CS - 314 OPERATING SYSTEMS LAB-7

(K.S.N Manikanta, 210010050)

Note: I changed the Python files into Python 3 since there is no python2 installed on my PC. So, I used python instead of py2 in the command while running the files.

QUESTION – 1

1. [Bound = Base + Limit & PA = Base + VA]. If calculated PA > Bound or VA > Limit, then it is a Segmentation violation, if PA < Bound or VA < Limit, then PA = Base + VA is valid.

For seed 1: Virtual Address (VA) Trace => Base: 13884, Limit: 290

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\relocation.py -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) --> PA or segmentation violation?
  VA  1: 0x00000105 (decimal:  261) --> PA or segmentation violation?
  VA  2: 0x000001fb (decimal:  507) --> PA or segmentation violation?
  VA  3: 0x000001cc (decimal:  460) --> PA or segmentation violation?
  VA  4: 0x0000029b (decimal:  667) --> PA or segmentation violation?

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple virtual address space of a given size.
```

VA No.	VA	In or out of Bounds	PA or Segmentation Violation
0	782	Out	Segmentation Violation (782 >290)
1	261	In	PA = 13884 + 261 = 14145 (0x00003741)
2	507	Out	Segmentation Violation (507 >290)
3	460	Out	Segmentation Violation (460 >290)
4	667	Out	Segmentation Violation (667 >290)

This can be verified as below:

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> 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
```

For seed 2: Virtual Address Trace => Base: 15529, Limit: 500

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\relocation.py -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) --> PA or segmentation violation?
  VA  1: 0x00000056 (decimal:   86) --> PA or segmentation violation?
  VA  2: 0x00000357 (decimal:  855) --> PA or segmentation violation?
  VA  3: 0x000002f1 (decimal:  753) --> PA or segmentation violation?
  VA  4: 0x000002ad (decimal:  685) --> PA or segmentation violation?

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple virtual address space of a given size.
```

VA No.	VA	In or out of Bounds	PA or Segmentation Violation
0	57	In	PA = 15529 + 57 = 15586 (0x00003ce2)
1	86	In	PA = 15529 + 86 = 15615 (0x00003cff)
2	855	Out	Segmentation Violation (855>500)
3	753	Out	Segmentation Violation (753 >500)
4	685	Out	Segmentation Violation (685 >500)

This can be verified as below:

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> 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
```

For seed 3: Virtual Address Trace => Base: 8916, Limit: 316

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\relocation.py -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) --> PA or segmentation violation?
  VA  1: 0x0000026a (decimal:  618) --> PA or segmentation violation?
  VA  2: 0x00000280 (decimal:  640) --> PA or segmentation violation?
  VA  3: 0x00000043 (decimal:   67) --> PA or segmentation violation?
  VA  4: 0x0000000d (decimal:   13) --> PA or segmentation violation?

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple virtual address space of a given size.
```

VA No.	VA	In or Out of bounds	PA or Segmentation Violation
0	378	Out	Segmentation Violation (378>316)
1	618	Out	Segmentation Violation (618>316)
2	640	Out	Segmentation Violation (640>316)
3	67	In	PA = 8916 + 67 = 8983 (0x00002317)
4	13	In	PA = 8916 + 13 = 8929 (0x000022e1)

This can be verified as below:

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> 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)
```

2. We can see from the screenshot below, that the Base is 12418, the Limit is 472 and the VA accesses are 430, 265, 523, 414, 802, 310, 488, 597, 929, 516. The highest of them being 929, We can set the **bounds register value to 930** so that all generated VAs are within the bounds.

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\relocation.py -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) --> PA or segmentation violation?
  VA  1: 0x00000109 (decimal:  265) --> PA or segmentation violation?
  VA  2: 0x0000020b (decimal:  523) --> PA or segmentation violation?
  VA  3: 0x0000019e (decimal:  414) --> PA or segmentation violation?
  VA  4: 0x00000322 (decimal:  802) --> PA or segmentation violation?
  VA  5: 0x00000136 (decimal:  310) --> PA or segmentation violation?
  VA  6: 0x000001e8 (decimal:  488) --> PA or segmentation violation?
  VA  7: 0x00000255 (decimal:  597) --> PA or segmentation violation?
  VA  8: 0x000003a1 (decimal:  929) --> PA or segmentation violation?
  VA  9: 0x00000204 (decimal:  516) --> PA or segmentation violation?

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple virtual address space of a given size.
```

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> 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
```

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\relocation.py -s 0 -n 10 -l 930 -c

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

Base-and-Bounds register information:

  Base   : 0x0000360b (decimal 13835)
  Limit  : 930

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

3. When we run with the given flags (seed = 1, no of VA accesses = 10, limit = 100), Size of physical memory = 16kB = 16384 bytes. Since the limit is 100, the **maximum value** that the **base** can be set to will be 16384 – 100 = **16284**.

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\relocation.py -s 1 -n 10 -l 100

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) --> PA or segmentation violation?
  VA  1: 0x0000030e (decimal:  782) --> PA or segmentation violation?
  VA  2: 0x00000105 (decimal:  261) --> PA or segmentation violation?
  VA  3: 0x000001fb (decimal:  507) --> PA or segmentation violation?
  VA  4: 0x000001cc (decimal:  460) --> PA or segmentation violation?
  VA  5: 0x0000029b (decimal:  667) --> PA or segmentation violation?
  VA  6: 0x00000327 (decimal:  807) --> PA or segmentation violation?
  VA  7: 0x00000060 (decimal:   96) --> PA or segmentation violation?
  VA  8: 0x0000001d (decimal:   29) --> PA or segmentation violation?
  VA  9: 0x00000357 (decimal:  855) --> PA or segmentation violation?

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple virtual address space of a given size.
```

4. **Case 1**: Let's consider an address space of 1 MB and a Physical memory size of 16 MB.
Let's run for part 2 and part 3 of this question 1 again. First, let's check for part 2 of this question 1: run with flags -s 0 -n 10 -a 1m -p 16m. Base is 12716364, Limit is 483504. Of all 10 VA accesses, 952225 is the highest/largest. So, if we set the **Limit to 952226,** all VAs will be within the bounds.

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\relocation.py -s 0 -n 10 -a 1m -p 16m

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

Base-and-Bounds register information:

  Base   : 0x00c2094c (decimal 12716364)
  Limit  : 483504

Virtual Address Trace
  VA  0: 0x0006baa9 (decimal: 441001) --> PA or segmentation violation?
  VA  1: 0x00042485 (decimal: 271493) --> PA or segmentation violation?
  VA  2: 0x00082e2e (decimal: 536110) --> PA or segmentation violation?
  VA  3: 0x00067a9c (decimal: 424604) --> PA or segmentation violation?
  VA  4: 0x000c8a70 (decimal: 821872) --> PA or segmentation violation?
  VA  5: 0x0004da5e (decimal: 318046) --> PA or segmentation violation?
  VA  6: 0x0007a024 (decimal: 499748) --> PA or segmentation violation?
  VA  7: 0x00095588 (decimal: 611720) --> PA or segmentation violation?
  VA  8: 0x000e87a1 (decimal: 952225) --> PA or segmentation violation?
  VA  9: 0x00081332 (decimal: 529202) --> PA or segmentation violation?

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple virtual address space of a given size.
```

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\relocation.py -s 0 -n 10 -a 1m -p 16
m -l 952226 -c

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

Base-and-Bounds register information:

  Base   : 0x00d82c07 (decimal 14167047)
  Limit  : 952226

Virtual Address Trace
  VA  0: 0x000c2094 (decimal: 794772) --> VALID: 0x00e44c9b (decimal: 14961819)
  VA  1: 0x0006baa9 (decimal: 441001) --> VALID: 0x00dee6b0 (decimal: 14608048)
  VA  2: 0x00042485 (decimal: 271493) --> VALID: 0x00dc508c (decimal: 14438540)
  VA  3: 0x00082e2e (decimal: 536110) --> VALID: 0x00e05a35 (decimal: 14703157)
  VA  4: 0x00067a9c (decimal: 424604) --> VALID: 0x00dea6a3 (decimal: 14591651)
  VA  5: 0x000c8a70 (decimal: 821872) --> VALID: 0x00e4b677 (decimal: 14988919)
  VA  6: 0x0004da5e (decimal: 318046) --> VALID: 0x00dd0665 (decimal: 14485093)
  VA  7: 0x0007a024 (decimal: 499748) --> VALID: 0x00dfcc2b (decimal: 14666795)
  VA  8: 0x00095588 (decimal: 611720) --> VALID: 0x00e1818f (decimal: 14778767)
  VA  9: 0x000e87a1 (decimal: 952225) --> VALID: 0x00e6b3a8 (decimal: 15119272)
```

Next, let's check for part 3 of question 1: run with flags -s 1 -n 10 -l 100 -a 1m -p 16m. Physical memory size is 16 MB which is 16777216 bytes. Since the Limit is 100, the **maximum value** that the **base** can be set to will be 16777216 – 100 = **16777116.**

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\relocation.py -s 1 -n 10 -l 100 -a 1m -p 16m

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

Base-and-Bounds register information:

  Base   : 0x002265b1 (decimal 2254257)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x000d8f16 (decimal: 888598) --> PA or segmentation violation?
  VA  1: 0x000c386b (decimal: 800875) --> PA or segmentation violation?
  VA  2: 0x000414c3 (decimal: 267459) --> PA or segmentation violation?
  VA  3: 0x0007ed4d (decimal: 519501) --> PA or segmentation violation?
  VA  4: 0x0007311d (decimal: 471325) --> PA or segmentation violation?
  VA  5: 0x000a6cec (decimal: 683244) --> PA or segmentation violation?
  VA  6: 0x000c9e9c (decimal: 827036) --> PA or segmentation violation?
  VA  7: 0x00018072 (decimal: 98418) --> PA or segmentation violation?
  VA  8: 0x0000741c (decimal: 29724) --> PA or segmentation violation?
  VA  9: 0x000d5f4b (decimal: 876363) --> PA or segmentation violation?

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple virtual address space of a given size.
```

Case 2: Let's consider an address space of 4 KB and a Physical memory size of 128 KB.

Let's run for part 2 and part 3 of this question 1 again. First, let's check for part 2 of this question 1: run with flags -s 0 -n 10 -a 4k -p 128k. Base is 18412 and Limit is 1888. Of all the accesses, 3719 is the highest. So, if we set the limit to 3720, all Vas will be within the bound.

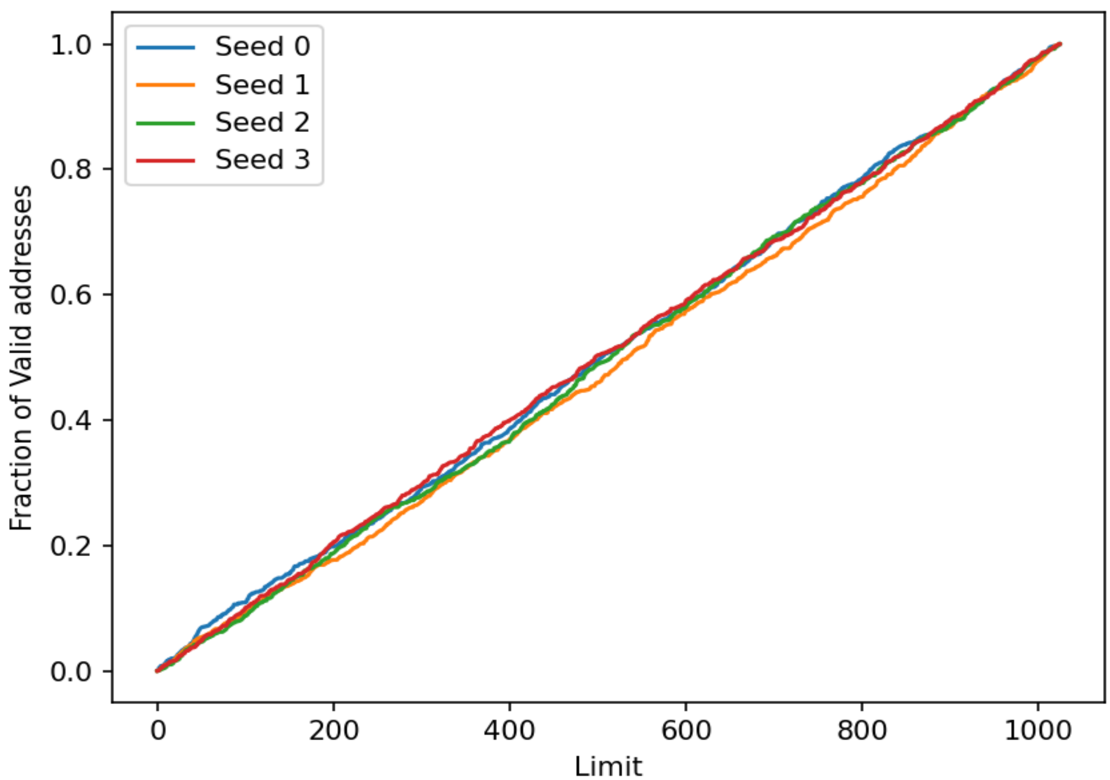
```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\relocation.py -s 0 -n 10 -a 4k -p 12
8k -c

ARG seed 0
ARG address space size 4k
ARG phys mem size 128k

Base-and-Bounds register information:

  Base   : 0x00018412 (decimal 99346)
  Limit  : 1888

Virtual Address Trace
  VA  0: 0x000006ba (decimal: 1722) --> VALID: 0x00018acc (decimal: 101068)
  VA  1: 0x00000424 (decimal: 1060) --> VALID: 0x00018836 (decimal: 100406)
  VA  2: 0x0000082e (decimal: 2094) --> SEGMENTATION VIOLATION
  VA  3: 0x0000067a (decimal: 1658) --> VALID: 0x00018a8c (decimal: 101004)
  VA  4: 0x00000c8a (decimal: 3210) --> SEGMENTATION VIOLATION
  VA  5: 0x000004da (decimal: 1242) --> VALID: 0x000188ec (decimal: 100588)
  VA  6: 0x000007a0 (decimal: 1952) --> SEGMENTATION VIOLATION
  VA  7: 0x00000955 (decimal: 2389) --> SEGMENTATION VIOLATION
  VA  8: 0x00000e87 (decimal: 3719) --> SEGMENTATION VIOLATION
  VA  9: 0x00000813 (decimal: 2067) --> SEGMENTATION VIOLATION
```

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\relocation.py -s 0 -n 10 -a 4k -p 12
8k -l 3720 -c

ARG seed 0
ARG address space size 4k
ARG phys mem size 128k

Base-and-Bounds register information:

  Base   : 0x0001b058 (decimal 110680)
  Limit  : 3720

Virtual Address Trace
  VA  0: 0x00000c20 (decimal: 3104) --> VALID: 0x0001bc78 (decimal: 113784)
  VA  1: 0x000006ba (decimal: 1722) --> VALID: 0x0001b712 (decimal: 112402)
  VA  2: 0x00000424 (decimal: 1060) --> VALID: 0x0001b47c (decimal: 111740)
  VA  3: 0x0000082e (decimal: 2094) --> VALID: 0x0001b886 (decimal: 112774)
  VA  4: 0x0000067a (decimal: 1658) --> VALID: 0x0001b6d2 (decimal: 112338)
  VA  5: 0x00000c8a (decimal: 3210) --> VALID: 0x0001bce2 (decimal: 113890)
  VA  6: 0x000004da (decimal: 1242) --> VALID: 0x0001b532 (decimal: 111922)
  VA  7: 0x000007a0 (decimal: 1952) --> VALID: 0x0001b7f8 (decimal: 112632)
  VA  8: 0x00000955 (decimal: 2389) --> VALID: 0x0001b9ad (decimal: 113069)
  VA  9: 0x00000e87 (decimal: 3719) --> VALID: 0x0001bedf (decimal: 114399)
```

Next, let's check for part 3 of question 1: run with flags -s 1 -n 10 -l 100 -a 4k -p 128k. Physical memory size is 128 KB which is 131072 bytes. Since the Limit is 100, the **maximum value** that the **base** can be set to will be 131072 – 100 = **130972.**

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\relocation.py -s 1 -n 10 -l 100 -a 4
k -p 128k

ARG seed 1
ARG address space size 4k
ARG phys mem size 128k

Base-and-Bounds register information:

  Base   : 0x000044cb (decimal 17611)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x00000d8f (decimal: 3471) --> PA or segmentation violation?
  VA  1: 0x00000c38 (decimal: 3128) --> PA or segmentation violation?
  VA  2: 0x00000414 (decimal: 1044) --> PA or segmentation violation?
  VA  3: 0x000007ed (decimal: 2029) --> PA or segmentation violation?
  VA  4: 0x00000731 (decimal: 1841) --> PA or segmentation violation?
  VA  5: 0x00000a6c (decimal: 2668) --> PA or segmentation violation?
  VA  6: 0x00000c9e (decimal: 3230) --> PA or segmentation violation?
  VA  7: 0x00000180 (decimal:  384) --> PA or segmentation violation?
  VA  8: 0x00000074 (decimal:  116) --> PA or segmentation violation?
  VA  9: 0x00000d5f (decimal: 3423) --> PA or segmentation violation?
```

5. The Graph can be found below: We can see an almost linear graph for all seeds as it is obvious that more addresses become valid with increasing value of the limit.



Question – 2

1. Ideally, all addresses up to half the value of address space size will be in segment 0 and the rest in segment 1. For segment 0, If VA > segment_0_limit, there will be a segmentation violation but if VA < segment_0_limit, then the PA mapping would be valid and PA = segment_0_base + VA. For segment 1, if VA < address_space_size – segment_1_limit, then there will be a segmentation violation but if VA >= address_space_size – segment_1_limit, then the corresponding PA mapping would be valid and PA = segment_1_base – (address_space_size – VA).

   For seed 0 => From the screenshot below, we can see that
   Segment 0 base: 0 and Segment 0 limit: 20 => VA 0-19 would map to 0-19 in PA
   Segment 1 base: 512 and Segment 1 limit: 20 => VA 108-127 would map to 492-511 in PA

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -
L 20 -s 0
ARG seed 0
ARG address space size 128
ARG phys mem size 512

Segment register information:

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

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

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

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple address space with two segments: the top
bit of the virtual address can thus be used to check whether the virtual address
is in segment 0 (topbit=0) or segment 1 (topbit=1). Note that the base/limit pairs
given to you grow in different directions, depending on the segment, i.e., segment 0
grows in the positive direction, whereas segment 1 in the negative.
```

VA No.	VA	PA or Segmentation Violation
0	108	PA = 512 – (128-108) = 492 (0x000001ec)
1	97	Segmentation Violation (97 < 128-20)
2	53	Segmentation Violation (53 > 20)
3	33	Segmentation Violation (33 > 20)
4	65	Segmentation Violation (65 < 128-20)

This can be verified as below:

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\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)
```

For seed 1 => From the screenshot below, we can see that:
Segment 0 base: 0 and Segment 0 limit: 20 => VA 0-19 would map to 0-19 in PA
Segment 1 base: 512 and Segment 1 limit: 20 => VA 108-127 would map to 492-511 in PA

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -
L 20 -s 1
ARG seed 1
ARG address space size 128
ARG phys mem size 512

Segment register information:

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

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

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

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple address space with two segments: the top
bit of the virtual address can thus be used to check whether the virtual address
is in segment 0 (topbit=0) or segment 1 (topbit=1). Note that the base/limit pairs
given to you grow in different directions, depending on the segment, i.e., segment 0
grows in the positive direction, whereas segment 1 in the negative.
```

VA No.	VA	PA or Segmentation Violation
0	17	PA = 0 + 17 = 17 (0x00000011)
1	108	PA = 512 – (128-108) = 492 (0x000001ec)
2	97	Segmentation Violation (97 < 128 – 20)
3	32	Segmentation Violation (32 > 20)
4	63	Segmentation Violation (63 >20)

This can be verified as below:

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\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)
```

For seed 2: From the screenshot below, we can see that:
Segment 0 base: 0 and Segment 0 limit: 20 => VA 0-19 would map to 0-19 in PA
Segment 1 base: 512 and Segment 1 limit: 20 => VA 108-127 would map to 492-511 in PA

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -
L 20 -s 2
ARG seed 2
ARG address space size 128
ARG phys mem size 512

Segment register information:

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

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

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

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple address space with two segments: the top
bit of the virtual address can thus be used to check whether the virtual address
is in segment 0 (topbit=0) or segment 1 (topbit=1). Note that the base/limit pairs
given to you grow in different directions, depending on the segment, i.e., segment 0
grows in the positive direction, whereas segment 1 in the negative.
```

VA No.	VA	PA or Segmentation Violation
0	122	PA = 512 – (128-122) = 506 (0x000001fa)
1	121	PA = 512 – (128-121) = 505 (0x000001f9)
2	7	PA = 0 + 7 = 7 (0x00000007)
3	10	PA = 0 + 10 = 10 (0x0000000a)
4	106	Segmentation Violation (106 < 128 – 20)

This can be verified as below:

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\segmentation.py -a 128 -p 512 -b 0 -
l 20 -B 512 -L 20 -s 2 -c
ARG seed 2
ARG address space size 128
ARG phys mem size 512

Segment register information:

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

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

Virtual Address Trace
  VA  0: 0x0000007a (decimal:  122) --> VALID in SEG1: 0x000001fa (decimal:  506)
  VA  1: 0x00000079 (decimal:  121) --> VALID in SEG1: 0x000001f9 (decimal:  505)
  VA  2: 0x00000007 (decimal:    7) --> VALID in SEG0: 0x00000007 (decimal:    7)
  VA  3: 0x0000000a (decimal:   10) --> VALID in SEG0: 0x0000000a (decimal:   10)
  VA  4: 0x0000006a (decimal:  106) --> SEGMENTATION VIOLATION (SEG1)
```

2. The highest legal virtual address in segment 0 would be 19.
   The lowest legal virtual address in segment 1 would be 108.
   The highest illegal virtual address would be 107.
   The lowest illegal virtual address would be 20.
   To check if we are right, run the command (using the -A flag):

```
python segmentation.py -a 128 -p 512 -b 0 -l
20 -B 512 -L 20 -s 0 -A 19,20,107,108 -c
```

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -
L 20 -s 0 -A 19,20,107,108 -c
ARG seed 0
ARG address space size 128
ARG phys mem size 512

Segment register information:

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

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

Virtual Address Trace
  VA  0: 0x00000013 (decimal:   19) --> VALID in SEG0: 0x00000013 (decimal:   19)
  VA  1: 0x00000014 (decimal:   20) --> SEGMENTATION VIOLATION (SEG0)
  VA  2: 0x0000006b (decimal:  107) --> SEGMENTATION VIOLATION (SEG1)
  VA  3: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492)
```

3. According to the question, we have a 16-byte address space and 128-byte physical memory and the valid virtual addresses are 0,1,14 and 15. For this to happen, we need to have:
   Segment 0 base: 0, Limit: 2
   Segment 1 base: 128, Limit: 2
   Therefore, **b0 would be 0, l0 would be 2, b1 would be 128 and l1 would be 2**. We can check this by running the command:

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

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python  .\segmentation.py -a 16 -p 128 -A 0,1,2,3,4,5,6,7,
8,9,10,11,12,13,14,15 --b0 0 --l0 2 --b1 128 --l1 2 -c
ARG seed 0
ARG address space size 16
ARG phys mem size 128

Segment register information:

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

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

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

4. To configure the simulator such that 90% of the virtual addresses are valid, we can set both limits in such a way that their sum is more than 90% of the address space size. Parameters l0(segment 0 limit) and l1(segment1 limit) along with the address space size are important to get this outcome. Below's an example:

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python  .\segmentation.py -a 1000 -p 16k --b0
 0 --l0 477 --b1 1000 --l1 477 -n 10 -c
ARG seed 0
ARG address space size 1000
ARG phys mem size 16k

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 477

  Segment 1 base  (grows negative) : 0x000003e8 (decimal 1000)
  Segment 1 limit                  : 477

Virtual Address Trace
  VA  0: 0x0000034c (decimal:  844) --> VALID in SEG1: 0x0000034c (decimal:  844)
  VA  1: 0x000002f5 (decimal:  757) --> VALID in SEG1: 0x000002f5 (decimal:  757)
  VA  2: 0x000001a4 (decimal:  420) --> VALID in SEG0: 0x000001a4 (decimal:  420)
  VA  3: 0x00000102 (decimal:  258) --> VALID in SEG0: 0x00000102 (decimal:  258)
  VA  4: 0x000001ff (decimal:  511) --> SEGMENTATION VIOLATION (SEG1)
  VA  5: 0x00000194 (decimal:  404) --> VALID in SEG0: 0x00000194 (decimal:  404)
  VA  6: 0x0000030f (decimal:  783) --> VALID in SEG1: 0x0000030f (decimal:  783)
  VA  7: 0x0000012f (decimal:  303) --> VALID in SEG0: 0x0000012f (decimal:  303)
  VA  8: 0x000001dc (decimal:  476) --> VALID in SEG0: 0x000001dc (decimal:  476)
  VA  9: 0x00000247 (decimal:  583) --> VALID in SEG1: 0x00000247 (decimal:  583)
```

We have an address space of 1000 bytes and both l1 and l2 are set to 477 bytes, hence giving 90% or more valid addresses.

5. Yes, we can run the simulator such that no virtual addresses are valid by having the (both l0 and l1) segmentation limits equal to zero. We can test this by running the command:

```
python segmentation.py -a 128 -p 1k --b0 0 --
l0 0 --b1 256 --l1 0 -s 1 -n 10 -c
```

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python .\segmentation.py  -a 128 -p 1k --b0 0
 --l0 0 --b1 256 --l1 0 -s 1 -n 10 -c
ARG seed 1
ARG address space size 128
ARG phys mem size 1k

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: 0x00000011 (decimal:   17) --> SEGMENTATION VIOLATION (SEG0)
  VA  1: 0x0000006c (decimal:  108) --> SEGMENTATION VIOLATION (SEG1)
  VA  2: 0x00000061 (decimal:   97) --> SEGMENTATION VIOLATION (SEG1)
  VA  3: 0x00000020 (decimal:   32) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x0000003f (decimal:   63) --> SEGMENTATION VIOLATION (SEG0)
  VA  5: 0x00000039 (decimal:   57) --> SEGMENTATION VIOLATION (SEG0)
  VA  6: 0x00000053 (decimal:   83) --> SEGMENTATION VIOLATION (SEG1)
  VA  7: 0x00000064 (decimal:  100) --> SEGMENTATION VIOLATION (SEG1)
  VA  8: 0x0000000c (decimal:   12) --> SEGMENTATION VIOLATION (SEG0)
  VA  9: 0x00000003 (decimal:    3) --> SEGMENTATION VIOLATION (SEG0)
```

Question-3

Case 1: **python paging-linear-size.py -v 16 -e 2 -p 1k**

In this case, we have
no. of bits in virtual address = 16
size of page table entry = 2 bytes
page size = 1k = 2^10 bytes => which means we require 10 bits for offset.
size of memory = 2^16
no. of pages = no. of page table entries = 2^16 / 2^10 = 2^6 = 64
So, **size of page table = 64 * 2 = 128 bytes**

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python paging-linear-size.py -v 16 -e 2 -p 1k
 -c
ARG bits in virtual address 16
ARG page size 1k
ARG pte size 2

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

The number of bits in the virtual address: 16
The page size: 1024 bytes
Thus, the number of bits needed in the offset: 10
Which leaves this many bits for the VPN: 6
Thus, a virtual address looks like this:

VVVVVV|OOOOOOOOOO

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): 64.0
- The size of each page table entry, which is: 2
And then multiply them together. The final result:
  128 bytes
  in KB: 0.125
  in MB: 0.0001220703125
```

<u>Case 2</u>: **python paging-linear-size.py -v 16 -e 4 -p 4k**

In this case, we have

no. of bits in virtual address = 16

size of page table entry = 4 bytes

page size = 4k = 2^12 bytes => which means we require 12 bits for offset.

size of memory = 2^16

no. of pages = no. of page table entries = 2^16 / 2^12 = 2^4 = 16

So, **size of page table = 16 * 4 = 64 bytes**

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python paging-linear-size.py -v 16 -e 4 -p 4k
 -c
ARG bits in virtual address 16
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: 16
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 4
Thus, a virtual address looks like this:

VVVV|OOOOOOOOOOOO

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): 16.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  64 bytes
  in KB: 0.0625
  in MB: 6.103515625e-05
```

Case 3: **python paging-linear-size.py -v 32 -e 4 -p 4k** [default case]

In this case, we have
no of bits in virtual address = 32
size of page table entry = 4 bytes
page size = 4k = 4096 bytes = 2^12 => which means we require 12 bits for offset.
size of physical memory = 2^32
no. of pages = no. of page table entries = 2^32 / 2^12 = 2^20
So, **size of page table = 2^20 * 4 = 2^22 = 4194304 bytes = 4MB**

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python paging-linear-size.py -v 32 -e 4 -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:

VVVVVVVVVVVVVVVVVVVV|OOOOOOOOOOOO

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
```

Case 4: **python paging-linear-size.py -v 32 -e 8 -p 16k**

In this case, we have
no of bits in virtual address = 32
size of page table entry = 8 bytes
page size = 16k = 2^14 bytes => which means we require 14 bits for offset.
size of memory = 2^32
no. of pages = no. of page table entries = 2^32 / 2^14 = 2^18
So, **size of page table** = 2^18 * 8 = 2^21 = **2097152 bytes = 2 MB**

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python paging-linear-size.py -v 32 -e 8 -p 16
k -c
ARG bits in virtual address 32
ARG page size 16k
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: 16384 bytes
Thus, the number of bits needed in the offset: 14
Which leaves this many bits for the VPN: 18
Thus, a virtual address looks like this:

VVVVVVVVVVVVVVVVVV|OOOOOOOOOOOOOO

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 262144.0
- The size of each page table entry, which is: 8
And then multiply them together. The final result:
  2097152 bytes
  in KB: 2048.0
  in MB: 2.0
```

Case 5: **python paging-linear-size.py -v 64 -e 8 -p 16k**

In this case, we have

no of bits in virtual address = 64

size of page table entry = 8 bytes

page size = 16k = 2^14 bytes => which means we require 14 bits for offset.

size of memory = 2^64

no. of pages = no. of page table entries = 2^64 / 2^14 = 2^50

So, **size of page table** = 2^50 * 8 = 2^53 = **9007199254740992 bytes = 8589934592 MB**

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python paging-linear-size.py -v 64 -e 8 -p 16
k -c
ARG bits in virtual address 64
ARG page size 16k
ARG pte size 8

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

The number of bits in the virtual address: 64
The page size: 16384 bytes
Thus, the number of bits needed in the offset: 14
Which leaves this many bits for the VPN: 50
Thus, a virtual address looks like this:

VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV|OOOOOOOOOOOOOO

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): 1125899906842624.0
- The size of each page table entry, which is: 8
And then multiply them together. The final result:
  9007199254740992 bytes
  in KB: 8796093022208.0
  in MB: 8589934592.0
```

From all the above cases we can conclude that the page table size is directly proportional to the size of the page table entry and indirectly proportional to the page size. It is exponentially proportional to the no. of bits in the virtual address, i.e., directly proportional to 2 ^ (no. of bits).

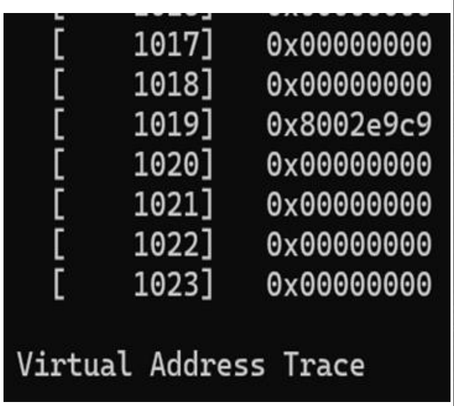
Question-4

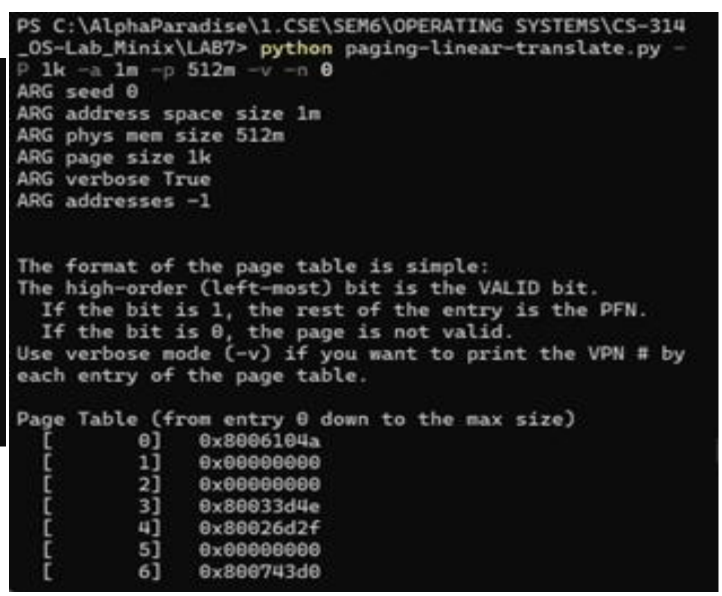
1. **Before translation**:
   Using default seed 0

a. Increasing address space size:

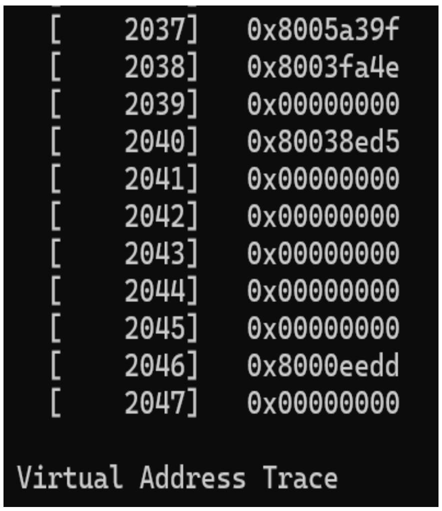
```
python paging-linear-translate.py -P 1k -a 1m
-p 512m -v -n 0
```

> page size = 1k = 2^10 bytes
> address space size = 1m = 2^20 bytes
> no. of pages = **no. of page table entries** = 2^20 / 2^10 = 2^10 = **1024**, i.e., from 0 to 1023
> size of page table entry = 4
> **size of page table** = 2^10*4 = 2^12 = 4096 bytes = **4KB**

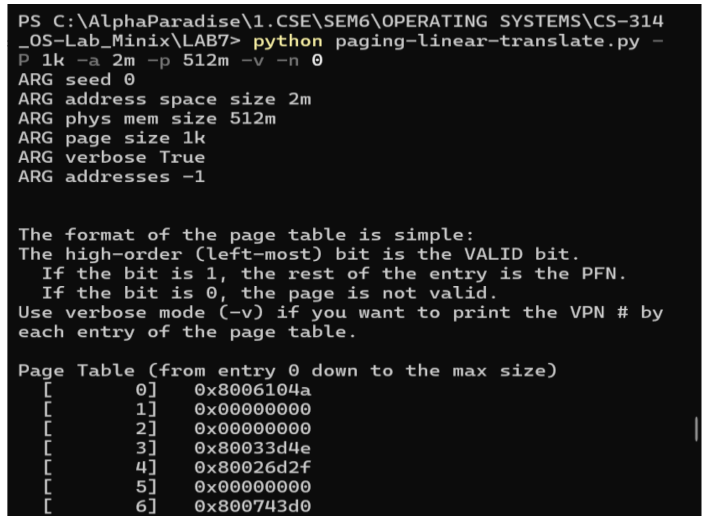




```
python paging-linear-translate.py -P 1k -a 2m
-p 512m -v -n 0
```



```
  [    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
```

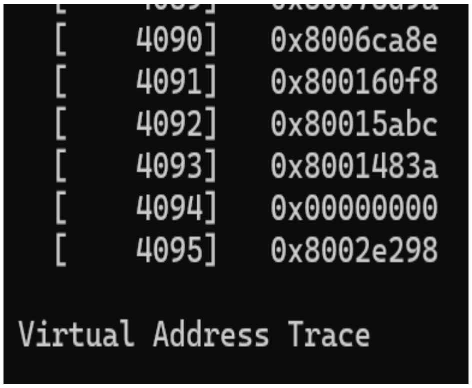


```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314
_OS-Lab_Minix\LAB7> python paging-linear-translate.py -
P 1k -a 2m -p 512m -v -n 0
ARG seed 0
ARG address space size 2m
ARG phys mem size 512m
ARG page size 1k
ARG verbose True
ARG addresses -1


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
```

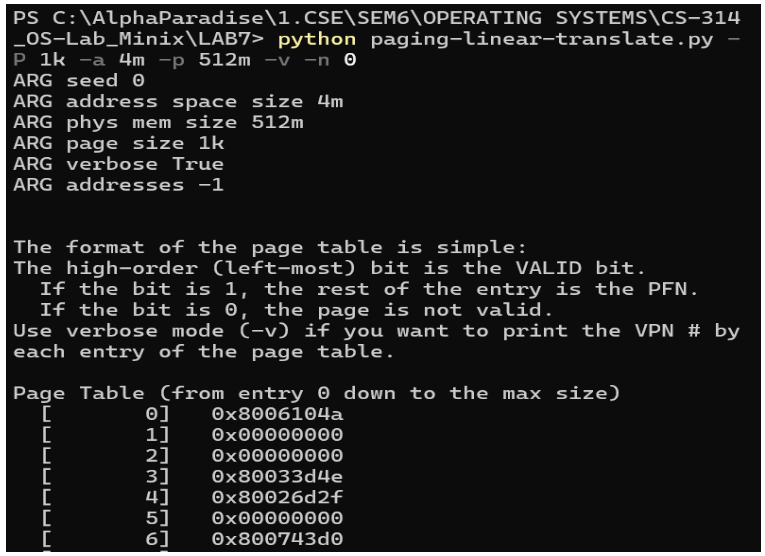
> page size = 1k = 2^10 bytes
> address space size = 2m = 2* 2^20 bytes = 2^21 bytes
> no. of pages = **no. of page table entries** = 2^21 / 2^10 = 2^11 = **2048**, i.e., from 0 to 2047
> size of page table entry = 4
> **size of page table** = 2^11*4 = 2^13 = 8192 bytes = **8KB**

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

> page size = 1k = 2^10 bytes
> address space size = 4m = 4* 2^20 bytes = 2^22 bytes
> no. of pages = **no. of page table entries** = 2^22 / 2^10 = 2^12 = **4096**, i.e., from 0 to 4095
> size of page table entry = 4
> **size of page table** = 2^12*4 = 2^14 = 16384 bytes = **16KB**



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

Virtual Address Trace
```



```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314
_OS-Lab_Minix\LAB7> python paging-linear-translate.py -
P 1k -a 4m -p 512m -v -n 0
ARG seed 0
ARG address space size 4m
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
```

Thus, we can conclude that the size of the page table is directly proportional to the address space size because, when the address space size doubles, the size of the page table also doubles. each entry in the page table corresponds to a portion of the address space. Therefore, as the address space size increases, more entries are needed in the page table to map each virtual address to its corresponding physical address.

b. <u>Increasing page size</u>

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

> page size = 1k = 2^10 bytes
> address space size = 1m = 2^20 bytes
> no. of pages = **no. of page table entries** = 2^20 / 2^10 = 2^10 = **1024**, i.e., from 0 to 1023
> size of page table entry = 4
> **size of page table** = 2^10*4 = 2^12 = 4096 bytes = **4KB**

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

Virtual Address Trace
```

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314
_OS-Lab_Minix\LAB7> python paging-linear-translate.py -
P 1k -a 1m -p 512m -v -n 0
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
```

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

> page size = 2k = 2^11 bytes

> address space size = 1m = 2^20 bytes

> no. of pages = **no. of page table entries** = 2^20 / 2^11 = 2^9 = **512**, i.e., from 0 to 511

> size of page table entry = 4

> **size of page table** = 2^9*4 = 2^11 = 2048 bytes = **2KB**

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314
_OS-Lab_Minix\LAB7> python paging-linear-translate.py -
P 2k -a 1m -p 512m -v -n 0
ARG seed 0
ARG address space size 1m
ARG phys mem size 512m
ARG page size 2k
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]   0x80030825
  [       1]   0x00000000
  [       2]   0x00000000
  [       3]   0x80019ea7
  [       4]   0x80013697
  [       5]   0x00000000
  [       6]   0x8003a1e8
  [       7]   0x8001209a
  [       8]   0x80027935
  [       9]   0x00000000
  [      10]   0x8003ee5f
```

```
[     502]   0x8000309b
[     503]   0x8003ea63
[     504]   0x00000000
[     505]   0x00000000
[     506]   0x00000000
[     507]   0x00000000
[     508]   0x8001a7f2
[     509]   0x8001c337
[     510]   0x00000000
[     511]   0x00000000

Virtual Address Trace
```

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

> page size = 4k = 2^12 bytes
> address space size = 1m = 2^20 bytes
> no. of pages = **no. of page table entries** = 2^20 / 2^12 = 2^8 = **256**, i.e., from 0 to 255
> size of page table entry = 4
> **size of page table** = 2^8*4 = 2^10 = 1024 bytes = **1KB**

```
[      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
```

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314
_OS-Lab_Minix\LAB7> python paging-linear-translate.py -
P 4k -a 1m -p 512m -v -n 0
ARG seed 0
ARG address space size 1m
ARG phys mem size 512m
ARG page size 4k
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]   0x80018412
  [        1]   0x00000000
  [        2]   0x00000000
  [        3]   0x8000cf53
  [        4]   0x80009b4b
  [        5]   0x00000000
  [        6]   0x8001d0f4
  [        7]   0x8000904d
  [        8]   0x80013c9a
```

Thus, we can conclude that the size of the page table is inversely proportional to the page size because, when the page size doubles, the size of the page table becomes half. When the page size increases, fewer pages are needed to cover the entire address space. This means that each entry in the page table covers a larger range of addresses. Consequently, the page table can have fewer entries since each entry represents a larger chunk of the address space.

We should not use big pages in general because it can lead to internal fragmentation, i.e., the entire big page is always allocated but only a fraction of it is used, while a lot of unused memory space remains. With larger page sizes, the TLB can hold fewer entries due to the larger amount of memory each translation occupies. This increases the likelihood of TLB misses, leading to slower memory access times. If the working set size of a process is smaller than the page size, using large pages may lead to unnecessary memory overhead.

2. Since the address space size is 16k or 2^14 bytes, the required no. of bits for the virtual address is 14 and the page size is 1k or 2^10, so the no. of bits required for offset is 10. Hence, no of bits for VPN will be 14 – 10 = 4 bits. No. of pages will be 2^14 / 2^10 = 2^4 or 16.

   As we increase the percentage of virtual address space used (percentage of pages allocated), more pages are allocated, i.e., no. of page table entries increase. Hence, no. of page hits would increase over page faults, i.e., more no. of memory accesses (VA) become valid, as seen in the screenshots below. As more of the address space is utilized by processes, there is a higher demand for memory resources to store the data and instructions needed by those processes. This leads to more frequent access to virtual memory addresses to fetch or store data.

```
python 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)
  VA 0x000013d9 (decimal:     5081) -->  Invalid (VPN 4 not valid)
```

Here, we can see that all addresses are deemed invalid since no pages are allocated.

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

We can see 6 entries are filled in 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)
```

```
python paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 50
```

We can see 9 entries are filled in the page table.

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

Virtual Address Trace
  VA 0x00003385 (decimal:    13189) --> 00003f85 (decimal    16261) [VPN 12]
  VA 0x0000231d (decimal:     8989) -->  Invalid (VPN 8 not valid)
  VA 0x000000e6 (decimal:      230) --> 000060e6 (decimal    24806) [VPN 0]
  VA 0x00002e0f (decimal:    11791) -->  Invalid (VPN 11 not valid)
  VA 0x00001986 (decimal:     6534) --> 00007586 (decimal    30086) [VPN 6]
```

```
python paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 75
```

We can see all the entries are filled in 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]
```

```
python paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 100
```

We can see all the entries are filled in 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]
```

3.

```
1) python paging-linear-translate.py -P 8 -a 32 -p 1024 -v -s 1
```

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python paging-linear-translate.py -P 8 -a 32 -p 1024 -v -s
 1 -c
ARG seed 1
ARG address space size 32
ARG phys mem size 1024
ARG page size 8
ARG verbose True
ARG addresses -1


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

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

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

This Parameter combination seems unrealistic because the address space and physical memory size are too small to be practically used. Also, the page size of 8 bytes is too small and we can accommodate only a maximum of 4 pages, which won’t be efficient, making it unrealistic.

```
2) python paging-linear-translate.py -P 8k -a 32k -p 1m -v -s 2
```

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python paging-linear-translate.py -P 8k -a 32k -p 1m -v -s
 2 -c
ARG seed 2
ARG address space size 32k
ARG phys mem size 1m
ARG page size 8k
ARG verbose True
ARG addresses -1


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

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

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

This case also seems a little unrealistic. Though the page size is realistic (8 KB), with an address space of 32 KB, we can accommodate only 4 pages. This case (case 2) is better compared to case 1 in terms of the address space, physical memory and pages being large enough in size. But, even then we accommodate only a maximum of 4 pages, thus making it a little unrealistic.

```
3) python paging-linear-translate.py -P 1m -a 256m -p 512m -v -s 3 -c
```

This case is also unrealistic because the page size (1 MB) is too much as most of the page would be unused or empty and can also lead to internal fragmentation. Also, having large pages eats up memory and thus reducing the memory available for other processes running.

```
  [     244]   0x80000049
  [     245]   0x800000f5
  [     246]   0x800000ef
  [     247]   0x800001a4
  [     248]   0x800000f6
  [     249]   0x00000000
  [     250]   0x800001eb
  [     251]   0x00000000
  [     252]   0x00000000
  [     253]   0x00000000
  [     254]   0x80000159
  [     255]   0x00000000

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

Thus, overall case 3 seems the most unrealistic combination followed by case 1. Though case 2 seems reasonable, still it is a little unrealistic. (**Unrealistic-ness order: 3>>1>>2**).

4. The program code has some requirements like the physical memory size being greater than the address space size for the program to work correctly as expected. Some of the requirements/limitations of the program are:

1. All the values such as address space size, physical memory size, limit etc., should be positive.

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python paging-linear-translate.py -P 8 -a -32
 -p -1024 -v -s 1
ARG seed 1
ARG address space size -32
ARG phys mem size -1024
ARG page size 8
ARG verbose True
ARG addresses -1

Error: must specify a non-zero physical memory size.
```

2. Physical memory size should be greater than or equal to the address space size. Otherwise, it would not be possible to accommodate all available addresses.

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python paging-linear-translate.py -P 8 -a 32
-p 24 -v -s 1
ARG seed 1
ARG address space size 32
ARG phys mem size 24
ARG page size 8
ARG verbose True
ARG addresses -1

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

3. Both Physical memory size and address space size should be less than 1 GB.

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python paging-linear-translate.py -P 8 -a 32
-p 2G -v -s 1
ARG seed 1
ARG address space size 32
ARG phys mem size 2G
ARG page size 8
ARG verbose True
ARG addresses -1

Error: must use smaller sizes (less than 1 GB) for this simulation.
```

4. The address space size and physical memory size must be in powers of 2 so that there won't be any discontinuities in memory addressing.

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python paging-linear-translate.py -P 9 -a 99
-p 999 -v -s 1
ARG seed 1
ARG address space size 99
ARG phys mem size 999
ARG page size 9
ARG verbose True
ARG addresses -1

Error in argument: address space must be a power of 2
```

5. The page size must also be in powers of 2 as both the physical memory size and address space size must be multiples of page size so that there would be an integer no. of pages.

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python paging-linear-translate.py -P 8 -a 16
 -p 999 -v -s 1
ARG seed 1
ARG address space size 16
ARG phys mem size 999
ARG page size 8
ARG verbose True
ARG addresses -1

Error in argument: physical memory must be a multiple of the pagesize
```

If the address space size is more than the physical memory size, the program returns an error:

```
PS C:\AlphaParadise\1.CSE\SEM6\OPERATING SYSTEMS\CS-314_OS-Lab_Minix\LAB7> python paging-linear-translate.py -P 8k -a 32k -p 16k -v -
s 2 -c
ARG seed 2
ARG address space size 32k
ARG phys mem size 16k
ARG page size 8k
ARG verbose True
ARG addresses -1

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

This is because, if the address space size is more than the physical memory size, we cannot fit/accommodate all the available addresses in the address space in the physical memory. The mapping from VA to PA would be erroneous.