Operating Systems Lab

Lab – 7

Rahul Cheryala, 210010012

Q1.

1) In Base and Bound, Physical Address corresponding to the given Virtual Address is calculated as Physical Address = Base + Virtual Address.

And here, Virtual Address is valid only if 0 <= Virtual Address < Bound.

```
C:\Users\Rahul\Documents\SEM-6\OPERATING_SYSTEMS\OS Lab\Lab_7>py -2 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.
```

ARGUMENTS seed=1, address space size=1k, physical memory size=16k
Base-and-Bounds register information:
Base : 0x0000363c (decimal 13884)
Limit : 290

Virtual Address Trace
VA 0: 0x0000030e (decimal: 782) --> OUT OF BOUNDS
VA 1: 0x00000105 (decimal: 261) --> IN BOUNDS
Corresponding Physical Address: 0x00003741 (decimal: 14145)
VA 2: 0x000001fb (decimal: 507) --> OUT OF BOUNDS
VA 3: 0x000001cc (decimal: 460) --> OUT OF BOUNDS
VA 4: 0x0000029b (decimal: 667) --> OUT OF BOUNDS

```
C:\Users\Rahul\Documents\SEM-6\OPERATING_SYSTEMS\OS Lab\Lab_7>py -2 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.
```

ARGUMENTS seed=2, address space size=1k, physical memory size=16k,

Base-and-Bounds register information:

Base : 0x00003ca9 (decimal 15529)

Limit : 500

Virtual Address Trace

VA 0: 0x00000039 (decimal: 57) --> IN BOUNDS

Corresponding Physical Address: 0x00003ce2 (decimal: 15586)

VA 1: 0x00000056 (decimal: 86) --> IN BOUNDS

Corresponding Physical Address: 0x00003cff (decimal: 15615)

VA 2: 0x00000357 (decimal: 855) --> OUT OF BOUNDS

VA 3: 0x000002f1 (decimal: 753) --> OUT OF BOUNDS

VA 4: 0x000002ad (decimal: 685) --> OUT OF BOUNDS

```
C:\Users\Rahul\Documents\SEM-6\OPERATING_SYSTEMS\OS Lab\Lab_7>py -2 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.
```

ARGUMENTS seed=3, address space size=1k, physical memory size=16k
Base-and-Bounds register information:
Base : 0x000022d4 (decimal 8916)
Limit : 316

Virtual Address Trace
VA 0: 0x0000017a (decimal: 378) --> OUT OF BOUNDS
VA 1: 0x0000026a (decimal: 618) --> OUT OF BOUNDS
VA 2: 0x00000280 (decimal: 640) --> OUT OF BOUNDS
VA 3: 0x00000043 (decimal: 67) --> IN BOUNDS
Corresponding Physical Address: 0x00002317 (decimal: 8983)
VA 4: 0x0000000d (decimal: 13) --> IN BOUNDS
Corresponding Physical Address: 0x000022e1 (decimal: 8929)

2)

```
C:\Users\Rahul\Documents\SEM-6\OPERATING SYSTEMS\OS Lab\Lab_7>py -2 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.
```

I have set -l value to 930 or greater to ensure that all the generated virtual addresses are within bounds. This is because the maximum virtual address generated is 929. So, to make every generated virtual address valid, we need to have every virtual address less than bound. Therefore, the value of bound or limit must be 930 or greater and less than 16384.

3)

```
C:\Users\Rahul\Documents\SEM-6\OPERATING_SYSTEMS\OS Lab\Lab_7>py -2 relocation.py -s 0 -n 10 -l 100

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

Base-and-Bounds register information:

  Base   : 0x0000360b (decimal 13835)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x00000308 (decimal:  776) --> PA or segmentation violation?
  VA  1: 0x000001ae (decimal:  430) --> PA or segmentation violation?
  VA  2: 0x00000109 (decimal:  265) --> PA or segmentation violation?
  VA  3: 0x0000020b (decimal:  523) --> PA or segmentation violation?
  VA  4: 0x0000019e (decimal:  414) --> PA or segmentation violation?
  VA  5: 0x00000322 (decimal:  802) --> PA or segmentation violation?
  VA  6: 0x00000136 (decimal:  310) --> PA or segmentation violation?
  VA  7: 0x000001e8 (decimal:  488) --> PA or segmentation violation?
  VA  8: 0x00000255 (decimal:  597) --> PA or segmentation violation?
  VA  9: 0x000003a1 (decimal:  929) --> 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.
```

The maximum value that the base can be set to, such that the address space fits into physical memory in its entirety, is 16284. This is because, here the limit value is 100 and physical memory is 16k (16384). For the whole address space to fit into physical memory, we need to have base + limit <= physical memory. Base can have a maximum value of physical memory - limit. So, the maximum value of base here = 16384 - 100 = 16284.

4) For this problem, I am setting Physical Memory Size to 1GB and Virtual Address Space Size to 16MB. And here, I am doing 1) with seed value 0 and 2) parts above.

For 1) with seed value 0:

```
C:\Users\Rahul\Documents\SEM-6\OPERATING_SYSTEMS\OS Lab\Lab_7>py -2 relocation.py -s 0 -a 16m -p 1g -c

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

Base-and-Bounds register information:

  Base   : 0x3082532f (decimal 813847343)
  Limit  : 7736065

Virtual Address Trace
  VA  0: 0x006baa94 (decimal: 7056020) --> VALID: 0x30edfdc3 (decimal: 820903363)
  VA  1: 0x0042485e (decimal: 4343902) --> VALID: 0x30c49b8d (decimal: 818191245)
  VA  2: 0x0082e2e6 (decimal: 8577766) --> SEGMENTATION VIOLATION
  VA  3: 0x0067a9c3 (decimal: 6793667) --> VALID: 0x30e9fcf2 (decimal: 820641010)
  VA  4: 0x00c8a706 (decimal: 13149958) --> SEGMENTATION VIOLATION
```

ARGUMENTS seed=0, address space size=16m, physical memory size=1g
Base-and-Bounds register information:
Base : 0x3082532f (decimal 813847343)
Limit : 7736065

Virtual Address Trace
VA 0: 0x006baa94 (decimal: 7056020) --> IN BOUNDS
Corresponding Physical Address: 0x30edfdc3 (decimal: 820903363)
VA 1: 0x0042485e (decimal: 4343902) --> IN BOUNDS
Corresponding Physical Address: 0x30c49b8d (decimal: 818191245)
VA 2: 0x0082e2e6 (decimal: 8577766) --> OUT OF BOUNDS
VA 3: 0x0067a9c3 (decimal: 6793667) --> IN BOUNDS
Corresponding Physical Address: 0x30e9fcf2 (decimal: 820641010)
VA 4: 0x00c8a706 (decimal: 13149958) --> OUT OF BOUNDS

For 2): Here is the details and Virtual Address Trace that we have got:

```
C:\Users\Rahul\Documents\SEM-6\OPERATING_SYSTEMS\OS Lab\Lab_7>py -2 relocation.py -s 0 -n 10 -a 16m -p 1g

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

Base-and-Bounds register information:

  Base   : 0x3082532f (decimal 813847343)
  Limit  : 7736065

Virtual Address Trace
  VA  0: 0x006baa94 (decimal: 7056020) --> PA or segmentation violation?
  VA  1: 0x0042485e (decimal: 4343902) --> PA or segmentation violation?
  VA  2: 0x0082e2e6 (decimal: 8577766) --> PA or segmentation violation?
  VA  3: 0x0067a9c3 (decimal: 6793667) --> PA or segmentation violation?
  VA  4: 0x00c8a706 (decimal: 13149958) --> PA or segmentation violation?
  VA  5: 0x004da5e7 (decimal: 5088743) --> PA or segmentation violation?
  VA  6: 0x007a0242 (decimal: 7995970) --> PA or segmentation violation?
  VA  7: 0x00955886 (decimal: 9787526) --> PA or segmentation violation?
  VA  8: 0x00e87a16 (decimal: 15235606) --> PA or segmentation violation?
  VA  9: 0x00813328 (decimal: 8467240) --> 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.
```

ARG seed 0
ARG address space size 16m
ARG phys mem size 1g
Base-and-Bounds register information:
Base : 0x3082532f (decimal 813847343)
Limit : 7736065

Virtual Address Trace

VA 0: 0x006baa94 (decimal: 7056020) --> PA or segmentation violation?

VA 1: 0x0042485e (decimal: 4343902) --> PA or segmentation violation?

VA 2: 0x0082e2e6 (decimal: 8577766) --> PA or segmentation violation?

VA 3: 0x0067a9c3 (decimal: 6793667) --> PA or segmentation violation?

VA 4: 0x00c8a706 (decimal: 13149958) --> PA or segmentation violation?

VA 5: 0x004da5e7 (decimal: 5088743) --> PA or segmentation violation?

VA 6: 0x007a0242 (decimal: 7995970) --> PA or segmentation violation?

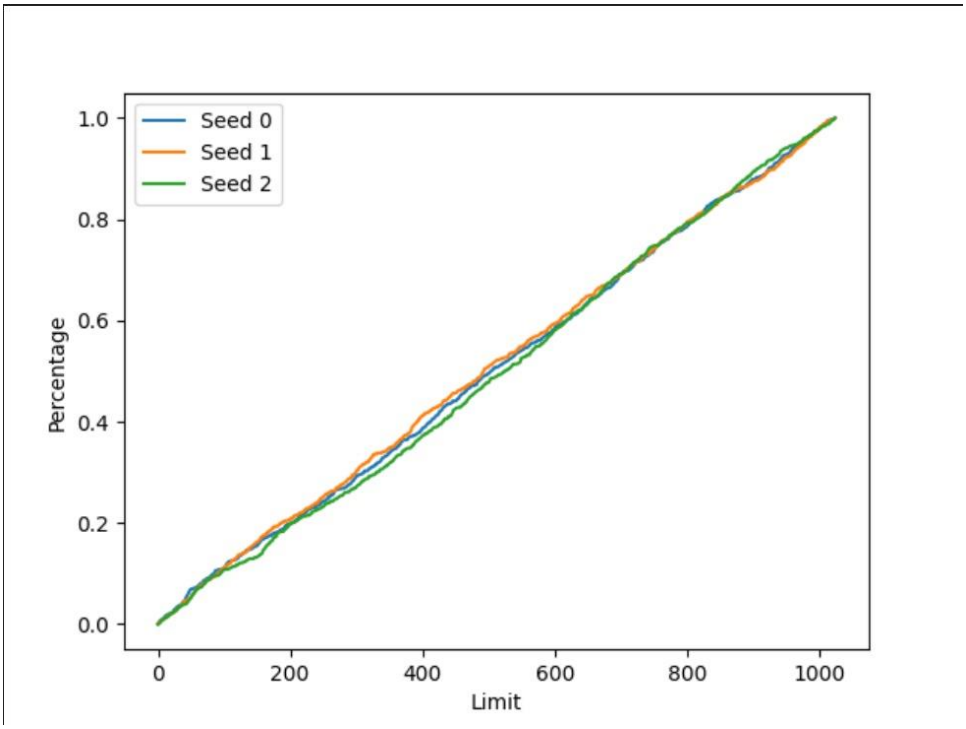
VA 7: 0x00955886 (decimal: 9787526) --> PA or segmentation violation?

VA 8: 0x00e87a16 (decimal: 15235606) --> PA or segmentation violation?

VA 9: 0x00813328 (decimal: 8467240) --> PA or segmentation violation?

We can clearly see that the maximum virtual address generated is 15235606. To make all these virtual addresses valid, the limit value must be set to 15235607 or greater.

5)



x-axis: Max limit is 1024 (1kb)

y-axis: fraction of valid translations.

Q2.

1)

segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0

```
C:\Users\Rahul\Documents\SEM-6\OPERATING_SYSTEMS\OS Lab\Lab_7>py -2 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.
```

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) --> IN BOUNDS in SEG1: 0x000001ec (decimal: 492)

VA 1: 0x00000061 (decimal: 97) --> OUT OF BOUNDS (SEG1)

VA 2: 0x00000035 (decimal: 53) --> OUT OF BOUNDS (SEG0)

VA 3: 0x00000021 (decimal: 33) --> OUT OF BOUNDS (SEG0)

VA 4: 0x00000041 (decimal: 65) --> OUT OF BOUNDS (SEG1)

segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1

```
C:\Users\Rahul\Documents\SEM-6\OPERATING_SYSTEMS\OS Lab\Lab_7>py -2 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.
```

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) --> IN BOUNDS in SEG0: 0x00000011 (decimal: 17)

VA 1: 0x0000006c (decimal: 108) --> IN BOUNDS in SEG1: 0x000001ec (decimal: 492)

VA 2: 0x00000061 (decimal: 97) --> OUT OF BOUNDS (SEG1)

VA 3: 0x00000020 (decimal: 32) --> OUT OF BOUNDS (SEG0)

VA 4: 0x0000003f (decimal: 63) --> OUT OF BOUNDS (SEG0)

segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2

```
C:\Users\Rahul\Documents\SEM-6\OPERATING_SYSTEMS\OS Lab\Lab_7>py -2 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.
```

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) --> IN BOUNDS in SEG1: 0x000001fa (decimal: 506)

VA 1: 0x00000079 (decimal: 121) --> IN BOUNDS in SEG1: 0x000001f9 (decimal: 505)

VA 2: 0x00000007 (decimal: 7) --> IN BOUNDS in SEG0: 0x00000007 (decimal: 7)

VA 3: 0x0000000a (decimal: 10) --> IN BOUNDS in SEG0: 0x0000000a (decimal: 10)

VA 4: 0x0000006a (decimal: 106) --> OUT OF BOUNDS (SEG1)

Here, we have two segments, i.e., segment 0 and segment 1. And, we have segment 0 growing positive, whereas segment 1 growing negative. We can distinguish the segments with the top bit of the virtual address. Segment 0's top bit is 0 and segment 1's top bit is 1. For positive growing segments, virtual address is valid only if it is in between 0 and limit-1 (both inclusive).

Physical Address = Base + Virtual Address.

As we have base of segment 0 as 0, limit of segment 0 as 20, valid virtual addresses are 0, 1, …, 19 and their corresponding physical addresses are 0, 1, …, 19 respectively.

For the negative growing segment, virtual address is valid only if it is in between virtual address size - limit and virtual address size - 1(both inclusive).

Physical Address = Base - Virtual Address Size + Virtual Address.

For segment 1, we have base as 512, limit as 20 and Virtual Address as 128. So, valid virtual addresses are 127, 126, …, 108 and their corresponding physical addresses are 511, 510, 509, …, 492 respectively.

2)

```
C:\Users\Rahul\Documents\SEM-6\OPERATING_SYSTEMS\OS Lab\Lab_7>py -2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -c true -A 19
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)


C:\Users\Rahul\Documents\SEM-6\OPERATING_SYSTEMS\OS Lab\Lab_7>py -2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -c true -A 20
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)
```

The highest legal virtual address in segment 0 = 19.

Because the highest legal virtual address in a positive growing segment = limit - 1.

The lowest legal virtual address in segment 1 = 108.

Because the lowest legal virtual address in a negative growing segment = virtual address size - limit.

```
C:\Users\Rahul\Documents\SEM-6\OPERATING_SYSTEMS\OS Lab\Lab_7>py -2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -c true -A 107
ARG seed 0
ARG address space size 128
ARG phys mem size 512

Segment register information:

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

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

Virtual Address Trace
  VA  0: 0x0000006b (decimal:  107) --> SEGMENTATION VIOLATION (SEG1)


C:\Users\Rahul\Documents\SEM-6\OPERATING_SYSTEMS\OS Lab\Lab_7>py -2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -c true -A 108
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)
```

Lowest illegal virtual address = 20.
Highest illegal virtual address = 107.
We can set the value of -A to whatever virtual address we want to check and -c to True to obtain whether that virtual address is valid or not.

3)

```
C:\Users\Rahul\Documents\SEM-6\OPERATING_SYSTEMS\OS Lab\Lab_7>py -2 segmentation.py -a 16 -p 128 --b0 0 --l0 2 --b1 128 --l1 2 -A 0,1,2,3,4,5,6,7,8,9,1
0,11,12,13,14,15 -c true
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)
```

We need to set limit values as 2 for both the segments, i.e., l0 = 2, l1 = 2.
For base values, we need to take two values such that both the segments do not overlap.
I took b0 = 0, b1 = 128 and this is the output that I got.
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 get 90% of the randomly generated virtual addresses valid, we need to have valid address space length = 90% of the total virtual address space. But, in this simulation, we have valid address space length = limit of segment 0 + limit of segment 1.
Therefore, we need to give l0 and l1 values such that l0 + l1 = 0.9 * virtual address space.

In a system with an address space of 128 bytes divided into two segments, where Segment 0 covers addresses 0-63 and Segment 1 covers addresses 64-127, each segment should roughly cover 45% of the total address space to achieve approximately 90% coverage. With a physical memory size of 256 bytes, 100 virtual addresses were generated, resulting in around 89% valid virtual addresses.

5) To get 0% of the randomly generated virtual addresses valid, we need to have valid address space length = 0% of the total virtual address space. But, in this simulation, we have valid address space length = limit of segment 0 + limit of segment 1.
Therefore, we need to give l0 and l1 values such that l0 + l1 = 0 * virtual address space. So, we need to set l0 = 0, l1 = 0.

```
C:\Users\Rahul\Documents\SEM-6\OPERATING_SYSTEMS\OS Lab\Lab_7>py -2 segmentation.py -a 16 -p 128 -b 0 -l 0 -B 128 -L 0 -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) : 0x00000080 (decimal 128)
  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)
```

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) : 0x00000080 (decimal 128)
  Segment 1 limit 0

Virtual Address Trace
  VA 0: 0x000001ae (decimal: 13) --> SEGMENTATION VIOLATION (SEG1)
  VA 1: 0x00000109 (decimal: 12) --> SEGMENTATION VIOLATION (SEG1)
  VA 2: 0x0000020b (decimal: 6) --> SEGMENTATION VIOLATION (SEG0)
  VA 3: 0x0000019e (decimal: 4) --> SEGMENTATION VIOLATION (SEG0)
  VA 4: 0x00000322 (decimal: 8) --> SEGMENTATION VIOLATION (SEG1)

Q3.

Let's suppose that we have 'v' no. of bits in the virtual address, 'p' be the page size (in bytes), 'e' be the size of each page table entry (in bytes). As we have p as the page size, no. of offset bits = lg p. Therefore, we have v - (lg p) bits in the VPN. No. of possible VPNs = 2^(v- (lg p)). As we need 1 entry for each VPN, we must have page table size = e*(2^(v-(lg p))).

Case 1: paging-linear-size.py -v 16 -e 4 -p 1k
The number of bits in the virtual address: 16
The page size: 1024 bytes
The size of each page table entry, which is: 4
Thus, the number of bits needed in the offset: 10
Which leaves this many bits for the VPN: 6
To compute the size of the linear page table, we need to know:
- The no. of entries in the table, which is 2^ (num of VPN bits): 64

Page Table Size: 256 bytes

```
C:\Users\Rahul\Documents\SEM-6\OPERATING_SYSTEMS\OS Lab\Lab_7>py -2 paging-linear-size.py -v 16 -e 4 -p 1k -c
ARG bits in virtual address 16
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: 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:

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): 64.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  256 bytes
  in KB: 0.25
  in MB: 0.000244140625
```

Case 2: paging-linear-size.py -v 20 -e 8 -p 2k

The number of bits in the virtual address: 20

The page size: 2048 bytes

The size of each page table entry, which is: 8

Thus, the number of bits needed in the offset: 11

Which leaves this many bits for the VPN: 9

To compute the size of the linear page table, we need to know:

- The no. of entries in the table, which is 2^ (num of VPN bits): 512

Page Table Size: 4096 bytes

```
C:\Users\Rahul\Documents\SEM-6\OPERATING_SYSTEMS\OS Lab\Lab_7>py -2 paging-linear-size.py -v 20 -e 8 -p 2k -c
ARG bits in virtual address 20
ARG page size 2k
ARG pte size 8

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

The number of bits in the virtual address: 20
The page size: 2048 bytes
Thus, the number of bits needed in the offset: 11
Which leaves this many bits for the VPN: 9
Thus, a virtual address looks like this:

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): 512.0
- The size of each page table entry, which is: 8
And then multiply them together. The final result:
  4096 bytes
  in KB: 4.0
  in MB: 0.00390625
```

Case 3: paging-linear-size.py -v 24 -e 16 -p 2k

The number of bits in the virtual address: 24
The page size: 2048 bytes
The size of each page table entry, which is: 16
Thus, the number of bits needed in the offset: 11
Which leaves this many bits for the VPN: 13
To compute the size of the linear page table, we need to know:
– The no. of entries in the table, which is 2^(num of VPN bits): 8192
Page Table Size: 131072 bytes

```
C:\Users\Rahul\Documents\SEM-6\OPERATING_SYSTEMS\OS Lab\Lab_7>py -2 paging-linear-size.py -v 24 -e 16 -p 2k -c
ARG bits in virtual address 24
ARG page size 2k
ARG pte size 16

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

The number of bits in the virtual address: 24
The page size: 2048 bytes
Thus, the number of bits needed in the offset: 11
Which leaves this many bits for the VPN: 13
Thus, a virtual address looks like this:

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): 8192.0
- The size of each page table entry, which is: 16
And then multiply them together. The final result:
  131072 bytes
  in KB: 128.0
  in MB: 0.125
```

Q4.

1)

Case 1: As the address space increases, the Page Table Size also increases. paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0 = 1024

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

Virtual Address Trace
```

paging-linear-translate.py -P 1k -a 2m -p 512m -v -n 0 = 2048

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

Virtual Address Trace
```

paging-linear-translate.py -P 1k -a 4m -p 512m -v -n 0 = 4096

```
[    4088]   0x00000000
[    4089]   0x80078d9a
[    4090]   0x8006ca8e
[    4091]   0x800160f8
[    4092]   0x80015abc
[    4093]   0x8001483a
[    4094]   0x00000000
[    4095]   0x8002e298

Virtual Address Trace
```

Case 2: As the Page size increases, the Page Table Size decreases.
paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0 = 1024
paging-linear-translate.py -P 2k -a 1m -p 512m -v -n 0 = 512
paging-linear-translate.py -P 4k -a 1m -p 512m -v -n 0 = 256

2)

For -P 1k -a 16k -p 32k -v -u 0, we have no pages that are valid.

```
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
```

For -P 1k -a 16k -p 32k -v -u 25, we have 25% pages valid.

```
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
```

For -P 1k -a 16k -p 32k -v -u 50, we have 50% pages valid.

```
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
```

For -P 1k -a 16k -p 32k -v -u 75, we have 75% pages valid.

```
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
```

For -P 1k -a 16k -p 32k -v -u 100, we have 100% pages valid.

```
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
```

As we increase the percentage of valid pages, no. of pages allocated in the Page Frame Table increases as we have no. of Valid pages in the Page Descriptor Table increasing.

3)

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

```
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)
```

We set the page size to be 8 bytes. It is a bit small; it leads to a lot of page entries on the page table. It leads to external fragmentation.

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

```
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)
```

We set page size to be 8KB = 8192 bytes.

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

```
  [     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]
```

We set the page size to be 1MB. It is too large; it leads to internal fragmentation.

Case a and case c are unrealistic.

4) Some of the cases where the program doesn't work anymore.

i) If address space is greater than physical memory.
ii) If page size is greater than address space or physical memory.
iii) The physical space size and virtual address space size must be non-negative
iv) Address space must be a multiple of the page size, and physical memory must be a multiple of the size