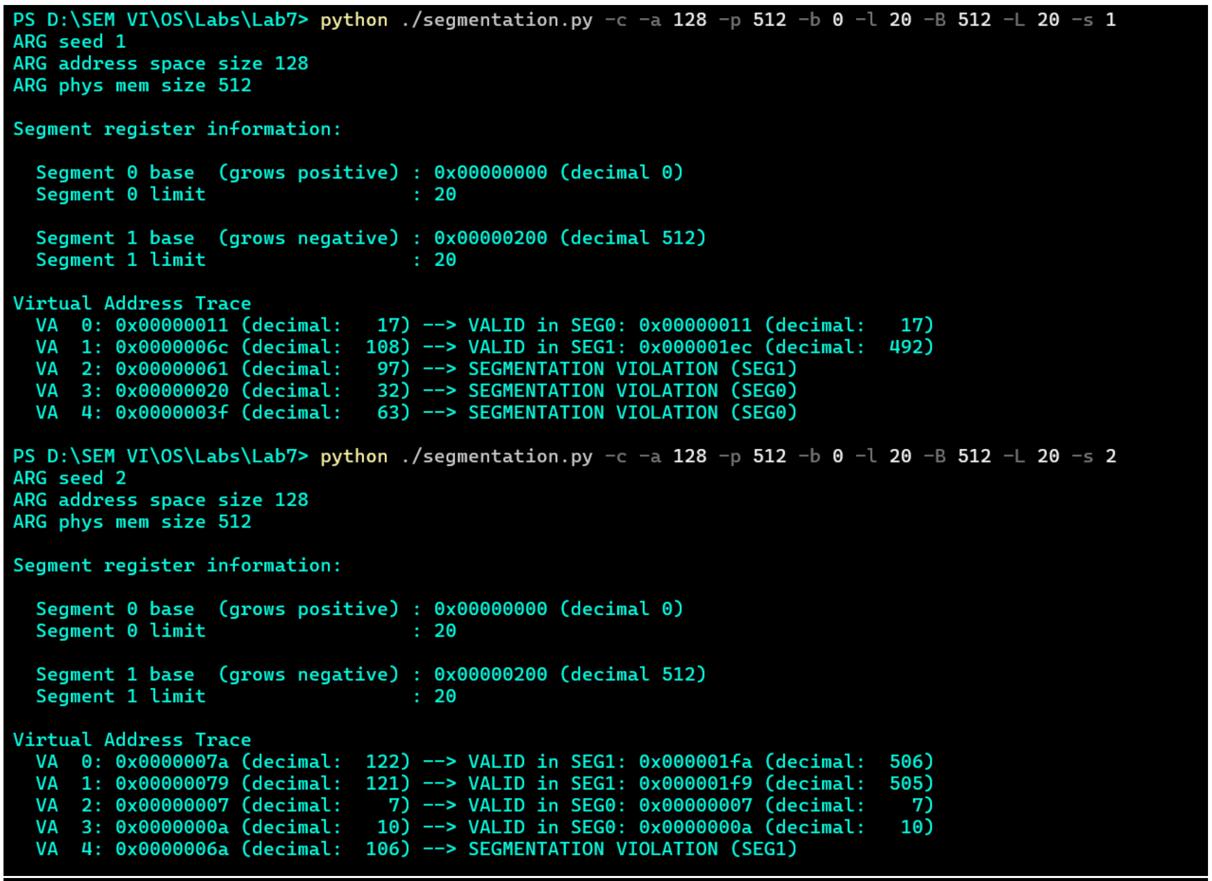
Part2 - segmentation.py The program segmentation.py allows you to see how address translations are performed in a system with segmentation

Q1. First let's use a tiny address space to translate some addresses. Here's a simple set of parameters with a few different random seeds; can you translate the addresses?

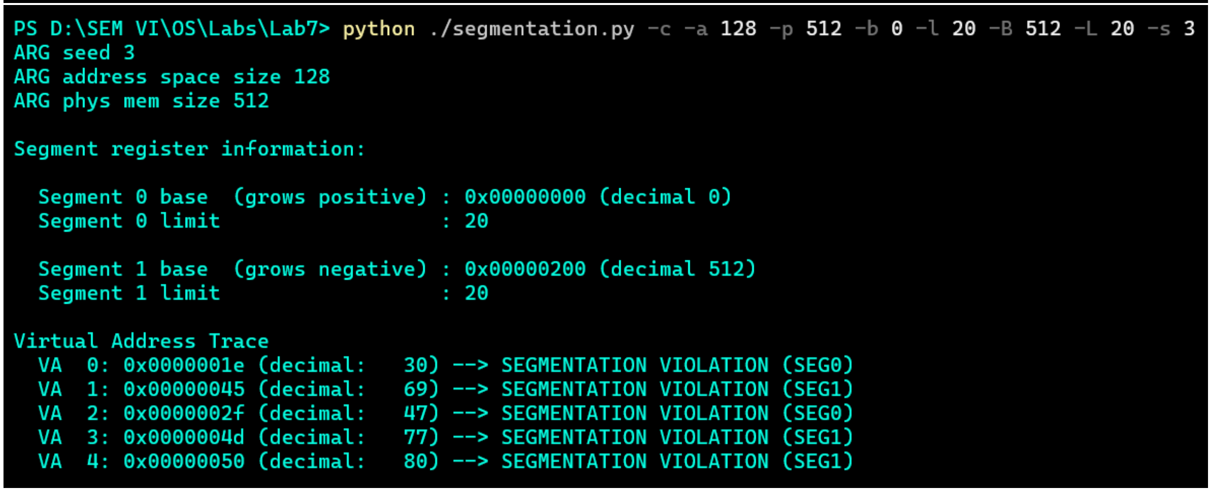
A] • For positive addressing we need to add the VA to Base address. To translate segment 0 addresses, simply add the virtual address value to the base address (here 0) of segment 0.

•Segment 1 grows in negative direction,(towards decreasing physical address) to translate segment 1 addresses:
If abs(VA-Address space size>=bound) -> Segmentation Fault
Else PA = base address + VA – address space size

• For example: in below screen shot, – VA0 (17) is a valid segment 0 address (since it is less than 20) – VA1 (108) is a valid segment 1 address (since base address = 512, address size = 128, so, physical address is 512 + 108 - 128 = 492)



```
PS D:\SEM VI\OS\Labs\Lab7> python ./segmentation.py -c -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1
ARG seed 1
ARG address space size 128
ARG phys mem size 512

Segment register information:

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

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

Virtual Address Trace
  VA  0: 0x00000011 (decimal:   17) --> 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)

PS D:\SEM VI\OS\Labs\Lab7> python ./segmentation.py -c -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2
ARG seed 2
ARG address space size 128
ARG phys mem size 512

Segment register information:

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

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

Virtual Address Trace
  VA  0: 0x0000007a (decimal:  122) --> 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)
```

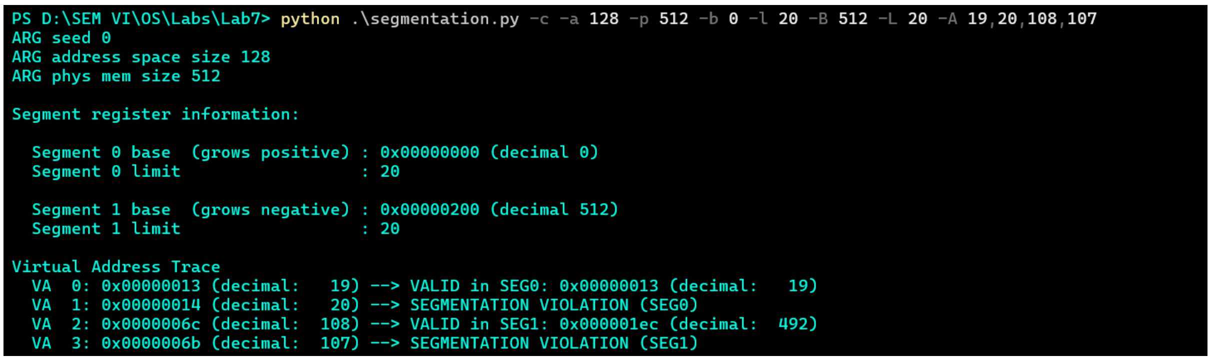


```
PS D:\SEM VI\OS\Labs\Lab7> python ./segmentation.py -c -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 3
ARG seed 3
ARG address space size 128
ARG phys mem size 512

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: 0x0000001e (decimal:   30) --> SEGMENTATION VIOLATION (SEG0)
  VA  1: 0x00000045 (decimal:   69) --> SEGMENTATION VIOLATION (SEG1)
  VA  2: 0x0000002f (decimal:   47) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x0000004d (decimal:   77) --> SEGMENTATION VIOLATION (SEG1)
  VA  4: 0x00000050 (decimal:   80) --> SEGMENTATION VIOLATION (SEG1)
```

Q2] Now, let’s see if we understand this tiny address space we’ve constructed (using the parameters from the question above). What is the highest legal virtual address in segment 0? What about the lowest legal virtual address in segment 1? What are the lowest and highest illegal addresses in this entire address space? Finally, would you run segmentation.py with −A flag to test if you are right?

- Segment 0 grows downwards so it’s range is (0,Base + Limit),
  Highest legal Virtual Address = 19.

- Segment 1 grows upwards negative direction so range is (Base,Base-Limit)
  Lowest legal Virtual Address = 128-20 = 108
- Lowest Illegal Address for Segment 0 = 20 (Just after the Legal address)
- Highest Illegal Address for Segment 1= 107 (Just before the Legal address)



```
PS D:\SEM VI\OS\Labs\Lab7> python .\segmentation.py -c -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -A 19,20,108,107
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: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492)
  VA  3: 0x0000006b (decimal:  107) --> SEGMENTATION VIOLATION (SEG1)
```

Q3] Let's say we have a tiny 16-byte address space in a 128-byte physical memory. What base and bounds would you set up so as to get the simulator to generate the following translation results for the specified address stream: valid, valid, violation, ... violation, valid, valid? Assume the following parameters: segmentation.py -a 16 -p 128 -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 --b0 ? --l0 ? --b1 ? --l1 ?

- We want the bound to be such that it should not violate for Virtual Address (decimal) >2 in segment 0 and for Virtual Address (decimal) >=126 for segment 1.
- base =0 and limit = 2 (Segment 0)
- base = 128 and limit = 2 (Segment 1 PA = VA + base – AS)

Q4]Assume we want to generate a problem where roughly 90% of the randomly-generated virtual addresses are valid (not segmentation violations). How should you configure the simulator to do so? Which parameters are important to getting this outcome?

A] For 90% of the randomly generated virtual addresses to be valid we need to ensure that these virtual address fall within 90% Address space.
len(Seg-1) + len(Seg0) = 0.9*Address Space
To satisfy this condition we will require to keep the base at 0 or max value but the bounds should be adjusted to cover 90% of address space.

Q5] Can you run the simulator such that no virtual addresses are valid? How?

A] For the virtual addresses to be invalid all the virtual address should be out of bound. As per Fig1 if the bound is less than the virtual address then seg fault should occur. Therefore we need to keep the bound as less as possible.

Keep the bound/Limit = 0.

Part3: The program paging-linear-size.py lets you figure out the size of a linear page table given a variety of input parameters. Compute how big a linear page table is with the characteristics such as different number of bits in the address space, different page size, different page table entry size. Explain your answers for various cases

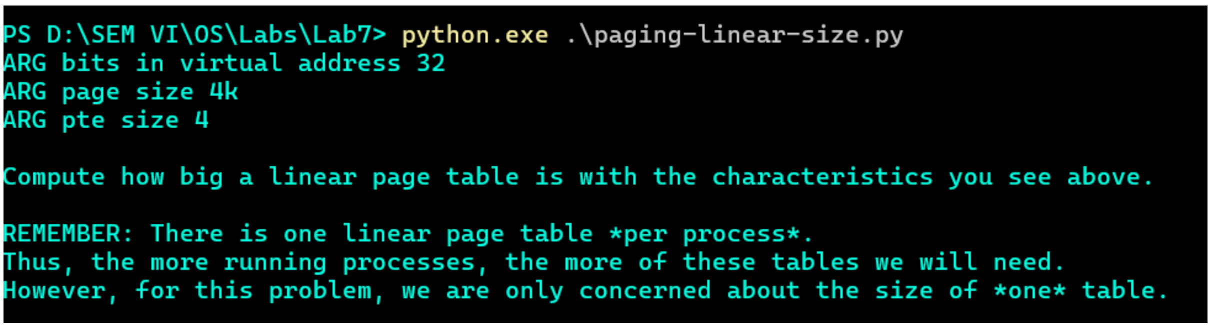
A]

- **Page Table**: Data structure used by the operating system to map virtual addresses to physical addresses, enabling address translation in virtual memory systems.
- **Page Table Entry Size**: The size, in bits or bytes, of each entry in the page table, representing the mapping information for a single page in virtual memory.
- **Page Size**: The fixed-size contiguous block of memory used in virtual memory systems. Greater the Page size less are the number of Pages in the Page table.

Size of Page Table = #PTE x Size of Page Entry

#PTE = Virtual address Size / Page Size

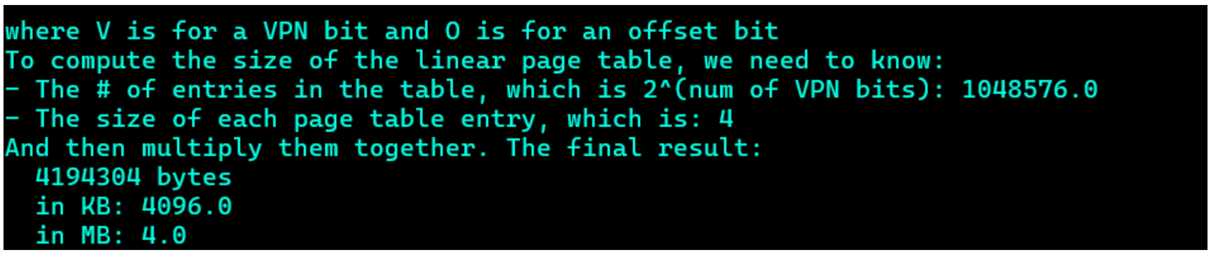
a)



```
PS D:\SEM VI\OS\Labs\Lab7> python.exe .\paging-linear-size.py
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 4

Compute how big a linear page table is with the characteristics you see above.

REMEMBER: There is one linear page table *per process*.
Thus, the more running processes, the more of these tables we will need.
However, for this problem, we are only concerned about the size of *one* table.
```

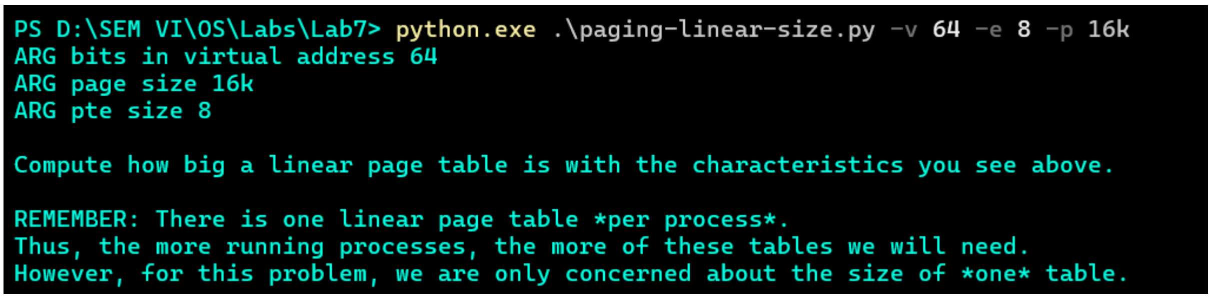
Page Size = 4Kb = 2^2*2^20 = 2^22. bytes
Virtual Address size = 2^32. byyes
# PTE = 2^32/2^22 = 2^10 = 1024

Page table Size = 1024*4 = 4096 bytes



```
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
```

b)



```
PS D:\SEM VI\OS\Labs\Lab7> python.exe .\paging-linear-size.py -v 64 -e 8 -p 16k
ARG bits in virtual address 64
ARG page size 16k
ARG pte size 8

Compute how big a linear page table is with the characteristics you see above.

REMEMBER: There is one linear page table *per process*.
Thus, the more running processes, the more of these tables we will need.
However, for this problem, we are only concerned about the size of *one* table.
```

Page Table Size = #PTE x PTE Size

= (VA size/page Size) x PTE Size

= (2^64 / 2^14) x 2^3

= 2^53 bytes

Page Table Size = 9007199254740992 bytes

Part 4: You will use the program, "paging-linear-translate.py" to see if you understand how simple virtual-to-physical address translation works with linear page tables. See the README_paging for more details

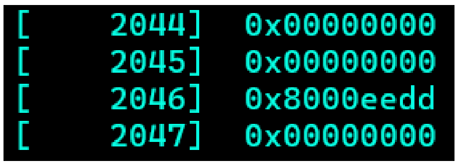
Q1] Before doing any translations, let's use the simulator to study how linear page tables change size given different parameters. Compute the size of linear page tables as different parameters change. Some suggested inputs are below; by using the -v flag, you can see how many page-table entries are filled. First, to understand how linear page table size changes as the address space grows:

A] The size of Page table depends on the size of the address space size, physical memory size and page size.

- Page Table Size ∝ Address space size
- Page Table Size ∝ 1/(Page Size)

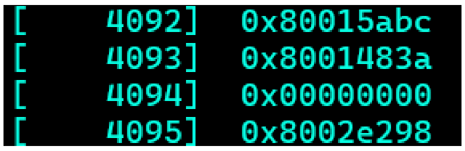
The trend can be observed in the outputs below.

paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0



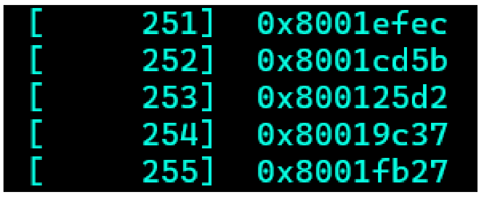
```
[     2044]   0x00000000
[     2045]   0x00000000
[     2046]   0x8000eedd
[     2047]   0x00000000
```

paging-linear-translate.py -P 2k -a 1m -p 512m -v -n 0



```
[     4092]   0x80015abc
[     4093]   0x8001483a
[     4094]   0x00000000
[     4095]   0x8002e298
```

paging-linear-translate.py -P 4k -a 1m -p 512m -v -n 0



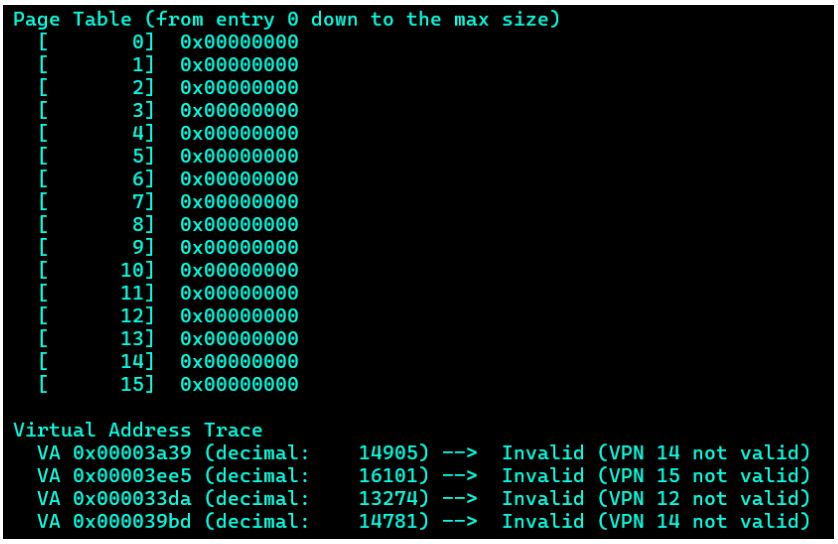
```
[      251]   0x8001efec
[      252]   0x8001cd5b
[      253]   0x800125d2
[      254]   0x80019c37
[      255]   0x8001fb27
```

By using really big pages reduces the page table size but leads to internal fragmentation as the free space within each page is wasted.

Q2] Now let's do some translations. Start with some small examples, and change the number of pages that are allocated to the address space with the -u flag. What happens as you increase the percentage of pages that are allocated in each address space?

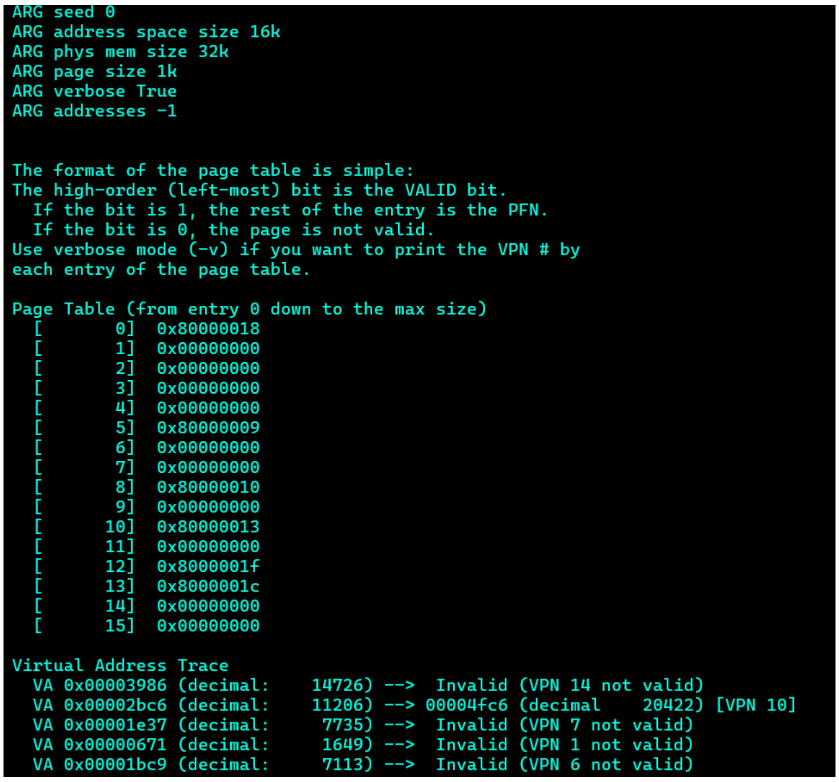
A] We can increase the percentage of pages that we have allocated in each address space by setting the -u flag (determines % VA space used). The used space increases and the number of pages with the valid bit 0 (first bit) decreases. That is more number of pages come become valid in the memory.

- *paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 0*



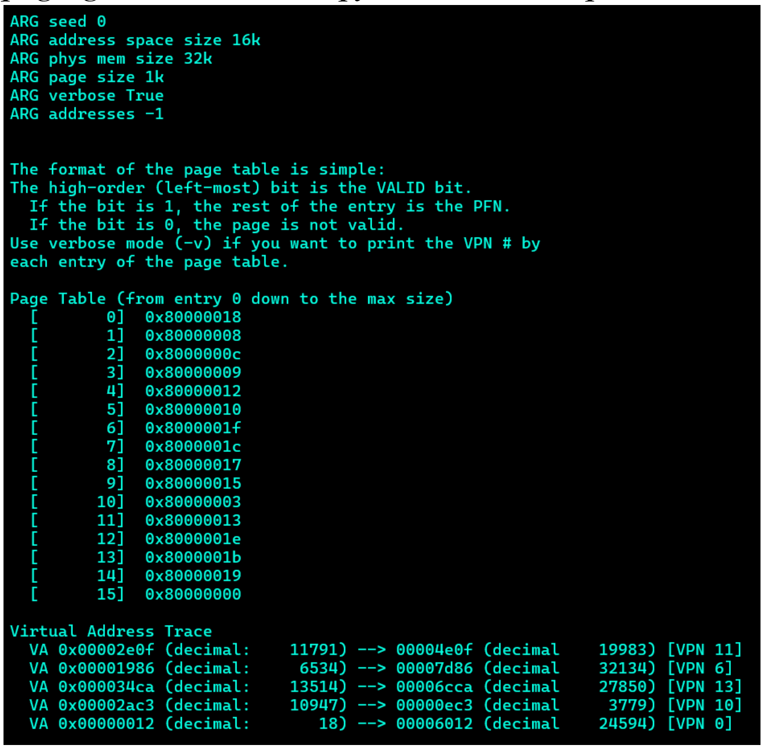
```
Page Table (from entry 0 down to the max size)
  [       0]  0x00000000
  [       1]  0x00000000
  [       2]  0x00000000
  [       3]  0x00000000
  [       4]  0x00000000
  [       5]  0x00000000
  [       6]  0x00000000
  [       7]  0x00000000
  [       8]  0x00000000
  [       9]  0x00000000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

Virtual Address Trace
  VA 0x00003a39 (decimal:    14905) -->  Invalid (VPN 14 not valid)
  VA 0x00003ee5 (decimal:    16101) -->  Invalid (VPN 15 not valid)
  VA 0x000033da (decimal:    13274) -->  Invalid (VPN 12 not valid)
  VA 0x000039bd (decimal:    14781) -->  Invalid (VPN 14 not valid)
```

- *paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 25*



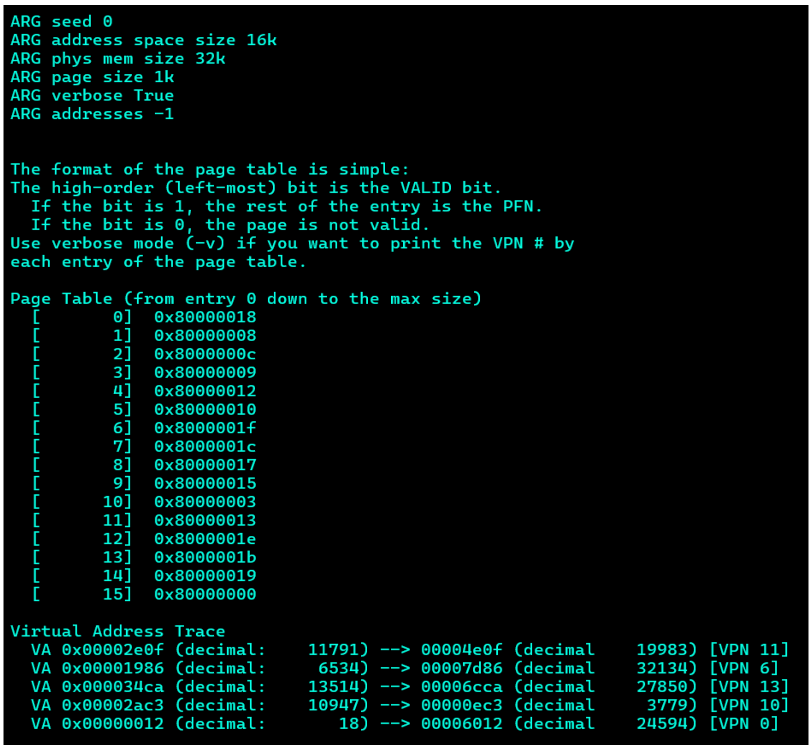
```
ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 1k
ARG verbose True
ARG addresses -1


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

Virtual Address Trace
  VA 0x00003986 (decimal:    14726) -->  Invalid (VPN 14 not valid)
  VA 0x00002bc6 (decimal:    11206) --> 00004fc6 (decimal    20422) [VPN 10]
  VA 0x00001e37 (decimal:     7735) -->  Invalid (VPN 7 not valid)
  VA 0x00000671 (decimal:     1649) -->  Invalid (VPN 1 not valid)
  VA 0x00001bc9 (decimal:     7113) -->  Invalid (VPN 6 not valid)
```

- *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

Virtual Address Trace
  VA 0x00002e0f (decimal:    11791) --> 00004e0f (decimal    19983) [VPN 11]
  VA 0x00001986 (decimal:     6534) --> 00007d86 (decimal    32134) [VPN 6]
  VA 0x000034ca (decimal:    13514) --> 00006cca (decimal    27850) [VPN 13]
  VA 0x00002ac3 (decimal:    10947) --> 00000ec3 (decimal     3779) [VPN 10]
  VA 0x00000012 (decimal:       18) --> 00006012 (decimal    24594) [VPN 0]
```

- *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

Virtual Address Trace
  VA 0x00002e0f (decimal:    11791) --> 00004e0f (decimal    19983) [VPN 11]
  VA 0x00001986 (decimal:     6534) --> 00007d86 (decimal    32134) [VPN 6]
  VA 0x000034ca (decimal:    13514) --> 00006cca (decimal    27850) [VPN 13]
  VA 0x00002ac3 (decimal:    10947) --> 00000ec3 (decimal     3779) [VPN 10]
  VA 0x00000012 (decimal:       18) --> 00006012 (decimal    24594) [VPN 0]
```

From the above screenshots we can verify the answer discussed above that a Decrease in unallocated pages with an increase in -u value. Free memory reduces

Q3] Now let's try some different random seeds, and some different (and sometimes quite crazy) address-space parameters, for variety:

A]

- *paging-linear-translate.py -P 8 -a 32 -p 1024 -v -s 1*
  This seems unrealistic as there can be only 4 pages possible, which is too small to virtualize the address space of any process.

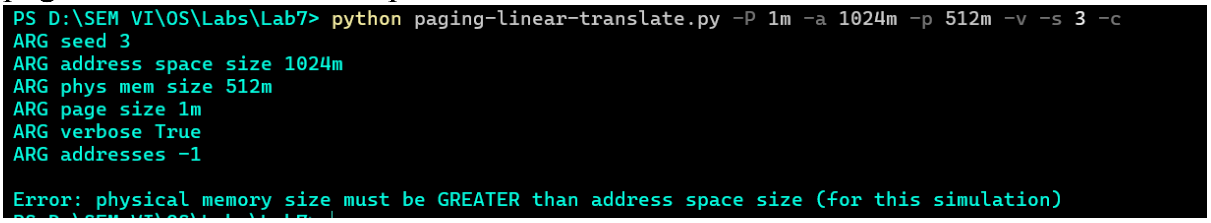
- *paging-linear-translate.py -P 8k -a 32k -p 1m -v -s 2*
  This too seems unrealistic as only 4 pages can be present, and most processes will require more than 4 pages of memory for virtualization.

- *paging-linear-translate.py -P 1m -a 256m -p 512m -v -s 3*
  This also seems to be unrealistic as the address space is divided into 256 pages, due to significant amount of internal fragmentation. Internal fragmentation affects memory paging and swapping, impacting virtual memory management by the operating system.

Q4] Use the program to try out some other problems. Can you find the limits of where the program doesn't work anymore? For example, what happens if the address-space size is bigger than physical memory?

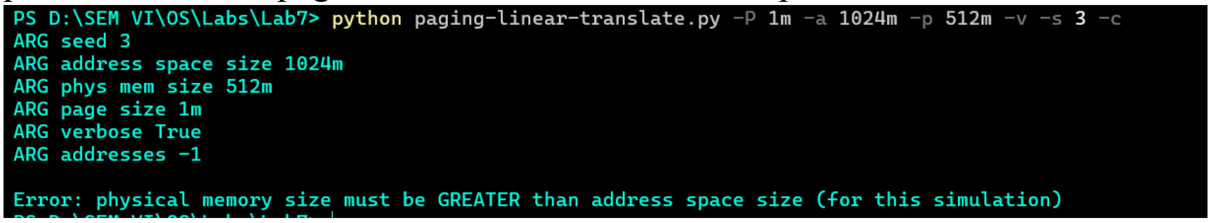
A] The program does not work for some cases listed below:

i. Size of address space is greater that size of physical memory: It is not possible for storing all the pages in the RAM, there will be lot of page faults (Need to fetch the pages from disk). Thrashing occurs when the system spends more time swapping pages between disk and memory than executing actual instructions. This results in severe performance degradation as the CPU spends a significant portion of its time handling page faults and disk I/O operations.



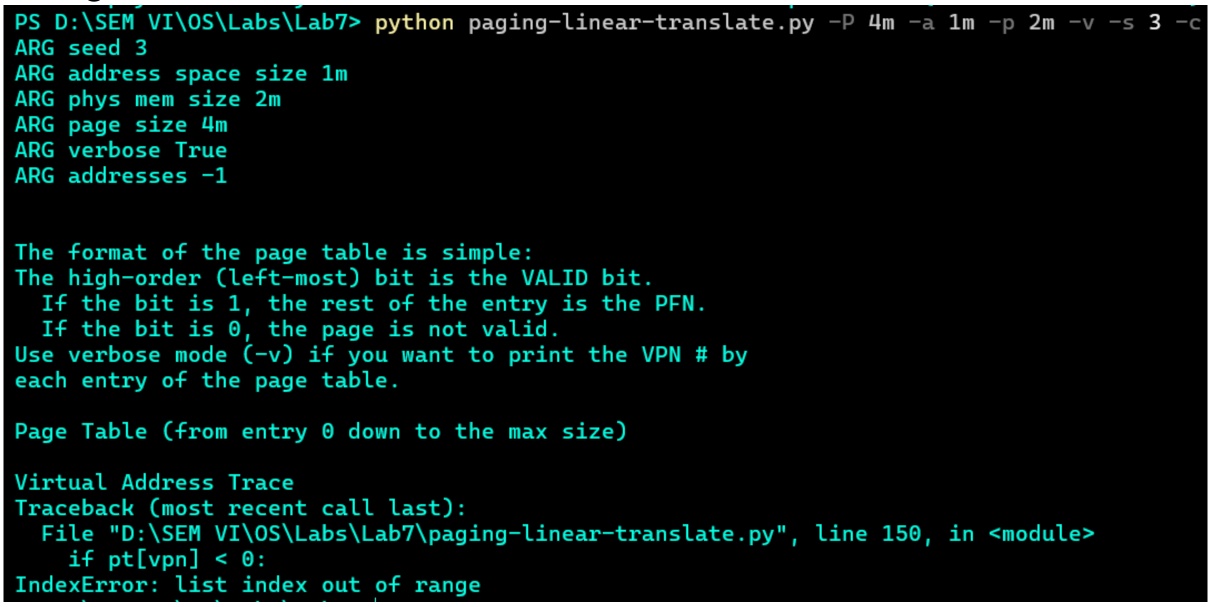
```
PS D:\SEM VI\OS\Labs\Lab7> python paging-linear-translate.py -P 1m -a 1024m -p 512m -v -s 3 -c
ARG seed 3
ARG address space size 1024m
ARG phys mem size 512m
ARG page size 1m
ARG verbose True
ARG addresses -1

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

ii. Size of address space is not multiple of 2: If the physical memory size is not a multiple of the page size, the last page in physical memory may not be fully utilized. This results in wasted memory space, as the remaining portion of the last page cannot be allocated to other processes or data.



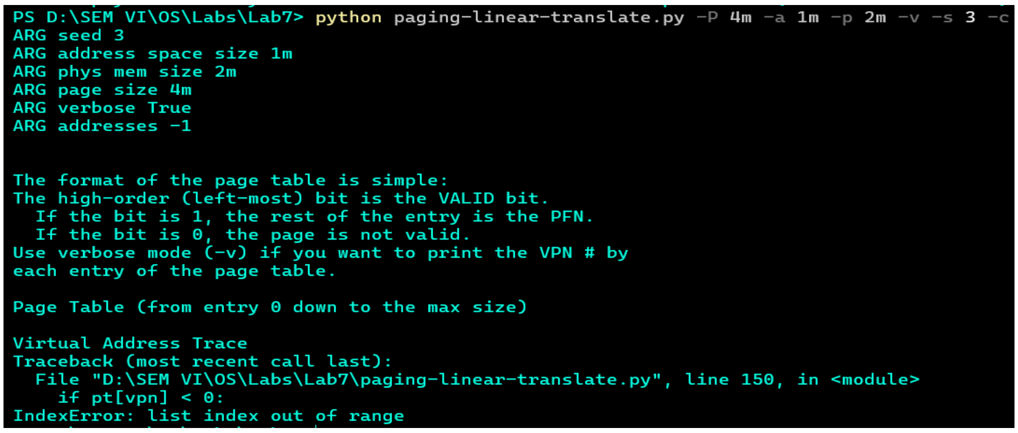
```
PS D:\SEM VI\OS\Labs\Lab7> python paging-linear-translate.py -P 1m -a 1024m -p 512m -v -s 3 -c
ARG seed 3
ARG address space size 1024m
ARG phys mem size 512m
ARG page size 1m
ARG verbose True
ARG addresses -1

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

iii. Size of Page greater than Physical Memory: A single page cannot be stored in the memory at a time. No use of Paging. Index out of range error generated.



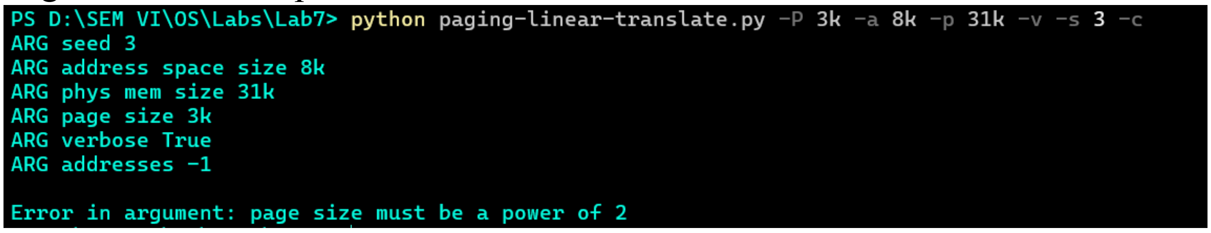
```
PS D:\SEM VI\OS\Labs\Lab7> python paging-linear-translate.py -P 4m -a 1m -p 2m -v -s 3 -c
ARG seed 3
ARG address space size 1m
ARG phys mem size 2m
ARG page size 4m
ARG verbose True
ARG addresses -1


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)

Virtual Address Trace
Traceback (most recent call last):
  File "D:\SEM VI\OS\Labs\Lab7\paging-linear-translate.py", line 150, in <module>
    if pt[vpn] < 0:
IndexError: list index out of range
```

iv. Negative address space, page size, physical memory size : It indicates a misconfiguration or error in the system parameters. Operating systems and hardware expect these parameters to be positive integers representing valid memory units, and negative values cannot be interpreted or processed correctly.



```
PS D:\SEM VI\OS\Labs\Lab7> python paging-linear-translate.py -P 4m -a 1m -p 2m -v -s 3 -c
ARG seed 3
ARG address space size 1m
ARG phys mem size 2m
ARG page size 4m
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)

Virtual Address Trace
Traceback (most recent call last):
  File "D:\SEM VI\OS\Labs\Lab7\paging-linear-translate.py", line 150, in <module>
    if pt[vpn] < 0:
IndexError: list index out of range
```

v. Page Size not multiple of 2:



```
PS D:\SEM VI\OS\Labs\Lab7> python paging-linear-translate.py -P 3k -a 8k -p 31k -v -s 3 -c
ARG seed 3
ARG address space size 8k
ARG phys mem size 31k
ARG page size 3k
ARG verbose True
ARG addresses -1

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