CS 314: Operating Systems Laboratory

Assignment 7

Ashwin Waghmare

210010060

Part 1

1) A virtual address generated by a process is in bound if VA<Limit

For seed 1, bound (Limit) = 290

Serial No.	Virtual Address (VA)	Within bounds?
0	782	No
1	261	Yes
2	507	No
3	460	No
4	667	No

For serial number 1: Physical Address = Virtual Address + Base

= 261 + 13884

= 14145

For seed 2, bound = 500

Serial No.	Virtual Address	Within bounds?
0	57	Yes
1	86	Yes
2	855	No
3	753	No
4	685	No

For serial number 0: Physical Address = Virtual Address + Base

= 57 + 15529

= 15586

For serial number 1: Physical Address = Virtual Address + Base

= 86 + 15529

= 15615

For seed 3, bound = 316

Serial No.	Virtual Address	Within bounds?
0	378	No
1	618	No
2	640	No
3	67	Yes
4	13	Yes

For serial number 3: Physical Address = Virtual Address + Base

= 67 + 8916

= 8983

For serial number 4: Physical Address = Virtual Address + Base

= 13 + 8916

= 8929

2) The value of the bound has to be set to 930, since the maximum virtual address is 929, to ensure that all the generated virtual addresses are within bounds.

3) The maximum value of base = Physical Address – Virtual Address

= 16384 – 100

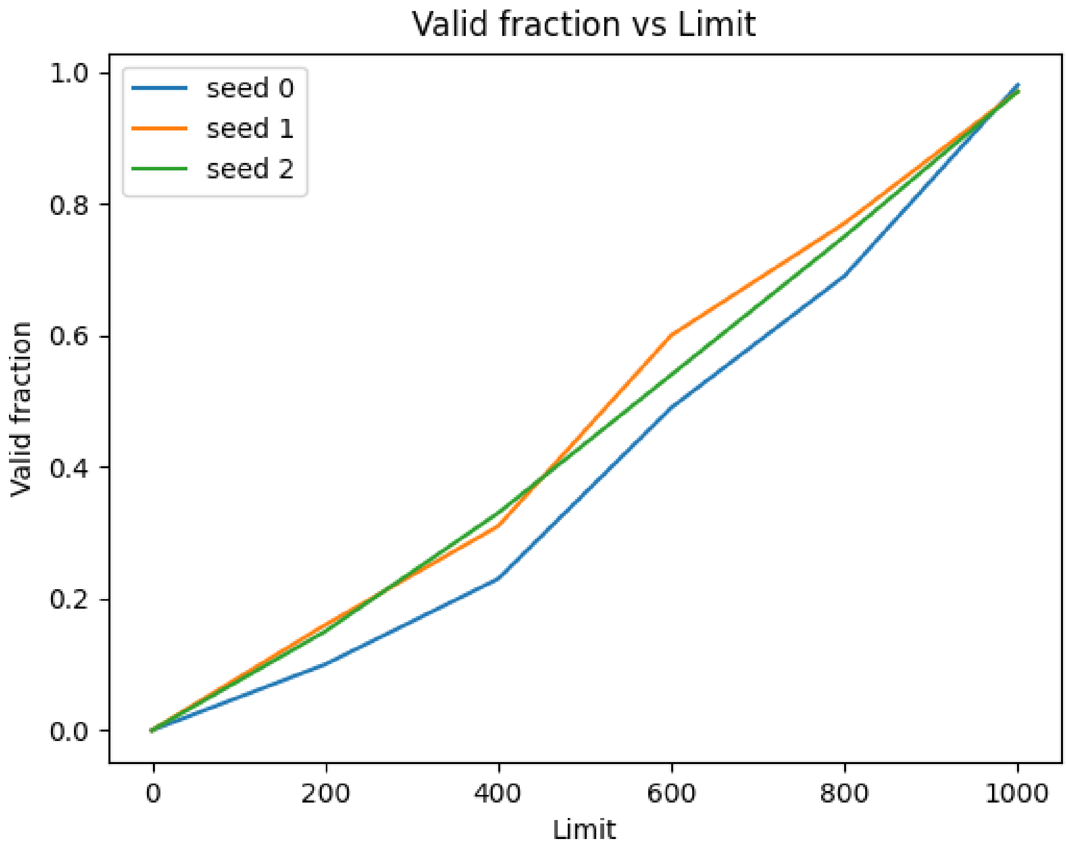
= 16284.

4) The address space is set to 32k and physical address is set to 1g

./relocation.py -s 1 -n 10 -l 100 -a 32m -p 1g -c

```
ARG seed 1
ARG address space size 32m
ARG phys mem size 1g

Base-and-Bounds register information:

  Base   : 0x08996c7c (decimal 144272508)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x01b1e2d5 (decimal: 28435157) --> SEGMENTATION VIOLATION
  VA  1: 0x01870d77 (decimal: 25628023) --> SEGMENTATION VIOLATION
  VA  2: 0x00829868 (decimal: 8558696) --> SEGMENTATION VIOLATION
  VA  3: 0x00fda9aa (decimal: 16624042) --> SEGMENTATION VIOLATION
  VA  4: 0x00e623b1 (decimal: 15082417) --> SEGMENTATION VIOLATION
  VA  5: 0x014d9d98 (decimal: 21863832) --> SEGMENTATION VIOLATION
  VA  6: 0x0193d38c (decimal: 26465164) --> SEGMENTATION VIOLATION
  VA  7: 0x00300e5d (decimal: 3149405) --> SEGMENTATION VIOLATION
  VA  8: 0x000e838f (decimal: 951183) --> SEGMENTATION VIOLATION
  VA  9: 0x01abe967 (decimal: 28043623) --> SEGMENTATION VIOLATION
```

5) Here is a graph of valid fraction of randomly generated virtual addresses as a function of bounds register.



Part 2

1) Here address space size is 128 and physical memory size is 512. If vaddr < asize/2 Then virtual address falls under segment 0,

• In bound : vaddr < limit0 and paddr = vaddr + base0

• Out of bound : vaddr >= limit0

If vaddr >= asize/2 Then virtual address falls under segment 1,

• In bound : asize − limit1 =< vaddr and paddr = base1 − (asize − vaddr)

• Out of bound : asize − limit1 > vaddr

For seed 0:

Serial No	Virtual Address	Segment No	Within bounds?
0	108	1	Yes
1	97	1	No
2	53	0	No
3	35	0	No

4	65	1	No

For serial number 0: Physical Address=Physical address size - (address space size – virtual address)

$$= 512 - (128 - 108)$$

$$= 492$$

For seed 1:

Serial No	Virtual Address	Segment No	Within bounds?
0	17	0	Yes
1	108	1	Yes
2	97	1	No
3	32	0	No
4	63	0	No

For serial number 0: Physical Address= 0 + Virtual Address

$$= 17$$

For serial number 1: Physical Address=Physical address size - (address space size – virtual address)

$$= 512 - (128 - 108)$$

$$= 492$$

For seed 2:

Serial No	Virtual Address	Segment No	Within bounds?
0	122	1	Yes
1	121	1	Yes
2	7	0	Yes
3	10	0	Yes
4	106	1	No

For serial number 0: Physical Address=Physical address size - (address space size – virtual address)

$$= 512 - (128 - 122)$$

$$= 506$$

For serial number 1: Physical Address=Physical address size - (address space size – virtual address)

$$= 512 - (128 - 121)$$

$$= 505$$

For serial number 2: Physical Address= 0 + Virtual Address

$$= 7$$

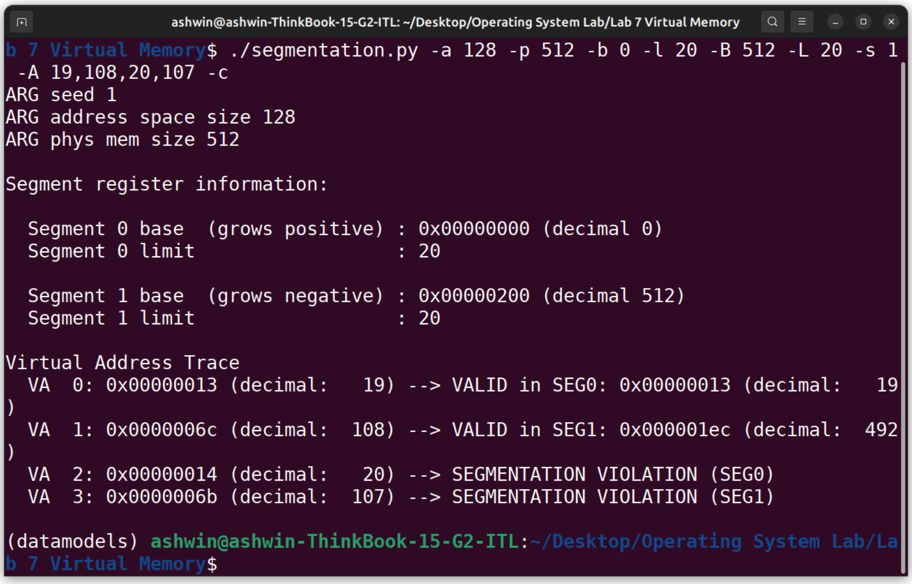
For serial number 3: Physical Address= 0 + Virtual Address

= 10

2) Highest legal virtual address in segment 0 is 19. Lowest legal virtual address in segment 1 is 108. Lowest illegal address 20. Highest illegal address 10.

To verify whether these values are correct, we execute the following command:

```
./segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1 -A 19,108,20,107 -c
```



3) According to the question, virtual addresses 0, 1, 14, 15 are valid while rest of them are invalid. The first segment will contain virtual addresses 0-7 so base0 should be 0 and we want only 0 and 1 to be valid in this segment thus limit0 should be set to 2. Similarly, virtual addresses 8-15 belong to the second segment so base 1 should be 15 with a limit of 2.

Also proof the stated solution is given below:

```
ashwin@ashwin-ThinkBook-15-G2-ITL: ~/Desktop/Operating System Lab/Lab 7 Virtual Memory
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.

(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/Lab 7 Virtual Memory$ ./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 15 --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) : 0x0000000f (decimal 15)
  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: 0x0000000d (decimal:   13)
  VA 15: 0x0000000f (decimal:   15) --> VALID in SEG1: 0x0000000e (decimal:   14)
(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/Lab 7 Virtual Memory$
```

4) Let us take an address space of size 20 bytes and a physical memory os size 100 bytes. To generate a problem where roughly 90% of the generated virtual addresses are valid, we should set the valid memory size roughly 90% of the actual virtual memory size. Now, size of segment 0 + size of segment 1 = 90% of the address space = 0.9 * 20 = 18. Thus we could set base0 = 0, base1 = 20, limit0 = 9, limit1 = 9. In the following screenshot we can observe that 90% of the virtual addresses generated are valid. Here, base and limit parameters are important to obtain desired results.

```
(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/Lab 7 Virtual M
emory$ ./segmentation.py -a 20 -p 100 --b0 0 --l0 9 --b1 20 --l1 9 -c -n 10 -s 1
ARG seed 1
ARG address space size 20
ARG phys mem size 100

Segment register information:

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

  Segment 1 base  (grows negative) : 0x00000014 (decimal 20)
  Segment 1 limit                  : 9

Virtual Address Trace
  VA  0: 0x00000002 (decimal:    2) --> VALID in SEG0: 0x00000002 (decimal:    2)
  VA  1: 0x00000010 (decimal:   16) --> VALID in SEG1: 0x00000010 (decimal:   16)
  VA  2: 0x0000000f (decimal:   15) --> VALID in SEG1: 0x0000000f (decimal:   15)
  VA  3: 0x00000005 (decimal:    5) --> VALID in SEG0: 0x00000005 (decimal:    5)
  VA  4: 0x00000009 (decimal:    9) --> SEGMENTATION VIOLATION (SEG0)
  VA  5: 0x00000008 (decimal:    8) --> VALID in SEG0: 0x00000008 (decimal:    8)
  VA  6: 0x0000000d (decimal:   13) --> VALID in SEG1: 0x0000000d (decimal:   13)
  VA  7: 0x0000000f (decimal:   15) --> VALID in SEG1: 0x0000000f (decimal:   15)
  VA  8: 0x00000001 (decimal:    1) --> VALID in SEG0: 0x00000001 (decimal:    1)
  VA  9: 0x00000000 (decimal:    0) --> VALID in SEG0: 0x00000000 (decimal:    0)
```

5) To obtain no valid virtual addresses we can set the parameters accordingly: base0 = 0, limit0=0, base1 = 15 and limit1 = 0. Thus there are no possible valid virtual addresses. Also, we can observe from the following screenshot.

```
  VA  9: 0x00000000 (decimal:    0) --> VALID in SEG0: 0x00000000 (decimal:    0)

(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/Lab 7 Virtual M
emory$ ./segmentation.py -a 20 -p 100 --b0 0 --l0 0 --b1 15 --l1 0 -c
ARG seed 0
ARG address space size 20
ARG phys mem size 100

Segment register information:

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

  Segment 1 base  (grows negative) : 0x0000000f (decimal 15)
  Segment 1 limit                  : 0

Virtual Address Trace
  VA  0: 0x00000010 (decimal:   16) --> SEGMENTATION VIOLATION (SEG1)
  VA  1: 0x0000000f (decimal:   15) --> SEGMENTATION VIOLATION (SEG1)
  VA  2: 0x00000008 (decimal:    8) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x00000005 (decimal:    5) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x0000000a (decimal:   10) --> SEGMENTATION VIOLATION (SEG1)

(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/Lab 7 Virtual M
emory$
```

Part 3

1) The linear page table can be varied based on the following parameters:

a) Number of bits in the address space

Suppose the number of bits in the address space is 16.

```
(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/La
b 7 Virtual Memory$ ./paging-linear-size.py -v 16
ARG bits in virtual address 16
ARG page size 4k
ARG pte size 4

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

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

Page size = 4k = 4096 bytes = 2^{12}

Number of bits needed in the offset = 12

Number of bits for the VPN = 4

Size of each page table entry = 4 bytes

Number of entries in the table = $2^{(\text{number of VPN bits})} = 2^4 = 16$

Size of page table = (size of each page table entry) × (Number of entries in the table)

= 4 × 16

= 64 bytes

b) <u>Page size</u>

Suppose the page size is 32.

```
(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/La
b 7 Virtual Memory$ ./paging-linear-size.py -p 32
ARG bits in virtual address 32
ARG page size 32
ARG pte size 4

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

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

Page size = 32 bytes = 2^5

Number of bits needed in the offset = 5

Number of bits for the VPN = 32 - 5 = 27

Size of each page table entry = 4 bytes

Number of entries in the table = $2^{(\text{number of VPN bits})} = 2^{27}$

Size of page table = (size of each page table entry) × (Number of entries in the table)

= 4×2^{27} bytes

c) <u>Page table entry size</u>

Suppose the page table entry size is 16 bytes.

```
(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/La
b 7 Virtual Memory$ ./paging-linear-size.py -e 16
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 16

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

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

Page size = 4096 bytes = 2^{12}

Number of bits needed in the offset = 12

Number of bits for the VPN = 32 - 12 = 20

Size of each page table entry = 16 bytes

Number of entries in the table = $2^{(\text{number of VPN bits})}$ = 220

Size of page table = (size of each page table entry) × (Number of entries in the table)

$= 16 \times 2^{20}$ bytes

Part 4

1) From the flags:

   -P 1k -a 1m -p 512m -v -n 0

   -P 1k -a 2m -p 512m -v -n 0

   -P 1k -a 4m -p 512m -v -n 0

   we can observe that the address space is doubling with each flag. From the formulae:

   Size of page table = (size of each page table entry) × (Number of entries in the table)

   Number of entries in the table = (address space size) / (page size)

   we can deduce that size of page table is directly proportional to address space size. Thus we can say that the page table size also doubles with each flag.

   From the flags:

   -P 1k -a 1m -p 512m -v -n 0

   -P 2k -a 1m -p 512m -v -n 0

   -P 4k -a 1m -p 512m -v -n 0

   we can observe that size of page doubles with every flag. Also, from the above stated formulae, we can deduce that size of page table is inversely proportional to size of page. Thus we can say that size of page tables halves with every flag.

   Larger page size will lead to less memory utilization as only one process takes lots of memory which goes unused. This unused space can be utilized by other process if the size of page is decreased.

2) For the first command:

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

```
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)

(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/La
```

We can observe that all the virtual addresses generated are invalid.

For the second command:

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

```
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)

(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/La
```

We can observe that one out of five virtual addresses generated is valid. Thus the percentage of valid addresses is roughly 20%.

For the third command:

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

```
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]

(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/L
```

we can observe that there are 3 valid addresses out 5. Thus the percentage of valid addresses is roughly 60% now.

For the fourth command:

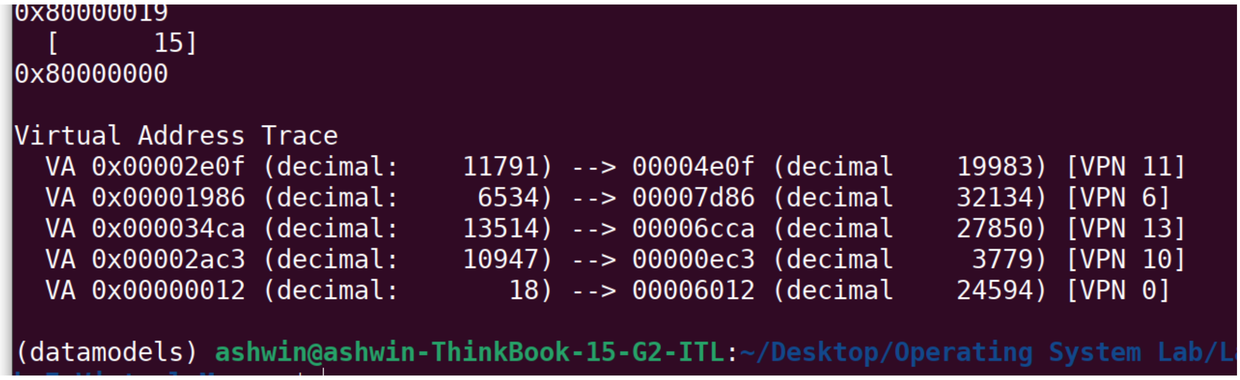
./paging-linear-translate.py -P 1k -a 16k -p 32k -v -c -u 75

```
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]

(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/L
```

we can observe that all the virtual addresses generated are valid. Thus the percentage of valid addresses is roughly 100%.

For the last command:

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

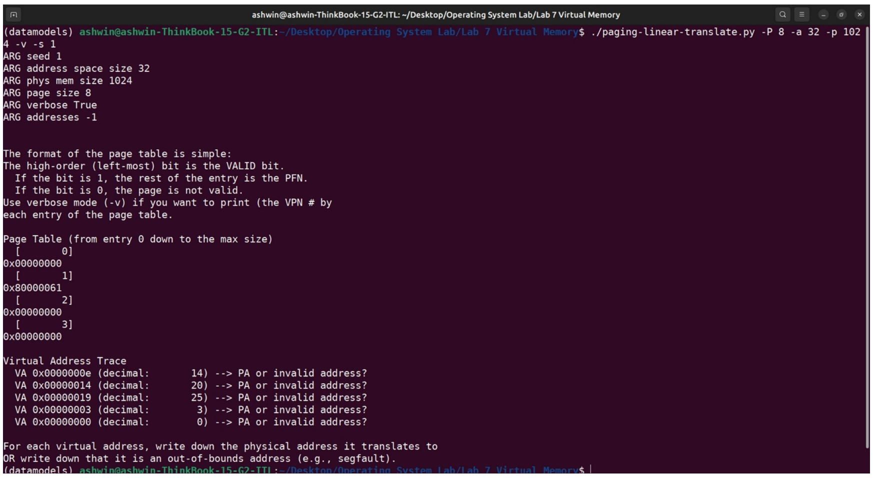


```
0x80000019
  [       15]
0x80000000

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]

(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/L
```

we can observe that all the virtual addresses generated are valid. Thus the percentage of valid addresses is roughly 100%. From this we can conclude that as we increase the percentage of pages that are allocated in each address space, the percentage of valid virtual addresses also increases.

3) For the command:

./paging-linear-translate.py -P 8 -a 32 -p 1024 -v -s 1



```
ashwin@ashwin-ThinkBook-15-G2-ITL: ~/Desktop/Operating System Lab/Lab 7 Virtual Memory
(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/Lab 7 Virtual Memory$ ./paging-linear-translate.py -P 8 -a 32 -p 102
4 -v -s 1
ARG seed 1
ARG address space size 32
ARG phys mem size 1024
ARG page size 8
ARG verbose True
ARG addresses -1


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) --> PA or invalid address?
  VA 0x00000014 (decimal:       20) --> PA or invalid address?
  VA 0x00000019 (decimal:       25) --> PA or invalid address?
  VA 0x00000003 (decimal:        3) --> PA or invalid address?
  VA 0x00000000 (decimal:        0) --> PA or invalid address?

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).
(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/Lab 7 Virtual Memory$
```

This page table is very tiny with 4 page entries of size 8 bytes each so, this page table is unrealistically small.

For the command:

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

```
ashwin@ashwin-ThinkBook-15-G2-ITL: ~/Desktop/Operating System Lab/Lab 7 Virtual Memory
OR write down that it is an out-of-bounds address (e.g., segfault).

(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/Lab 7 Virtual Memory$ ./paging-linear-translate.py -P 1m -a 256m -p
512m -v -s 3
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
  [       5]
0x80000084
  [       6]
0x00000000
  [       7]
0x800000f0
  [       8]
```

This combination of parameters are unrealistic as the size of page is too large.

4) The program does not work when the page is bigger than address space size. Suppose we set page size as 16k and address space size as 8k then:

```
(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/L
b 7 Virtual Memory$ ./paging-linear-translate.py -P 16k -a 8k -v -c
ARG seed 0
ARG address space size 8k
ARG phys mem size 64k
ARG page size 16k
ARG verbose True
ARG addresses -1

Error in argument: address space must be a multiple of the pagesize
(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/L
```

as we can see, there is an error.

The program also does not work when the physical memory size is less or equal to the address space size.

```
(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/L
b 7 Virtual Memory$ ./paging-linear-translate.py -p 4k
ARG seed 0
ARG address space size 16k
ARG phys mem size 4k
ARG page size 4k
ARG verbose False
ARG addresses -1

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

And finally, for the program to work, the address space must be a power of 2.

```
(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/La
b 7 Virtual Memory$ ./paging-linear-translate.py -a 9k
ARG seed 0
ARG address space size 9k
ARG phys mem size 64k
ARG page size 4k
ARG verbose False
ARG addresses -1

Error in argument: address space must be a multiple of the pagesize
(datamodels) ashwin@ashwin-ThinkBook-15-G2-ITL:~/Desktop/Operating System Lab/La
b 7 Virtual Memory$
```