
CS 314 – Operating Systems Lab

Lab7 Report

Name: Sahaja Nandyala

Roll-No: 200010032

1 relocation.py

1.1

If Virtual Address(VA)

- In bound : $VA < Limit$ and $PA = Base + VA$
- Out of bound : $VA > Limit$

1.1.1 For seed = 1

```
C:\Users\nandy\Desktop\IIIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 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.
```

Physical address calculation using Base Bound

VA 0 : 782 > 290 (Out of bound)

VA 1 : 261 < 290 (In bound) → Physical address: 13884 + 261 = 14145

VA 2 : 507 > 290 (Out of bound)

VA 3 : 460 > 290 (Out of bound)

VA 4 : 667 > 290 (Out of bound)

1.1.2 For seed = 2

Physical address calculation using Base Bound

VA 0 : 57 < 500 (In bound) → Physical address: 15529 + 57 = 15586

VA 1 : 86 < 500 (In bound) → Physical address: 15529 + 86 = 15615

VA 2 : 855 > 500 (Out of bound)

VA 3 : 753 > 500 (Out of bound)

VA 4 : 685 > 500 (Out of bound)

```

C:\Users\nandy\Desktop\IIIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 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 : 580

Virtual Address Trace
VA 0: 0x00000039 (decimal: 57) --> PA or segmentation violation?
VA 1: 0x00000056 (decimal: 86) --> PA or segmentation violation?
VA 2: 0x000000357 (decimal: 855) --> PA or segmentation violation?
VA 3: 0x0000002f1 (decimal: 753) --> PA or segmentation violation?
VA 4: 0x0000002ad (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.

```

1.1.3 For seed = 3

```

C:\Users\nandy\Desktop\IIIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 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.

```

Physical address calculation using Base Bound

VA 0 : 378 > 316 (Out of bound)

VA 1 : 618 > 316 (Out of bound)

VA 2 : 648 > 316 (Out of bound)

VA 3 : 67 < 316 (In bound) → Physical address: $8916 + 67 = 8983$

VA 4 : 13 < 316 (In bound) → Physical address: $8916 + 13 = 8929$

1.2

physical memory size (p) = 16kB

number of virtual addresses (n) = 10

seed (s) = 0

limit (l) = 930

From the following figure we can see that among all the maximum virtual address space value is 929 so, for limit 930 all virtual address will be valid for given parameters.

```

C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 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)

```

1.3

physical memory size (p) = 16kB = $(16 \times 1024) = 16384$

number of virtual addresses (n) = 10

limit (l) = 100

$b + 100 = 16384$

$b = 16384 - 100$

From the following screenshots we can see that for base = 16284 the address space fits in memory where as for base = 16285 address space doesn't fit in physical memory.

```

C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 relocation.py -s 1 -n 10 -l 100 -b 16284

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) --> PA or segmentation violation?
VA 1: 0x00000363 (decimal: 867) --> PA or segmentation violation?
VA 2: 0x0000030e (decimal: 782) --> PA or segmentation violation?
VA 3: 0x00000105 (decimal: 261) --> PA or segmentation violation?
VA 4: 0x000001fb (decimal: 507) --> PA or segmentation violation?
VA 5: 0x000001cc (decimal: 460) --> PA or segmentation violation?
VA 6: 0x0000029b (decimal: 667) --> PA or segmentation violation?
VA 7: 0x00000327 (decimal: 807) --> PA or segmentation violation?
VA 8: 0x00000060 (decimal: 96) --> PA or segmentation violation?
VA 9: 0x000001d (decimal: 29) --> 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.

```

Figure 1: base = 16284

```

C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 relocation.py -s 1 -n 10 -l 100 -b 16285

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

Base-and-Bounds register information:

  Base   : 0x00003f9d (decimal 16285)
  Limit  : 100

Error: address space does not fit into physical memory with those base/bounds values.
Base + Limit: 16385   Psize: 16384

```

Figure 2: base = 16285

1.4

Running above problem i.e., with parameters $s = 0$, $n = 10$, $l = 100$ with big address space size and physical memory size values.

- For address space size = 32m and physical memory size = 1g

```
C:\Users\nandy\Desktop\IIIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 relocation.py -a 32m -p 1g -n 10 -s 1 -l 100 -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: 0x01b1c2d5 (decimal: 28435157) --> SEGMENTATION VIOLATION
VA 1: 0x01970d77 (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
```

- For address space size = 1g and physical memory size = 10g

```
C:\Users\nandy\Desktop\IIIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 relocation.py -a 1g -p 10g -n 10 -s 1 -l 100 -c

ARG seed 1
ARG address space size 1g
ARG phys mem size 10g

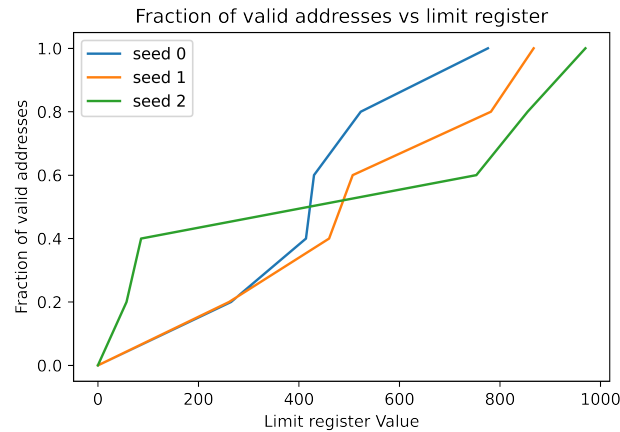
Base-and-Bounds register information:

Base : 0x55fe3cdd (decimal 1442725085)
Limit : 100

Virtual Address Trace
VA 0: 0x363c5ab6 (decimal: 909925046) --> SEGMENTATION VIOLATION
VA 1: 0x30e1aef0 (decimal: 820896752) --> SEGMENTATION VIOLATION
VA 2: 0x10530d08 (decimal: 273878208) --> SEGMENTATION VIOLATION
VA 3: 0x1fb5355e (decimal: 531969374) --> SEGMENTATION VIOLATION
VA 4: 0x1cc4762b (decimal: 482637355) --> SEGMENTATION VIOLATION
VA 5: 0x29b3b303 (decimal: 699642627) --> SEGMENTATION VIOLATION
VA 6: 0x327a7181 (decimal: 846885240) --> SEGMENTATION VIOLATION
VA 7: 0x0601cba3 (decimal: 100780963) --> SEGMENTATION VIOLATION
VA 8: 0x01d071ef (decimal: 30437871) --> SEGMENTATION VIOLATION
VA 9: 0x357d2ceb (decimal: 897395947) --> SEGMENTATION VIOLATION
```

1.5

Following graph is between fraction of randomly generated virtual addresses running with different seeds that are valid Vs limit values.



2 segmentation.py

2.1

address space size = 128

physical memory size = 512

If $vaddr < asize/2$ Then virtual address falls under segment 0,

- In bound : $vaddr < limit0$ and $paddr = vaddr + base0$
- Out of bound : $vaddr \geq limit0$

If $vaddr \geq asize/2$ Then virtual address falls under segment 1,

- In bound : $asize - limit1 \leq vaddr$ and $paddr = base1 - (asize - vaddr)$
- Out of bound : $asize - limit1 > vaddr$

$$asize/2 = 128/2 = 64$$

$$asize - limit1 = 128 - 20 = 108$$

From the above formulas

- For seed = 0

```
C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 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?
```

VA 0: $108 > 64 \rightarrow SEG1$ and $108 \geq 108 \rightarrow In\ bound$ and $paddr = 512 - (128 - 108) = 492$

VA 1: $97 > 64 \rightarrow SEG1$ and $97 < 108 \rightarrow Out\ of\ Bound$

VA 2: $53 < 64 \rightarrow SEG0$ and $53 > 20 \rightarrow Out\ of\ Bound$

VA 3: $33 < 64 \rightarrow SEG0$ and $33 > 20 \rightarrow Out\ of\ Bound$

VA 4: $65 > 64 \rightarrow SEG1$ and $65 < 108 \rightarrow Out\ of\ Bound$

- For seed = 1

```
C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 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?
```

VA 0 : $17 < 64 \rightarrow SEG0$ and $17 < 20 \rightarrow In\ bound$ and $paddr = 0 + 17 = 17$

VA 1 : $108 > 64 \rightarrow SEG1$ and $108 \geq 108 \rightarrow In\ bound$ and $paddr = 512 - (128 - 108) = 492$

VA 2 : $97 > 64 \rightarrow SEG1$ and $97 < 108 \rightarrow Out\ of\ Bound$

VA 3 : $32 < 64 \rightarrow SEG0$ and $32 > 20 \rightarrow Out\ of\ Bound$

VA 4 : $63 < 64 \rightarrow SEG0$ and $63 > 20 \rightarrow Out\ of\ Bound$

- For seed = 2

```
C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 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?
```

VA 0 : $122 > 64 \rightarrow SEG1$ and $122 > 108 \rightarrow In\ bound$ and $paddr = 512 - (128 - 122) = 506$

VA 1 : $121 > 64 \rightarrow SEG1$ and $121 > 108 \rightarrow In\ bound$ and $paddr = 512 - (128 - 121) = 505$

VA 2 : $7 < 64 \rightarrow SEG0$ and $7 < 20 \rightarrow In\ bound$ and $paddr = 0 + 7 = 7$

VA 3 : $10 < 64 \rightarrow SEG0$ and $10 < 20 \rightarrow In\ bound$ and $paddr = 0 + 10 = 10$

VA 4 : $106 > 64 \rightarrow SEG1$ and $106 < 108 \rightarrow Out\ of\ bound$

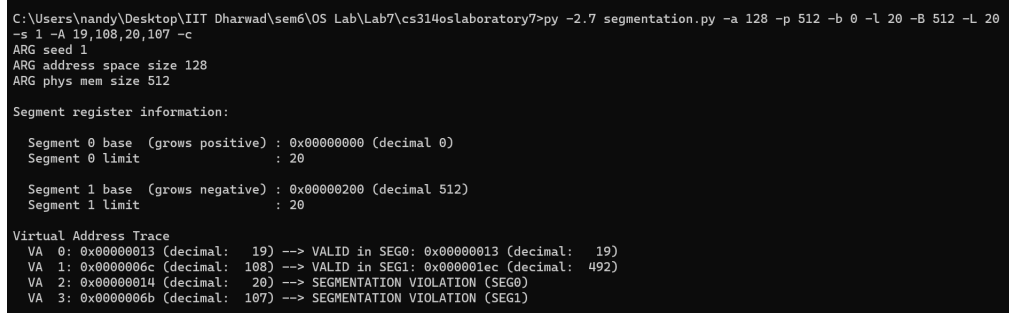
2.2

- 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 107

To verify whether these values are correct or not we have to execute segmentation.py using following command

Command :

```
py -2.7 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1  
-A 19,108,20,107 -c
```



```
C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20  
-s 1 -A 19,108,20,107 -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: 0x00000013 (decimal: 19) --> VALID in SEG0: 0x00000013 (decimal: 19)  
VA 1: 0x0000001c (decimal: 28) --> VALID in SEG1: 0x000001ec (decimal: 492)  
VA 2: 0x00000014 (decimal: 20) --> SEGMENTATION VIOLATION (SEG0)  
VA 3: 0x0000006b (decimal: 107) --> SEGMENTATION VIOLATION (SEG1)
```

2.3

Given Virtual addresses 0,1 should be valid and 2 to 13 should be invalid and 14,15 should be valid.

The first segment consists of 0,1 hence $base0 = 0$ and 2 consecutive addresses are valid so $limit0 = 2$

The second segment consists of 14, 15 hence $base1 = 15$ and 2 consecutive addresses are valid so $limit1 = 2$

Following is the screenshot of output of segmentation.py with $base0 = 0$ $limit0 = 0$ and $base1 = 15$ $limit1 = 2$

```

C:\Users\Nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 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)

```

2.4

To generate a problem where roughly 90% of the virtual addresses are valid then valid memory size should be 90% of the actual virtual memory size. so, the parameter that we should concentrate on are size of the virtual memory, base and limit values of two segments i.e., base0, limit0, base1, limit1

Let us take tiny address space of size 16 bytes in physical memory of size 128 bytes. Then our size of segment0 + size of segment 1 = 90% of address space (16 bytes) = $16 \times 0.9 = 14.4$

so, total memory by both segment 0 and segment 1 should be 14 i.e.,

limit0 + limit1 = 14

one of the possible answers is

base0 = 0

limit0 = 8

base1 = 15

limit1 = 6

Following is the screenshot with the respective parameters we can see that 90% of the virtual addresses are valid

```

C:\Users\Nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 segmentation.py -a 16 -p 128 --b0 0 --l0 8 --b1 15 --l1 6 -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 : 8
Segment 1 base (grows negative) : 0x0000000f (decimal 15)
Segment 1 limit : 6

Virtual Address Trace
VA 0: 0x00000000 (decimal: 0) --> VALID in SEG1: 0x0000000e (decimal: 12)
VA 1: 0x00000001 (decimal: 1) --> VALID in SEG1: 0x0000000b (decimal: 11)
VA 2: 0x00000002 (decimal: 2) --> VALID in SEG0: 0x00000006 (decimal: 6)
VA 3: 0x00000003 (decimal: 3) --> VALID in SEG0: 0x00000004 (decimal: 4)
VA 4: 0x00000004 (decimal: 4) --> SEGMENTATION VIOLATION (SEG1)

```


2.5

To get no virtual spaces valid then size of segment0 and size of segment1 should be 0.
One of the possible answers is

base0 = 0

limit0 = 0

base1 = 15

limit1 = 0

Following is the screenshot with the respective parameters we can see that all virtual address are invalid

```
C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 segmentation.py -a 16 -p 128 --b0 0 --l0 0 --b1 15 --l1 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) : 0x0000000f (decimal 15)
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)
```

3 paging-linear-size.py

The program paging-linear-size.py lets us figure out the size of a linear page table given a variety of input parameters. Computing how big a linear page table is with the following variations in characteristics

3.1 Varying number of bits in the address space

- number of bits = 16

Following is the screen shot after executing paging-linear-size.py with parameter -v 16

Number of bits in the virtual address = 16

```
C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 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

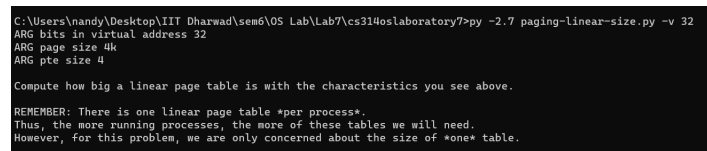
Number of entries in the table = $2^{(\text{number of VPN bits})} = 2^4 = 16.0$

Size of page table = (size of each page table entry) \times (Number of entries in the table)
 $= 4 \times 16 = 64 \text{ bytes} = 0.0625 \text{ KB}$

- **number of bits = 32**

Following is the screen shot after executing paging-linear-size.py with parameter -v 32

Number of bits in the virtual address = 32



```
C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 paging-linear-size.py -v 32
ARG bits in virtual address 32
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 = 32 - 12 = 20

Size of each page table entry = 4 bytes

Number of entries in the table = $2^{(\text{number of VPN bits})} = 2^{20}$

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

$= 4 \times 2^{20} = 4096 \text{ KB}$

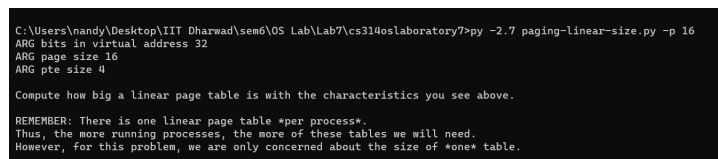
From above calculations we can say that size of page table increases with increase in number of bits in virtual address space.

3.2 Varying page size

- **page size = 16**

Following is the screen shot after executing paging-linear-size.py with parameter -p 16

Number of bits in the virtual address = 32



```
C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 paging-linear-size.py -p 16
ARG bits in virtual address 32
ARG page size 16
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 = 16 bytes = 2^4

Number of bits needed in the offset = 4

Number of bits for the VPN = 32 - 4 = 28

Size of each page table entry = 4 bytes

Number of entries in the table = $2^{(\text{number of VPN bits})} = 2^{28}$

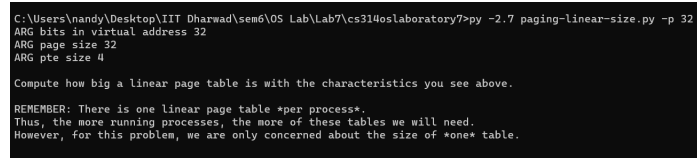
Size of page table = (size of each page table entry) \times (Number of entries in the table)

$$\begin{aligned} & \text{table}) \\ & = 4 \times 2^{28} = 1048576KB \end{aligned}$$

- **page size = 32**

Following is the screen shot after executing paging-linear-size.py with parameter -p 32

Number of bits in the virtual address = 32



```
C:\Users\nandy\Desktop\IIIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 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.
```

$$\begin{aligned} \text{Page size} &= 32 \text{ bytes} = 2^5 \\ \text{Number of bits needed in the offset} &= 5 \\ \text{Number of bits for the VPN} &= 32 - 5 = 27 \\ \text{Size of each page table entry} &= 4 \text{ bytes} \\ \text{Number of entries in the table} &= 2^{(\text{number of VPN bits})} = 2^{27} \\ \text{Size of page table} &= (\text{size of each page table entry}) \times (\text{Number of entries in the table}) \\ &= 4 \times 2^{27} = 524288KB \end{aligned}$$

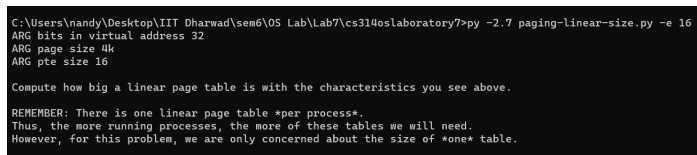
From above calculations we can see that the size of page table decreases with increase in page size.

3.3 Varying page table entry size

- **page table entry size = 16**

Following is the screen shot after executing paging-linear-size.py with parameter -e 16

Number of bits in the virtual address = 32



```
C:\Users\nandy\Desktop\IIIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 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.
```

$$\begin{aligned} \text{Page size} &= 4096 \text{ bytes} = 2^{12} \\ \text{Number of bits needed in the offset} &= 12 \\ \text{Number of bits for the VPN} &= 32 - 12 = 20 \\ \text{Size of each page table entry} &= 16 \text{ bytes} \\ \text{Number of entries in the table} &= 2^{(\text{number of VPN bits})} = 2^{20} \\ \text{Size of page table} &= (\text{size of each page table entry}) \times (\text{Number of entries in the table}) \end{aligned}$$

$$= 16 \times 2^{20} = 16384 \text{ KB}$$

- **page table entry size = 32**

Following is the screen shot after executing paging-linear-size.py with parameter -e 32

Number of bits in the virtual address = 32

```
C:\Users\Nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 paging-linear-size.py -e 32
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 32

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.
```

$$\text{Page size} = 4096 \text{ bytes} = 2^{12}$$

$$\text{Number of bits needed in the offset} = 12$$

$$\text{Number of bits for the VPN} = 32 - 12 = 20$$

$$\text{Size of each page table entry} = 32 \text{ bytes}$$

$$\text{Number of entries in the table} = 2^{(\text{number of VPN bits})} = 2^{20}$$

$$\text{Size of page table} = (\text{size of each page table entry}) \times (\text{Number of entries in the table})$$

$$= 32 \times 2^{20} = 32768 \text{ KB}$$

From the above calculations we can see that the size of page table increases with increase in size of page table entry.

4 paging-linear-translate.py

4.1

$$\text{Size of page table} = (\text{size of each page table entry}) \times (\text{Number of entries in the table})$$

$$\text{Number of entries in the table} = (\text{address space size}) / (\text{page size})$$

As the size of page table is directly proportional to number of entries which is directly proportional to size of address space and indirectly proportional to page size so,

- From the given flags we can see that the size of address space increase by factor of 2 so, the Page table size increases by factor of 2.

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
```

- From the given flags we can see that the size of page increases by factor of 2 so, the Page table size decrease by factor of 2.

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
```

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

4.2

As we increase the percentage of pages that are allocated in each address space are increased the number of valid address will increase.

address space size = $16k = 2^{14}$

physical memory size = $32k$

page size = $1k = 2^{10}$

Number of bits in virtual address = 14 Number of bits in offset = 10

Number of bits in VPN = 4

Hence first 4 bits in binary representation of virtual address represents VPN.

- $u = 0$

```
C:\Users\andy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 0
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) \rightarrow 1110 VPN = 14 as the left most digit is 0 this is not valid

VA 0x00003ee5 (decimal: 16101) \rightarrow 1111 VPN = 15 as the left most digit is 0 this is not valid

VA 0x000033da (decimal: 13274) \rightarrow 1100 VPN = 12 as the left most digit is 0 this is not valid

VA 0x000039bd (decimal: 14781) \rightarrow 1110 VPN = 14 as the left most digit is 0 this is not valid

VA 0x000013d9 (decimal: 5081) \rightarrow 0100 VPN = 4 as the left most digit is 0 this is not valid

- **u = 25**

```
C:\Users\randy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 25
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] 0x00000018
[ 1] 0x00000000
[ 2] 0x00000000
[ 3] 0x00000000
[ 4] 0x00000000
[ 5] 0x00000009
[ 6] 0x00000000
[ 7] 0x00000000
[ 8] 0x00000010
[ 9] 0x00000000
[10] 0x00000013
[11] 0x00000000
[12] 0x0000001f
[13] 0x0000001c
[14] 0x00000000
[15] 0x00000000
```

Virtual Address Trace

VA 0x00003986 (decimal: 14726) \rightarrow 1110 VPN = 14 as the left most bit is 0 is not valid

VA 0x00002bc6 (decimal: 11206) \rightarrow 1010 VPN = 10 , PFN = 20422

VA 0x00001e37 (decimal: 7735) \rightarrow 0111 VPN = 7 as the left most bit is 0 is not valid

VA 0x00000671 (decimal: 1649) \rightarrow 0001 VPN = 1 as the left most bit is 0 is not valid

VA 0x00001bc9 (decimal: 7113) \rightarrow 0110 VPN = 6 as the left most bit is 0 is not valid

- **u = 50**

Virtual Address Trace

VA 0x00003385 (decimal: 13189) \rightarrow 1100 VPN = 12, left-most bit = 1 so, valid PFN = 16261

VA 0x0000231d (decimal: 8989) \rightarrow 1000 VPN = 8, left-most bit = 0 so, invalid

VA 0x000000e6 (decimal: 230) \rightarrow 0000 VPN = 0, left-most bit = 1 so, valid PFN = 24806

```

C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 50
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] 0x80000000
[ 2] 0x80000000
[ 3] 0x8000000c
[ 4] 0x80000009
[ 5] 0x00000000
[ 6] 0x80000014
[ 7] 0x80000013
[ 8] 0x00000000
[ 9] 0x8000001f
[10] 0x8000001c
[11] 0x00000000
[12] 0x8000000f
[13] 0x00000000
[14] 0x00000000
[15] 0x80000008

```

VA 0x00002e0f (decimal: 11791) → 1011 VPN = 11, left-most bit = 0 so, invalid
VA 0x00001986 (decimal: 6534) → 0110 VPN = 6, left-most bit = 1 so, valid PFN
= 30086

- **u = 75**
Virtual Address Trace

```

C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 75
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

```

VA 0x00002e0f (decimal: 11791) → VPN = 11 left-most bit = 1 so, valid PFN = 19983
VA 0x00001986 (decimal: 6534) → VPN = 6 left-most bit = 1 so, valid PFN = 32134
VA 0x000034ca (decimal: 13514) → VPN = 13 left-most bit = 1 so, valid PFN = 27850
VA 0x00002ac3 (decimal: 10947) → VPN = 10 left-most bit = 1 so, valid PFN = 10947
VA 0x00000012 (decimal: 18) → VPN = 0 left-most bit = 1 so, valid PFN = 18

- **u = 100**
Virtual Address Trace

```

C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 100
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] 0x00000018
[ 1] 0x00000008
[ 2] 0x0000000c
[ 3] 0x00000009
[ 4] 0x00000012
[ 5] 0x00000010
[ 6] 0x0000001f
[ 7] 0x0000001c
[ 8] 0x00000017
[ 9] 0x00000015
[10] 0x00000003
[11] 0x00000013
[12] 0x0000001e
[13] 0x0000001b
[14] 0x00000019
[15] 0x00000000

```

VA 0x00002e0f (decimal: 11791) → VPN = 11 left-most bit = 1 so, valid PFN = 19983

VA 0x00001986 (decimal: 6534) → VPN = 6 left-most bit = 1 so, valid PFN = 32134

VA 0x000034ca (decimal: 13514) → VPN = 13 left-most bit = 1 so, valid PFN = 27850

VA 0x00002ac3 (decimal: 10947) → VPN = 10 left-most bit = 1 so, valid PFN = 10947

VA 0x00000012 (decimal: 18) → VPN = 0 left-most bit = 1 so, valid PFN = 18

4.3

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

Following is the screenshot of output after executing paging-linear-translate.py with given flags.

```

C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 paging-linear-translate.py -P 8 -a 32 -p 1024 -v -s 1
ARG seed 1
ARG address space size 32
ARG phys mem size 1024
ARG page size 8
ARG verbose True
ARG addresses -1

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] 0x000000c1
[ 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?

```

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

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

Following is the screenshot of output after executing paging-linear-translate.py with given flags.

```
C:\Users\nandy\Desktop\IIIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 paging-linear-translate.py -P 8k -a 32k -p 1m -v -s 2
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) --> PA or invalid address?
VA 0x00002771 (decimal: 10097) --> PA or invalid address?
VA 0x00004d0f (decimal: 19855) --> PA or invalid address?
VA 0x00004dab (decimal: 19883) --> PA or invalid address?
VA 0x00004a64 (decimal: 19044) --> PA or invalid address?
```

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

Following is the screenshot of output after executing paging-linear-translate.py with given flags.

```
C:\Users\nandy\Desktop\IIIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 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
```

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

4.4

The program doesn't work in following cases:

- Physical memory size is less or equal to the address space size.

The default values of Physical memory size is 64k and address space is 16k. we can see from the following screenshot that program doesn't work when we take Physical memory size as 8k which is less than address space size.

```
C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 paging-linear-translate.py -p 8k
ARG seed 0
ARG address space size 16k
ARG phys mem size 8k
ARG page size 4k
ARG verbose False
ARG addresses -1
Error: physical memory size must be GREATER than address space size (for this simulation)
```

- Page size is bigger than address space size.

The default values of address space size is 16k and page size is 4k. We can see from the following screenshot that program doesn't work when we set Page size to 32k which is greater than address space size.

```
C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 paging-linear-translate.py -P 32k -v -c
ARG seed 0
ARG address space size 16k
ARG phys mem size 64k
ARG page size 32k
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)

Virtual Address Trace
Traceback (most recent call last):
  File "paging-linear-translate.py", line 174, in <module>
    if pt[vpn] < 0:
IndexError: array index out of range
```

- Page size or address space size is not power of 2. This is because it will result in discontinuous addresses.

The default values of address space size is 16k. we can see from the following screenshot that the program doesn't work when the address space size is set to 7k bytes.

```
C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 paging-linear-translate.py -a 7k -v -c
ARG seed 0
ARG address space size 7k
ARG phys mem size 64k
ARG page size 4k
ARG verbose True
ARG addresses -1
Error in argument: address space must be a power of 2
```

- when address space size is set to 0.

```
C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 paging-linear-translate.py -a 0 -v -c
ARG seed 0
ARG address space size 0
ARG phys mem size 64k
ARG page size 4k
ARG verbose True
ARG addresses -1
Error: must specify a non-zero address-space size.
```

- when page size is set to 0.

```
C:\Users\nandy\Desktop\IIT Dharwad\sem6\OS Lab\Lab7\cs314oslaboratory7>py -2.7 paging-linear-translate.py -p 0 -v -c
ARG seed 0
ARG address space size 16k
ARG phys mem size 0
ARG page size 4k
ARG verbose True
ARG addresses -1
Error: must specify a non-zero physical memory size.
```