Lab 7

Suyash Gaurav (210010054)

Operating Systems

Q1 (a)

Executed the relocation.py program with seed values 1, 2, and 3, along with the -c flag to compute translations. Computed the translation for in-bound virtual addresses using the provided base and bounds register information.

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 relocation.py -s 1 -c

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

Base-and-Bounds register information:

  Base   : 0x0000363c (decimal 13884)
  Limit  : 290

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

Only Virtual Address 1 has been translated to physical address as it falls in bounds.

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 relocation.py -s 2 -c

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

Base-and-Bounds register information:

  Base   : 0x00003ca9 (decimal 15529)
  Limit  : 500

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

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 relocation.py -s 3 -c

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

Base-and-Bounds register information:

  Base   : 0x000022d4 (decimal 8916)
  Limit  : 316

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

Q1 (b)

To ensure all generated virtual addresses are within bounds (in a 1k address space), set the bounds register (-l flag) to cover the entire range. Since the address space is 1k, set the bounds register to at least 1024, covering addresses 0 to 1023. **Thus limit = 1024.**

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 relocation.py -s 0 -n 10 -l 1024 -c

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

Base-and-Bounds register information:

  Base   : 0x0000360b (decimal 13835)
  Limit  : 1024

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

Q1 (c)

To ensure that the entire address space fits within physical memory, we need to set the base register such that it allows for the full range of addresses (100 in this case) to reside within physical memory.

Maximum Base Register Value = **Physical Memory Size - Bounds Register**

= 16 * 1024 - 100

= 16384 - 100

= 16284

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 relocation.py -s 1 -n 10 -l 100 -b 1
6284 -c

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

Base-and-Bounds register information:

  Base   : 0x00003f9c (decimal 16284)
  Limit  : 100

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

Q1 (d)

I increased the address space size **(-a) to 4k (4 kilobytes)** and the physical memory size **(-p) to 64k (64 kilobytes)** and repeated Q1. With a larger address space (4k) and physical memory (64k), there's more room for virtual addresses to fit within the bounds set by the base and bounds registers.

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 relocation.py -s 1 -n 10 -a 4k -p 64k -l 16K -c

ARG seed 1
ARG address space size 4k
ARG phys mem size 64k

Base-and-Bounds register information:

  Base   : 0x00002265 (decimal 8805)
  Limit  : 16384

Virtual Address Trace
  VA  0: 0x00000d8f (decimal: 3471) --> VALID: 0x00002ff4 (decimal: 12276)
  VA  1: 0x00000c38 (decimal: 3128) --> VALID: 0x00002e9d (decimal: 11933)
  VA  2: 0x00000414 (decimal: 1044) --> VALID: 0x00002679 (decimal: 9849)
  VA  3: 0x000007ed (decimal: 2029) --> VALID: 0x00002a52 (decimal: 10834)
  VA  4: 0x00000731 (decimal: 1841) --> VALID: 0x00002996 (decimal: 10646)
  VA  5: 0x00000a6c (decimal: 2668) --> VALID: 0x00002cd1 (decimal: 11473)
  VA  6: 0x00000c9e (decimal: 3230) --> VALID: 0x00002f03 (decimal: 12035)
  VA  7: 0x00000180 (decimal:  384) --> VALID: 0x000023e5 (decimal: 9189)
  VA  8: 0x00000074 (decimal:  116) --> VALID: 0x000022d9 (decimal: 8921)
  VA  9: 0x00000d5f (decimal: 3423) --> VALID: 0x00002fc4 (decimal: 12228)
```

I increased the address space size **(-a) to 8k (8 kilobytes)** and the physical memory size **(-p) to 16m (16 megabytes)**, **(-l) to 4k** and repeated Q3.

Maximum Base Register Value = **Physical Memory Size - Bounds Register**

= 16 * 1024 * 1024 - 4 * 1024

= 16380 k

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 relocation.py -s 1 -n 10 -a 8k -p 16m -l 4k -b 16380k -c

ARG seed 1
ARG address space size 8k
ARG phys mem size 16m

Base-and-Bounds register information:

  Base   : 0x00fff000 (decimal 16773120)
  Limit  : 4096

Virtual Address Trace
  VA  0: 0x0000044c (decimal: 1100) --> VALID: 0x00fff44c (decimal: 16774220)
  VA  1: 0x00001b1e (decimal: 6942) --> SEGMENTATION VIOLATION
  VA  2: 0x00001870 (decimal: 6256) --> SEGMENTATION VIOLATION
  VA  3: 0x00000829 (decimal: 2089) --> VALID: 0x00fff829 (decimal: 16775209)
  VA  4: 0x00000fda (decimal: 4058) --> VALID: 0x00ffffda (decimal: 16777178)
  VA  5: 0x00000e62 (decimal: 3682) --> VALID: 0x00fffe62 (decimal: 16776802)
  VA  6: 0x000014d9 (decimal: 5337) --> SEGMENTATION VIOLATION
  VA  7: 0x0000193d (decimal: 6461) --> SEGMENTATION VIOLATION
  VA  8: 0x00000300 (decimal:  768) --> VALID: 0x00fff300 (decimal: 16773888)
  VA  9: 0x000000e8 (decimal:  232) --> VALID: 0x00fff0e8 (decimal: 16773352)
```

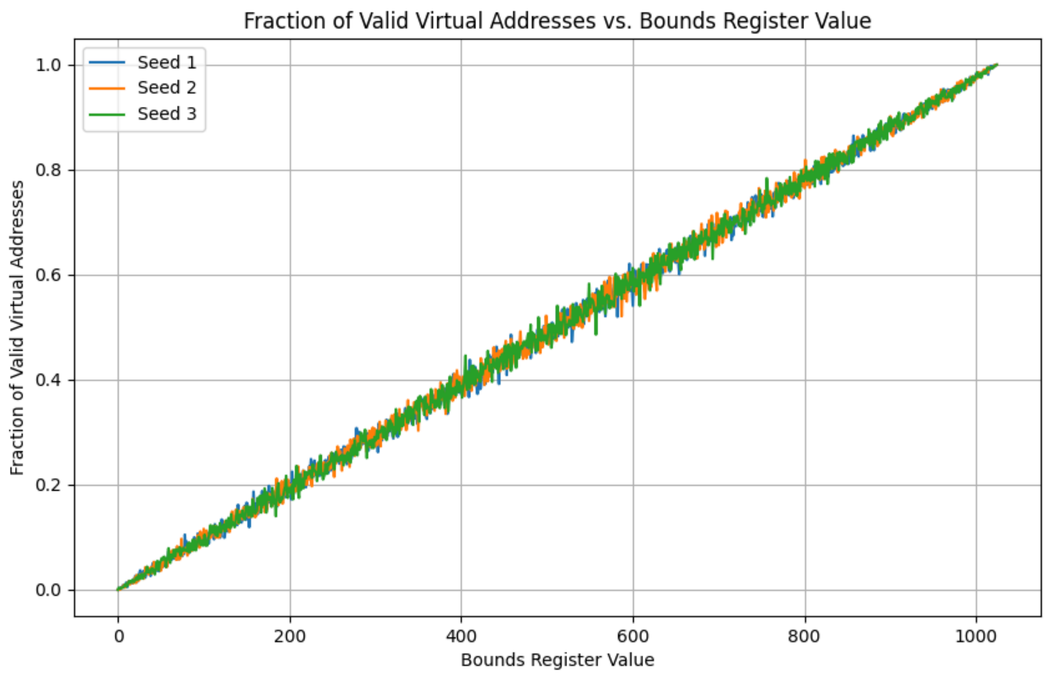
Q1 (e)

Experimental Setup:

```
num_addresses = 1000

address_space_size = 1024  # 1k address space size

physical_memory_size = 16384  # 16k physical memory size

bounds_values = range(0, address_space_size + 1)  # Range of bounds register values

random_seeds = [1, 2, 3]  # Random seeds for different runs
```

- As the bounds register value increases, more virtual addresses fall within the bounds, resulting in a higher fraction of valid virtual addresses.

- However, once the bounds register value exceeds the size of the address space, all virtual addresses become valid, resulting in a fraction of 1.0.



Q2 (a)

- Use the command -c at the end of the command line to perform translation.

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0 -c
ARG seed 0
ARG address space size 128
ARG phys mem size 512

Segment register information:

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

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

Virtual Address Trace
  VA  0: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492)
  VA  1: 0x00000061 (decimal:   97) --> SEGMENTATION VIOLATION (SEG1)
  VA  2: 0x00000035 (decimal:   53) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x00000021 (decimal:   33) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x00000041 (decimal:   65) --> SEGMENTATION VIOLATION (SEG1)
```

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20
-s 1 -c
ARG seed 1
ARG address space size 128
ARG phys mem size 512

Segment register information:

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

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

Virtual Address Trace
  VA  0: 0x00000011 (decimal:   17) --> VALID in SEG0: 0x00000011 (decimal:   17)
  VA  1: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492)
  VA  2: 0x00000061 (decimal:   97) --> SEGMENTATION VIOLATION (SEG1)
  VA  3: 0x00000020 (decimal:   32) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x0000003f (decimal:   63) --> SEGMENTATION VIOLATION (SEG0)
```

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20
-s 2 -c
ARG seed 2
ARG address space size 128
ARG phys mem size 512

Segment register information:

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

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

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

Q2 (b)

Segment 0's base: 0

Segment 0's limit: 20

The highest legal virtual address in segment 0 would be the base address (0) plus the limit (20), minus 1, since addresses are zero-indexed.

Highest legal virtual address in segment 0 = **0**+**20-1** = **19**

On using -A flag, for segment 0, **-A 20 -> Invalid and -A 19 -> Valid. Thus verified.**

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20
-s 0 -A 20 -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: 0x00000014 (decimal:   20) --> SEGMENTATION VIOLATION (SEG0)
```

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20
-s 0 -A 19 -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)
```

Segment 1's base: 512

Segment 1's limit: 20

The lowest legal virtual address in segment 1 would be the base address (127) minus the limit (20), plus 1, since addresses are zero-indexed.

Highest legal physical address in segment 1 = **512 - 20** + **1** = **493**

Highest legal virtual address in segment 1 = **128-20** = **108**

On using -A flag, for segment 1, **-A 107** -> **Invalid and -A 108 -> Valid. Thus verified.**

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20
-s 0 -A 107 -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: 0x0000006b (decimal:  107) --> SEGMENTATION VIOLATION (SEG1)
```

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20
-s 0 -A 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: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492)
```

Q2 (c)

Given a 16-byte address space and a 128-byte physical memory, the following is the setup:

Segment 0: Valid virtual addresses 0 and 1

Base: 0 (Starting at the beginning of physical memory)

Limit: 2 (Allocating 2 bytes for segment 0)

Segment 1: Valid virtual addresses 14 and 15

Base: 128 (Starting at the end of physical memory)

Limit: 2 (Allocating 2 bytes for segment 1)

Thus, **virtual addresses 0 and 1 will map to physical addresses 0 and 1, and virtual addresses 14 and 15 will map to physical addresses 126 and 127**. All other virtual addresses will result in segmentation violations.

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 segmentation.py -a 16 -p 128  -A 0,1,2,3,4,5,6,
7,8,9,10,11,12,13,14,15 --b0 0 --l0 2 --b1 128 --l1 2 -c
ARG seed 0
ARG address space size 16
ARG phys mem size 128

Segment register information:

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

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

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

Q2(d)

The key parameters to consider for achieving this outcome are the address space size (-a), physical memory size (-p), and the base and limit registers for each segment (--b0, --l0, --b1, --l1).

For example:

Assume a 20-byte address space and a 128-byte physical memory, the following is the setup:

Segment 0:

Base: 0 (Starting at the beginning of physical memory)

Limit: 9

Segment 1: Valid virtual addresses 14 and 15

Base: 128 (Starting at the end of physical memory)

Limit: 9

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 segmentation.py -a 20 -p 128 --b0 0 --l0 9 -
-b1 128 --l1 9 -n 20 -c
ARG seed 0
ARG address space size 20
ARG phys mem size 128

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 9

  Segment 1 base  (grows negative) : 0x00000080 (decimal 128)
  Segment 1 limit                  : 9

Virtual Address Trace
  VA  0: 0x00000010 (decimal:   16) --> VALID in SEG1: 0x0000007c (decimal:  124)
  VA  1: 0x0000000f (decimal:   15) --> VALID in SEG1: 0x0000007b (decimal:  123)
  VA  2: 0x00000008 (decimal:    8) --> VALID in SEG0: 0x00000008 (decimal:    8)
  VA  3: 0x00000005 (decimal:    5) --> VALID in SEG0: 0x00000005 (decimal:    5)
  VA  4: 0x0000000a (decimal:   10) --> SEGMENTATION VIOLATION (SEG1)
  VA  5: 0x00000008 (decimal:    8) --> VALID in SEG0: 0x00000008 (decimal:    8)
  VA  6: 0x0000000f (decimal:   15) --> VALID in SEG1: 0x0000007b (decimal:  123)
  VA  7: 0x00000006 (decimal:    6) --> VALID in SEG0: 0x00000006 (decimal:    6)
  VA  8: 0x00000009 (decimal:    9) --> SEGMENTATION VIOLATION (SEG0)
  VA  9: 0x0000000b (decimal:   11) --> VALID in SEG1: 0x00000077 (decimal:  119)
  VA 10: 0x00000012 (decimal:   18) --> VALID in SEG1: 0x0000007e (decimal:  126)
  VA 11: 0x0000000a (decimal:   10) --> SEGMENTATION VIOLATION (SEG1)
  VA 12: 0x00000005 (decimal:    5) --> VALID in SEG0: 0x00000005 (decimal:    5)
  VA 13: 0x0000000f (decimal:   15) --> VALID in SEG1: 0x0000007b (decimal:  123)
  VA 14: 0x0000000c (decimal:   12) --> VALID in SEG1: 0x00000078 (decimal:  120)
  VA 15: 0x00000005 (decimal:    5) --> VALID in SEG0: 0x00000005 (decimal:    5)
  VA 16: 0x00000012 (decimal:   18) --> VALID in SEG1: 0x0000007e (decimal:  126)
  VA 17: 0x00000013 (decimal:   19) --> VALID in SEG1: 0x0000007f (decimal:  127)
  VA 18: 0x00000010 (decimal:   16) --> VALID in SEG1: 0x0000007c (decimal:  124)
  VA 19: 0x00000012 (decimal:   18) --> VALID in SEG1: 0x0000007e (decimal:  126)
```

Thus, **% of valid virtual addresses generated** = **17/20 * 100** = **85%**

IInd Example:

Assume a 40-byte address space and a 128-byte physical memory, the following is the setup:

<u>Segment 0:</u>

Base: 0 (Starting at the beginning of physical memory)

Limit: 19

<u>Segment 1:</u> Valid virtual addresses 14 and 15

Base: 128 (Starting at the end of physical memory)

Limit: 19

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 segmentation.py -a 40 -p 128 --b0 0 --l0 19 --b1 128 --l1
 19 -n 40 -c
ARG seed 0
ARG address space size 40
ARG phys mem size 128

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 19

  Segment 1 base  (grows negative) : 0x00000080 (decimal 128)
  Segment 1 limit                  : 19

Virtual Address Trace
  VA  0: 0x00000021 (decimal:   33) --> VALID in SEG1: 0x00000079 (decimal:  121)
  VA  1: 0x0000001e (decimal:   30) --> VALID in SEG1: 0x00000076 (decimal:  118)
  VA  2: 0x00000010 (decimal:   16) --> VALID in SEG0: 0x00000010 (decimal:   16)
  VA  3: 0x0000000a (decimal:   10) --> VALID in SEG0: 0x0000000a (decimal:   10)
  VA  4: 0x00000014 (decimal:   20) --> SEGMENTATION VIOLATION (SEG1)
  VA  5: 0x00000010 (decimal:   16) --> VALID in SEG0: 0x00000010 (decimal:   16)
  VA  6: 0x0000001f (decimal:   31) --> VALID in SEG1: 0x00000077 (decimal:  119)
  VA  7: 0x0000000c (decimal:   12) --> VALID in SEG0: 0x0000000c (decimal:   12)
  VA  8: 0x00000013 (decimal:   19) --> SEGMENTATION VIOLATION (SEG0)
  VA  9: 0x00000017 (decimal:   23) --> VALID in SEG1: 0x0000006f (decimal:  111)
  VA 10: 0x00000024 (decimal:   36) --> VALID in SEG1: 0x0000007c (decimal:  124)
  VA 11: 0x00000014 (decimal:   20) --> SEGMENTATION VIOLATION (SEG1)
  VA 12: 0x0000000b (decimal:   11) --> VALID in SEG0: 0x0000000b (decimal:   11)
  VA 13: 0x0000001e (decimal:   30) --> VALID in SEG1: 0x00000076 (decimal:  118)
  VA 14: 0x00000018 (decimal:   24) --> VALID in SEG1: 0x00000070 (decimal:  112)
  VA 15: 0x0000000a (decimal:   10) --> VALID in SEG0: 0x0000000a (decimal:   10)
  VA 16: 0x00000024 (decimal:   36) --> VALID in SEG1: 0x0000007c (decimal:  124)
  VA 17: 0x00000027 (decimal:   39) --> VALID in SEG1: 0x0000007f (decimal:  127)
  VA 18: 0x00000020 (decimal:   32) --> VALID in SEG1: 0x00000078 (decimal:  120)
  VA 19: 0x00000024 (decimal:   36) --> VALID in SEG1: 0x0000007c (decimal:  124)
  VA 20: 0x0000000c (decimal:   12) --> VALID in SEG0: 0x0000000c (decimal:   12)
```

```
  VA 21: 0x0000001d (decimal:   29) --> VALID in SEG1: 0x00000075 (decimal:  117)
  VA 22: 0x00000023 (decimal:   35) --> VALID in SEG1: 0x0000007b (decimal:  123)
  VA 23: 0x0000001b (decimal:   27) --> VALID in SEG1: 0x00000073 (decimal:  115)
  VA 24: 0x00000012 (decimal:   18) --> VALID in SEG0: 0x00000012 (decimal:   18)
  VA 25: 0x00000004 (decimal:    4) --> VALID in SEG0: 0x00000004 (decimal:    4)
  VA 26: 0x00000011 (decimal:   17) --> VALID in SEG0: 0x00000011 (decimal:   17)
  VA 27: 0x00000018 (decimal:   24) --> VALID in SEG1: 0x00000070 (decimal:  112)
  VA 28: 0x00000024 (decimal:   36) --> VALID in SEG1: 0x0000007c (decimal:  124)
  VA 29: 0x00000026 (decimal:   38) --> VALID in SEG1: 0x0000007e (decimal:  126)
  VA 30: 0x00000013 (decimal:   19) --> SEGMENTATION VIOLATION (SEG0)
  VA 31: 0x00000022 (decimal:   34) --> VALID in SEG1: 0x0000007a (decimal:  122)
  VA 32: 0x0000000a (decimal:   10) --> VALID in SEG0: 0x0000000a (decimal:   10)
  VA 33: 0x00000020 (decimal:   32) --> VALID in SEG1: 0x00000078 (decimal:  120)
  VA 34: 0x00000015 (decimal:   21) --> VALID in SEG1: 0x0000006d (decimal:  109)
  VA 35: 0x00000000 (decimal:    0) --> VALID in SEG0: 0x00000000 (decimal:    0)
  VA 36: 0x0000001c (decimal:   28) --> VALID in SEG1: 0x00000074 (decimal:  116)
  VA 37: 0x0000000f (decimal:   15) --> VALID in SEG0: 0x0000000f (decimal:   15)
  VA 38: 0x00000020 (decimal:   32) --> VALID in SEG1: 0x00000078 (decimal:  120)
  VA 39: 0x0000001a (decimal:   26) --> VALID in SEG1: 0x00000072 (decimal:  114)
```

Thus, **% of valid virtual addresses generated** = **36/40 * 100 = 90%**

Q2(e)

I set up the segment base and limit registers in a way that all virtual addresses map outside the boundaries of the allocated memory segments.

Set the limit register to 0 for both segment 0 and segment 1.

Segment-0's lowest valid address is 0+0-1 = -1. (out of bounds)

Segment-1's lowest valid address is 128+0+1, or 128. (out of bounds segment)

Thus, no virtual address will be valid.

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 segmentation.py -a 16 -p 128 --b0 0 --l0 0 --b1 127 --l1
0 -n 20 -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                  : 0

  Segment 1 base  (grows negative) : 0x0000007f (decimal 127)
  Segment 1 limit                  : 0

Virtual Address Trace
  VA  0: 0x0000000d (decimal:   13) --> SEGMENTATION VIOLATION (SEG1)
  VA  1: 0x0000000c (decimal:   12) --> SEGMENTATION VIOLATION (SEG1)
  VA  2: 0x00000006 (decimal:    6) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x00000004 (decimal:    4) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x00000008 (decimal:    8) --> SEGMENTATION VIOLATION (SEG1)
  VA  5: 0x00000006 (decimal:    6) --> SEGMENTATION VIOLATION (SEG0)
  VA  6: 0x0000000c (decimal:   12) --> SEGMENTATION VIOLATION (SEG1)
  VA  7: 0x00000004 (decimal:    4) --> SEGMENTATION VIOLATION (SEG0)
  VA  8: 0x00000007 (decimal:    7) --> SEGMENTATION VIOLATION (SEG0)
  VA  9: 0x00000009 (decimal:    9) --> SEGMENTATION VIOLATION (SEG1)
  VA 10: 0x0000000e (decimal:   14) --> SEGMENTATION VIOLATION (SEG1)
  VA 11: 0x00000008 (decimal:    8) --> SEGMENTATION VIOLATION (SEG1)
  VA 12: 0x00000004 (decimal:    4) --> SEGMENTATION VIOLATION (SEG0)
  VA 13: 0x0000000c (decimal:   12) --> SEGMENTATION VIOLATION (SEG1)
  VA 14: 0x00000009 (decimal:    9) --> SEGMENTATION VIOLATION (SEG1)
  VA 15: 0x00000004 (decimal:    4) --> SEGMENTATION VIOLATION (SEG0)
  VA 16: 0x0000000e (decimal:   14) --> SEGMENTATION VIOLATION (SEG1)
  VA 17: 0x0000000f (decimal:   15) --> SEGMENTATION VIOLATION (SEG1)
  VA 18: 0x0000000c (decimal:   12) --> SEGMENTATION VIOLATION (SEG1)
  VA 19: 0x0000000e (decimal:   14) --> SEGMENTATION VIOLATION (SEG1)
```

Q3. 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.

Different parameters:

- Number of bits in the virtual address space (VASIZE).
- Size of each page table entry (PTESIZE).
- Size of the page (PAGESIZE).

Case 1: Different Number of Bits in the Address Space

Increasing the number of bits in the address space allows for more virtual addresses, leading to a larger page table. Page table size doubles with each additional bit of address space.

PTE count = 2^(VPN bits). The constant number of offset bits leads to an increase in VPN bits and PTEs.

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 paging-linear-size.py -v 32 -c
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 4

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

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

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

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

Thus, if VASIZE = 32, linear page table size = 4 MB

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 paging-linear-size.py -v 33 -c
ARG bits in virtual address 33
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: 33
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 21
Thus, a virtual address looks like this:

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

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

Thus, if VASIZE = 33, linear page table size = 8 MB

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 paging-linear-size.py -v 34 -c
ARG bits in virtual address 34
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: 34
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 22
Thus, a virtual address looks like this:

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

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 4194304.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  16777216 bytes
  in KB: 16384.0
  in MB: 16.0
```

Thus, if VASIZE = 34, linear page table size = 16 MB

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 paging-linear-size.py -v 35 -c
ARG bits in virtual address 35
ARG page size 4k
ARG pte size 4

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

The number of bits in the virtual address: 35
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 23
Thus, a virtual address looks like this:

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

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 8388608.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  33554432 bytes
  in KB: 32768.0
  in MB: 32.0
```

Thus, if VASIZE = 35, linear page table size = 32 MB

Case 2: Different Page Size

Larger page size reduces the number of pages, hence reducing the size of the page table.

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 paging-linear-size.py -p 1k -c
ARG bits in virtual address 32
ARG page size 1k
ARG pte size 4

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

The number of bits in the virtual address: 32
The page size: 1024 bytes
Thus, the number of bits needed in the offset: 10
Which leaves this many bits for the VPN: 22
Thus, a virtual address looks like this:

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

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 4194304.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  16777216 bytes
  in KB: 16384.0
  in MB: 16.0
```

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

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 paging-linear-size.py -p 2k -c
ARG bits in virtual address 32
ARG page size 2k
ARG pte size 4

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

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

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

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

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

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 paging-linear-size.py -p 4k -c
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 4

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

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

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

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

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

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 paging-linear-size.py -p 8k -c
ARG bits in virtual address 32
ARG page size 8k
ARG pte size 4

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

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

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

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

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

Case 3: Different Page Table Entry Size

Larger page table entry size means each entry occupies more memory, resulting in a larger page table size.

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 paging-linear-size.py -e 4 -c
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 4

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

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

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

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

If size of each page table entry is 4, linear page table size = 4 MB

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 paging-linear-size.py -e 8 -c
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 8

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

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

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

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

If size of each page table entry is 8, linear page table size = 8 MB

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 paging-linear-size.py -e 16 -c
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 16

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

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

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

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

If size of each page table entry is 16, linear page table size = 16 MB

Q4 (a)

- **Address space increases:**

```
  [     1008]   0x8003e4b7
  [     1009]   0x00000000
  [     1010]   0x8000bc33
  [     1011]   0x00000000
  [     1012]   0x8001d1ab
  [     1013]   0x8007df94
  [     1014]   0x800052d0
  [     1015]   0x00000000
  [     1016]   0x00000000
  [     1017]   0x00000000
  [     1018]   0x00000000
  [     1019]   0x8002e9c9
  [     1020]   0x00000000
  [     1021]   0x00000000
  [     1022]   0x00000000
  [     1023]   0x00000000

Virtual Address Trace
```

, Using -a 1m, there are total 1024 entries.

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

Using -a 2m, there are a total of 2048 entries in the page table

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

 Virtual Address Trace
```

Using -a 4m, there are total of 4096 entries in the page table.

Thus, as the address space gets bigger, the page table gets bigger as well.

- Page Size Increases:

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

Virtual Address Trace
```

If page size=1K, there are total of 1023 entries in the page table.

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

Virtual Address Trace
```

If page size=2K, there are total of 512entries in the page table.

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

Virtual Address Trace
```

If page size=4K, there are total of 256 entries in the page table.

Thus, as the size of the page table as page size increases, the number of page table entries decreases.

Q4 (b)

-u 0 (0% of pages allocated):

With no pages allocated, all translations will result in invalid addresses since none of the pages in the address space are mapped to physical memory.

As u increases, the percentage of allocated pages increases. This is because more pages in the address space are mapped to physical memory.

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 0 -c
```

```
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
```

Thus, 0% page allocated.(0x00000000)

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 25 -c
```

```
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
```

Thus, approx 25% allocated.

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 50 -c
```

```
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
```

Thus 50 % page allocated.

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 75 -c
```

```
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
```

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 100 -c
```

```
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
```

Since u = 100, thus, all translations will result in valid addresses since all pages in the address space are mapped to physical memory.

Q4 (c)

-P 8 -a 32 -p 1024 -v -s 1

This combination specifies a page size of 8 bytes and an address space size of 32 bytes. While technically valid, it's an unrealistic scenario as it's very small and impractical for real-world use cases.

```
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)
```

-P 8k -a 32k -p 1m -v -s 2

This combination specifies a more realistic scenario with a page size of 8 kilobytes and an address space size of 32 kilobytes.

```
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)
```

-P 1m -a 256m -p 512m -v -s 3

```
[      238]   0x00000000
[      239]   0x00000000
[      240]   0x00000000
[      241]   0x00000000
[      242]   0x00000000
[      243]   0x00000000
[      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]
```

This combination specifies a page size of 1 megabyte, an address space size of 256 megabytes, and a physical memory size of 512 megabytes. While technically feasible, it's an extreme scenario with a very large address space compared to physical memory.

Q4 (d)

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 paging-linear-translate.py -P 4k -a 64k -p 32k -c
ARG seed 0
ARG address space size 64k
ARG phys mem size 32k
ARG page size 4k
ARG verbose False
ARG addresses -1

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

In this case, the address space size (64 kilobytes) is larger than the physical memory size (32 kilobytes). Thus, problem occurred.

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 paging-linear-translate.py -P 3k -a 16k -p 32k -c
ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 3k
ARG verbose False
ARG addresses -1

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

In this case, page size is not a multiple of 2. Thus, error occurred.

```
suyas@Z-Sparrow:/mnt/c/Users/suyas/OneDrive/Documents/Minix3/LAB_7$ python2 paging-linear-translate.py -P 64k -a 16k -p 32k -c
ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 64k
ARG verbose False
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 "paging-linear-translate.py", line 174, in <module>
    if pt[vpn] < 0:
IndexError: array index out of range
```

In this case, page size is greater than address space size, thus, array index out of range error occurred.