190010033 | Pranav Kanire

13/03/2022

OS LAB: ASSIGNMENT - 6

1 Q1

1.

Please find below the translations for the virtual addresses that are in bound:

Seed 1:

```
pranav@pranav:~/Desktop/OS_lab/6$ 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

pranav@pranav:~/Desktop/OS_lab/6$
```

Seed 2:

```
pranav@pranav:~/Desktop/OS_lab/6$ 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
```

Seed 3:

```
pranav@pranav:~/Desktop/OS_lab/6$ 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)
```

2.

We have to set -l value to max_value + 1, to ensure all the generated virtual addresses are within bounds.

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 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.
```

Here the maximum value is 929. So, -l value should be set to **930**.

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 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.

-l 100: set limit as 100
Physical memory size: 16k = 16*1024 = 16384
Maximum value of base can be: Phy mem size - limit = 16384 -100 = 16284

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 relocation.py -s 1 -n 10 -l 100 -b 16284 -c

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

Base-and-Bounds register information:

  Base   : 0x00003f9c (decimal 16284)
  Limit  : 100

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

4.

Here, address space size: 32k
phy mem size: 64k = 64*1024 = 65536
Max base: phy mem size - limit = 65536 - 100 = 65436

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 relocation.py -s 1 -n 10 -l 100 -b 65436 -a 32k -p 64k -c

ARG seed 1
ARG address space size 32k
ARG phys mem size 64k

Base-and-Bounds register information:

  Base   : 0x0000ff9c (decimal 65436)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x00001132 (decimal: 4402) --> SEGMENTATION VIOLATION
  VA  1: 0x00006c78 (decimal: 27768) --> SEGMENTATION VIOLATION
  VA  2: 0x000061c3 (decimal: 25027) --> SEGMENTATION VIOLATION
  VA  3: 0x000020a6 (decimal: 8358) --> SEGMENTATION VIOLATION
  VA  4: 0x00003f6a (decimal: 16234) --> SEGMENTATION VIOLATION
  VA  5: 0x00003988 (decimal: 14728) --> SEGMENTATION VIOLATION
  VA  6: 0x00005367 (decimal: 21351) --> SEGMENTATION VIOLATION
  VA  7: 0x000064f4 (decimal: 25844) --> SEGMENTATION VIOLATION
  VA  8: 0x00000c03 (decimal: 3075) --> SEGMENTATION VIOLATION
  VA  9: 0x000003a0 (decimal:  928) --> SEGMENTATION VIOLATION
```

2 Q2

1.

Used -c flag for translation:

```
pranav@pranav:~/Desktop/OS_lab/6$ 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)
```

```
pranav@pranav:~/Desktop/OS_lab/6$ 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)
```

```
pranav@pranav:~/Desktop/OS_lab/6$ 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)
```

2.

With same parameters as in above

Highest legal virtual address in segment 0: 19
Lowest legal virtual address in segment 1: 108
Highest illegal virtual address in entire address space: 107
Lowest illegal virtual address in entire address space: 20

Confirming same with flag -A :

```
pranav@pranav:~/Desktop/OS_lab/6$ 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)
```

```
pranav@pranav:~/Desktop/OS_lab/6$ 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)
```

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0 -A 107,20 -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: 0x0000006b (decimal:  107) --> SEGMENTATION VIOLATION (SEG1)
  VA  1: 0x00000014 (decimal:   20) --> SEGMENTATION VIOLATION (SEG0)
```

3.

Since address space is 16 I chose to set b0 and b1 and 0 and 16 respectively.
After setting l0 and l1 both as 2 I got the desired results.(l0 = 2 & l1 = 2).

```
pranav@pranav:~/Desktop/OS_lab/6$ 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 16 --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) : 0x00000010 (decimal 16)
  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: 0x0000000e (decimal:   14)
  VA 15: 0x0000000f (decimal:   15) --> VALID in SEG1: 0x0000000f (decimal:   15)
```

4.

For this we need to set bound to be ~90% of the address space.
Here, address space: 1024 and physical memory size: 2048 and segments: 2
So, 90% of (1024/2) is around 461.

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 segmentation.py -a 1024 -p 2048 --b0 0 --l0 461 --b1 2048 --l1 461 -c -n 100
ARG seed 0
ARG address space size 1024
ARG phys mem size 2048

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 461

  Segment 1 base  (grows negative) : 0x00000800 (decimal 2048)
  Segment 1 limit                  : 461

Virtual Address Trace
  VA  0: 0x00000360 (decimal:  864) --> VALID in SEG1: 0x00000760 (decimal: 1888)
  VA  1: 0x00000308 (decimal:  776) --> VALID in SEG1: 0x00000708 (decimal: 1800)
  VA  2: 0x000001ae (decimal:  430) --> VALID in SEG0: 0x000001ae (decimal:  430)
  VA  3: 0x00000109 (decimal:  265) --> VALID in SEG0: 0x00000109 (decimal:  265)
  VA  4: 0x0000020b (decimal:  523) --> SEGMENTATION VIOLATION (SEG1)
```

5.

Run with flags: -l 0 -L 0
(setting address space limit and physical memory limit)

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 segmentation.py -l 0 -L 0 -c
ARG seed 0
ARG address space size 1k
ARG phys mem size 16k

Segment register information:

  Segment 0 base  (grows positive) : 0x0000360b (decimal 13835)
  Segment 0 limit                  : 0

  Segment 1 base  (grows negative) : 0x00003082 (decimal 12418)
  Segment 1 limit                  : 0

Virtual Address Trace
  VA  0: 0x000001ae (decimal:  430) --> SEGMENTATION VIOLATION (SEG0)
  VA  1: 0x00000109 (decimal:  265) --> SEGMENTATION VIOLATION (SEG0)
  VA  2: 0x0000020b (decimal:  523) --> SEGMENTATION VIOLATION (SEG1)
  VA  3: 0x0000019e (decimal:  414) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x00000322 (decimal:  802) --> SEGMENTATION VIOLATION (SEG1)
```

3 Q3

1.

Initially:

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 paging-linear-size.py -p 1024 -e 16 -v 32 -c
ARG bits in virtual address 32
ARG page size 1024
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: 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: 16
And then multiply them together. The final result:
  67108864 bytes
  in KB: 65536.0
  in MB: 64.0
```

Here, page size is 1024 so log2(1024) = 10 bits are reserved for offset. Now, 32 - 10 = 22 bits will be used to identify each page uniquely.
So, total number of pages = 2^22
Size of each page table entry = 16
Space required = size of PTE * total number of pages = 16 * 2^22 = 67108864 bytes

-> different page size

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 paging-linear-size.py -p 512 -e 16 -v 32 -c
ARG bits in virtual address 32
ARG page size 512
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: 512 bytes
Thus, the number of bits needed in the offset: 9
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

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: 16
And then multiply them together. The final result:
  134217728 bytes
  in KB: 131072.0
  in MB: 128.0
```

Here, page size is 512 so log2(512) = 9 bits are reserved for offset. Now, 32 - 9 = 23 bits will be used to identify each page uniquely.
So, total number of pages = 2^23
Size of each page table entry = 16
Space required = size of PTE * total number of pages = 16 * 2^23 = 134217728 bytes
Space required increases as page size decreases.

-> different page table entry size

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 paging-linear-size.py -p 1024 -e 8 -v 32 -c
ARG bits in virtual address 32
ARG page size 1024
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: 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: 8
And then multiply them together. The final result:
  33554432 bytes
  in KB: 32768.0
  in MB: 32.0
```

Here, page size is 1024 so log2(1024) = 10 bits are reserved for offset. Now, 32 - 10 = 22 bits will be used to identify each page uniquely.
So, total number of pages = 2^22
Size of each page table entry = 8
Space required = size of PTE * total number of pages = 8 * 2^22 = 33554432 bytes
Space required decreases as table entry size decreases.

-> different number of bits in virtual address space

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 paging-linear-size.py -p 1024 -e 16 -v 16 -c
ARG bits in virtual address 16
ARG page size 1024
ARG pte size 16

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: 16
And then multiply them together. The final result:
  1024 bytes
  in KB: 1.0
  in MB: 0.0009765625
```

Here, page size is 1024 so log2(1024) = 10 bits are reserved for offset. Now, 16 - 10 = 6 bits will be used to identify each page uniquely.
So, total number of pages = 2^6 = 64
Size of each page table entry = 16
Space required = size of PTE * total number of pages = 16 * 64 = 1024 bytes
Space required decreases as bits in virtual address space decreases.

4 Q4

1.

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 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
  [       7]   0x80024134
  [       8]   0x8004f26b
  [       9]   0x00000000
```

```
  [    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
```

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 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
  [       7]   0x80024134
  [       8]   0x8004f26b
  [       9]   0x00000000
```

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

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 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
```

```
[     4084]   0x00000000
[     4085]   0x80006de5
[     4086]   0x8004f319
[     4087]   0x8003f14c
[     4088]   0x00000000
[     4089]   0x80078d9a
[     4090]   0x8006ca8e
[     4091]   0x800160f8
[     4092]   0x80015abc
[     4093]   0x8001483a
[     4094]   0x00000000
[     4095]   0x8002e298
```

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 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
```

```
[     1015]   0x00000000
[     1016]   0x00000000
[     1017]   0x00000000
[     1018]   0x00000000
[     1019]   0x8002e9c9
[     1020]   0x00000000
[     1021]   0x00000000
[     1022]   0x00000000
[     1023]   0x00000000
```

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 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
```

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

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 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
```

```
[      249]   0x00000000
[      250]   0x00000000
[      251]   0x8001efec
[      252]   0x8001cd5b
[      253]   0x800125d2
[      254]   0x80019c37
[      255]   0x8001fb27
```

Because we require more pages to cover the entire address space, the size of the page table grows as the address space grows.
Because we require fewer pages (due to their larger size) to fill the entire address space, the page table size lowers as page sizes grow.
Because most processes only consume a little amount of memory, we should avoid using really large pages in general since they consume a lot of memory.

2.

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 0 -c
ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 1k
ARG verbose True
ARG addresses -1


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

Page Table (from entry 0 down to the max size)
  [       0]   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)
```

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 25 -c
ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 1k
ARG verbose True
ARG addresses -1


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

Page Table (from entry 0 down to the max size)
  [       0]   0x80000018
  [       1]   0x00000000
  [       2]   0x00000000
  [       3]   0x00000000
  [       4]   0x00000000
  [       5]   0x80000009
  [       6]   0x00000000
  [       7]   0x00000000
  [       8]   0x80000010
  [       9]   0x00000000
  [      10]   0x80000013
  [      11]   0x00000000
  [      12]   0x8000001f
  [      13]   0x8000001c
  [      14]   0x00000000
  [      15]   0x00000000

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

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 50 -c
ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 1k
ARG verbose True
ARG addresses -1


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

Page Table (from entry 0 down to the max size)
  [       0]   0x80000018
  [       1]   0x00000000
  [       2]   0x00000000
  [       3]   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]
```

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 75 -c
ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 1k
ARG verbose True
ARG addresses -1


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

Page Table (from entry 0 down to the max size)
  [       0]   0x80000018
  [       1]   0x80000008
  [       2]   0x8000000c
  [       3]   0x80000009
  [       4]   0x80000012
  [       5]   0x80000010
  [       6]   0x8000001f
  [       7]   0x8000001c
  [       8]   0x80000017
  [       9]   0x80000015
  [      10]   0x80000003
  [      11]   0x80000013
  [      12]   0x8000001e
  [      13]   0x8000001b
  [      14]   0x80000019
  [      15]   0x80000000

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

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 100 -c
ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 1k
ARG verbose True
ARG addresses -1


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

Page Table (from entry 0 down to the max size)
  [       0]   0x80000018
  [       1]   0x80000008
  [       2]   0x8000000c
  [       3]   0x80000009
  [       4]   0x80000012
  [       5]   0x80000010
  [       6]   0x8000001f
  [       7]   0x8000001c
  [       8]   0x80000017
  [       9]   0x80000015
  [      10]   0x80000003
  [      11]   0x80000013
  [      12]   0x8000001e
  [      13]   0x8000001b
  [      14]   0x80000019
  [      15]   0x80000000

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

From the above images we can conclude that, as the percentage of pages that are allocated in each address space increases, as more pages become valid more and more memory operations become valid. But free space decreases.

3.

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 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)
```

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 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)
```

```
pranav@pranav:~/Desktop/OS_lab/6$ python2 paging-linear-translate.py -P 1m -a 256m -p 512m -v -s 3 -c
ARG seed 3
ARG address space size 256m
ARG phys mem size 512m
ARG page size 1m
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]   0x800000bd
  [       2]   0x80000140
  [       3]   0x00000000
  [       4]   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]
```

Here I think the reality of the parameters depends on the physical memory size or address space. So in my opinion the first and third are unrealistic. First seems to be very small and the third seems to be very big.

4.

I tried different values for different flags like memory size, address space etc. From these I were able to find some limits for this program, which are as follows:

Program doesn't work when:

- page size is greater than address-space.
- Any memory/size is negative
- address space size is greater than the physical memory.
- physical memory size is not multiple of page size.
- address space is not multiple of page size.