CS314: Lab Report
Assignment 7

Devdatt N
`200010012@iitdh.ac.in`

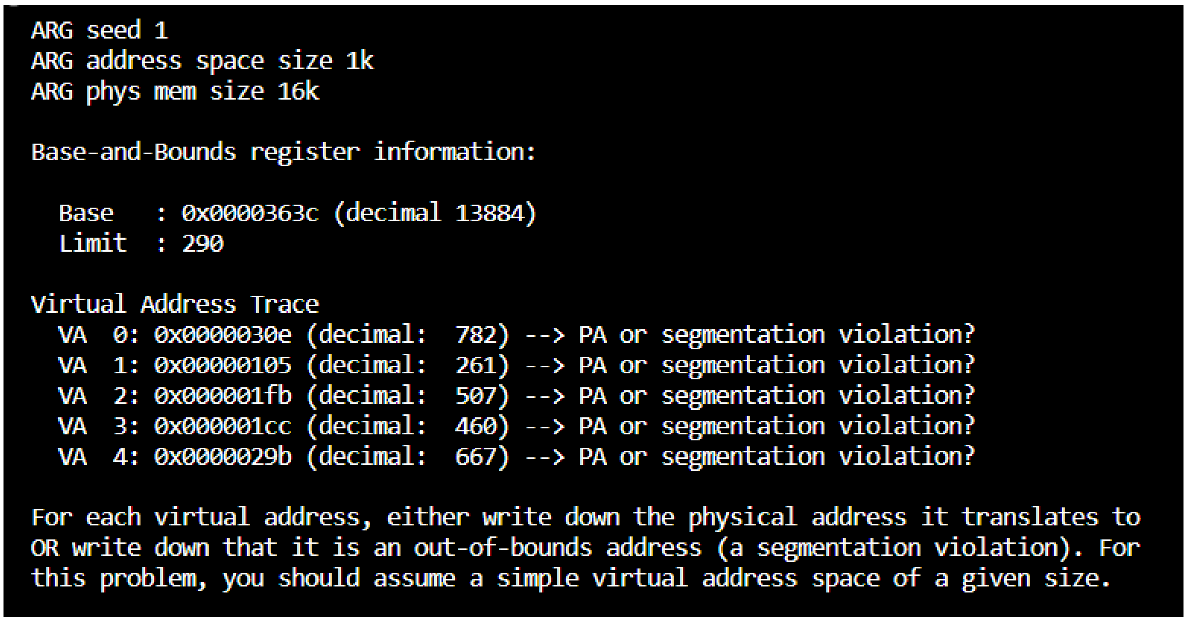
05 March 2023

1 System with Base and Bounds registers

A program, `relocation.py` has been given which allows us to see how address translation is done in a system with base and bounds registers. We will go through the questions and attempt each of them.

1.1 With different seed values

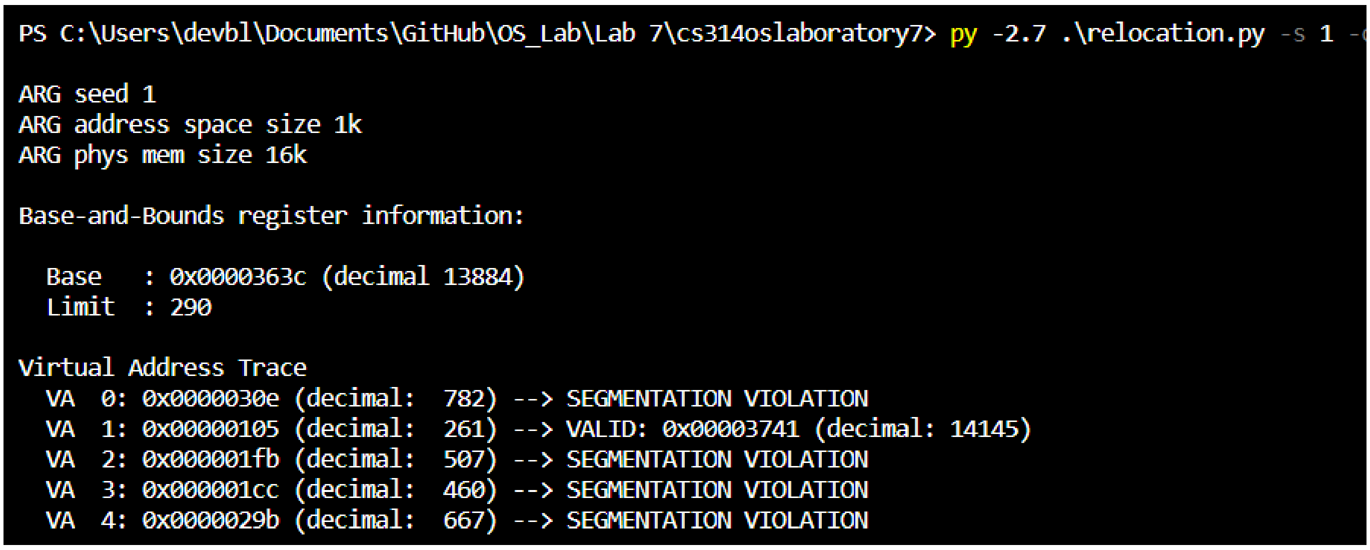
It has been asked to run the program with different seed values, namely 1,2 and 3, and compute whether each address generated is in or out of bounds. Here, it is quite easy to understand if it is



```
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.
```

Figure 1: Results for seed value 1

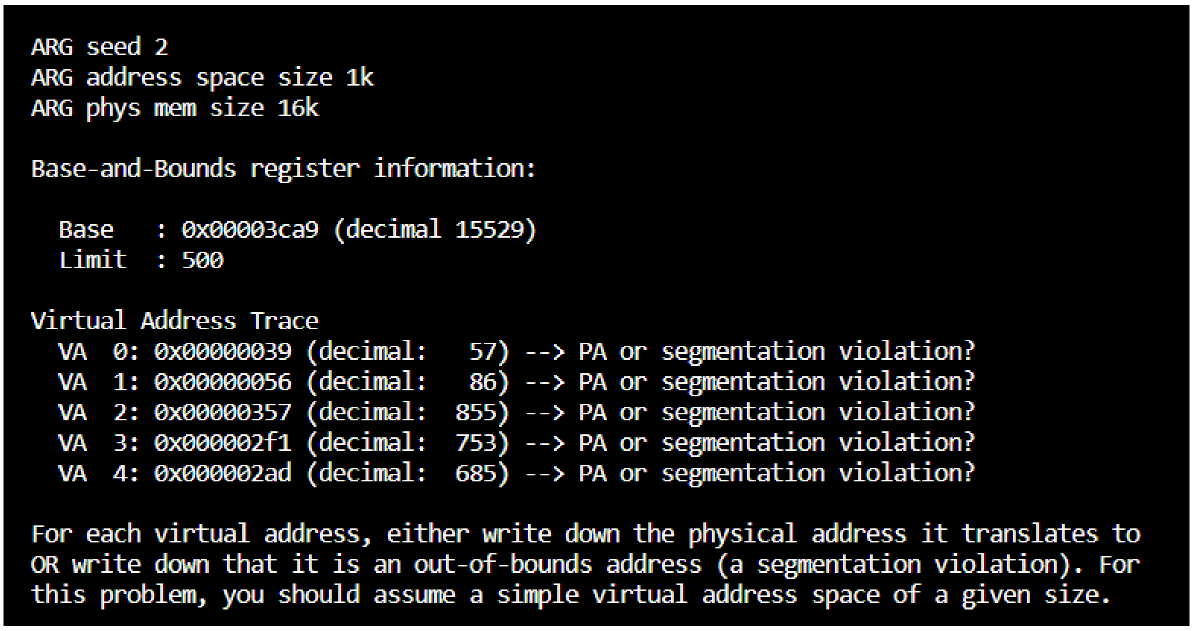
a segmentation violation or not. It has been given that the base register is 13884 in decimal. It has also been given that the limit is 290, i.e, the bound will be 290 added to the base register.
The program gives us various values of VA. Now, the sum of VA and the base register must not cross the bound, else it will be a segmentation fault. In other words, the VA in decimal must be lesser than 290. Among the 5 virtual addresses generated, only `VA1` satisfies this criterion. Therefore, only `VA1` can have a PA. The others have a segmentation violation.
For calculating its PA, we just need to add the base register and VA. We get `0x00003741` as the PA.



```
PS C:\Users\devbl\Documents\GitHub\OS_Lab\Lab 7\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) --> 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
```

Figure 2: Results for seed value 1

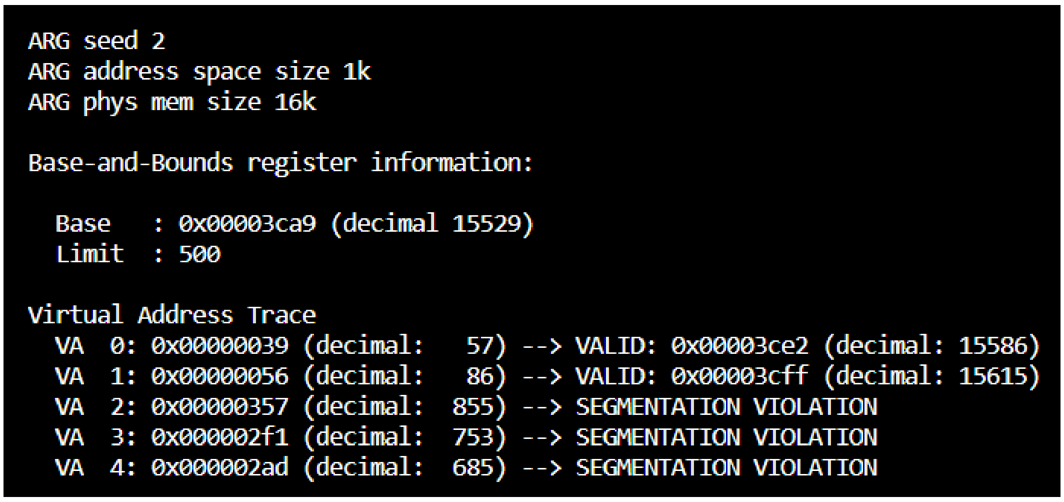
We can see that we got the correct answer. Now, for seed value 2, we run the program again. Here, we apply the same rules we applied earlier. Now the limit is 500. Therefore, VA0 and VA1



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

Base-and-Bounds register information:

  Base   : 0x00003ca9 (decimal 15529)
  Limit  : 500

Virtual Address Trace
  VA  0: 0x00000039 (decimal:   57) --> PA or segmentation violation?
  VA  1: 0x00000056 (decimal:   86) --> PA or segmentation violation?
  VA  2: 0x00000357 (decimal:  855) --> PA or segmentation violation?
  VA  3: 0x000002f1 (decimal:  753) --> PA or segmentation violation?
  VA  4: 0x000002ad (decimal:  685) --> PA or segmentation violation?

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple virtual address space of a given size.
```

Figure 3: Results for seed value 1

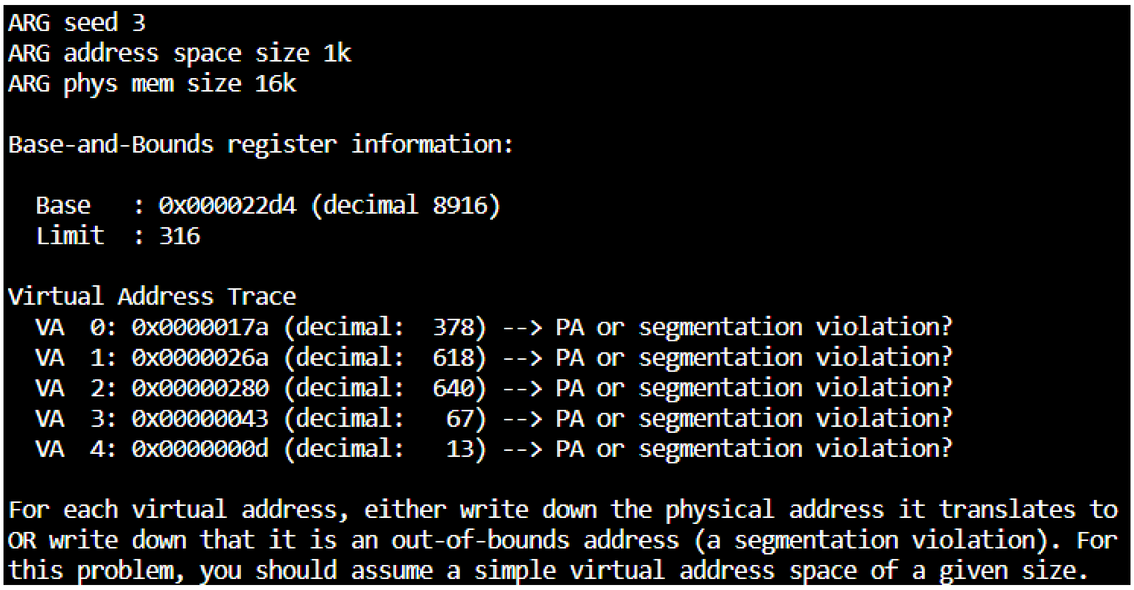
would qualify while the others would lead to a segmentation violation. On calculating the PA just like we did before, we would get `PA0 = 0x00003ca9 + 0x00000039` which gives us `0x00003ce2`. Similarly, `PA1 = 0x00003ca9 + 0x00000056` would give us `0x00003cff`.



```
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
```

Figure 4: Results for seed value 1

We can see that our answers are correct. Now, we run the program yet again with a seed value of 3 this time.



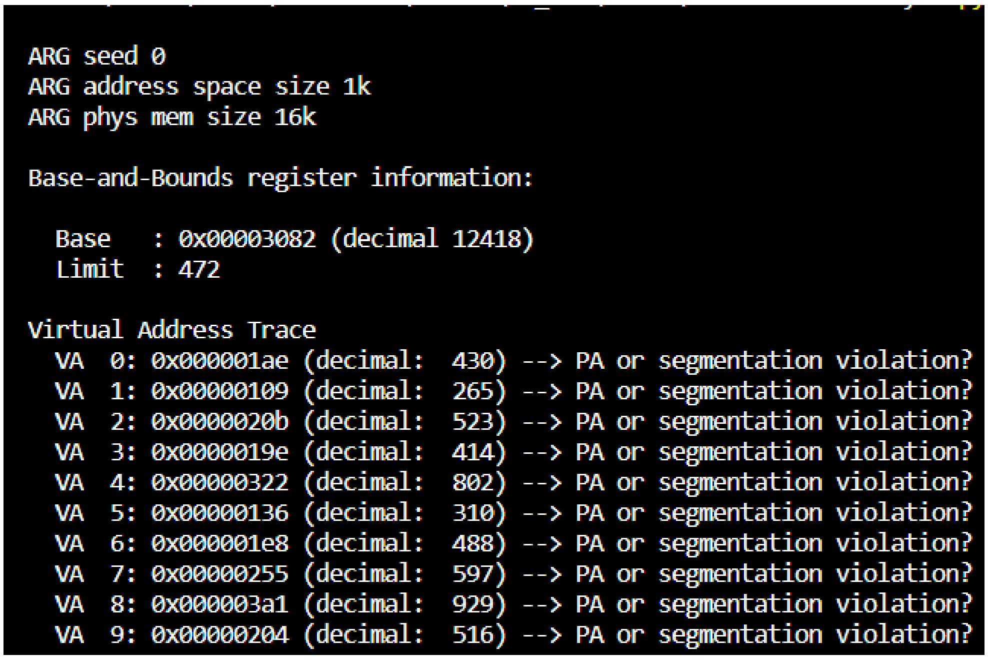
```
ARG seed 3
ARG address space size 1k
ARG phys mem size 16k

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.
```

Figure 5: Results for seed value 1

Here, we can see that the limit is 316, base register is `0x000022d4`. We can deduce from the output that VA3 and VA4 are valid, while all others are a segmentation violation. We get PA3 as `0x00002317` and PA4 as `0x000022e1` by following the same steps as before.

1.2 Finding limit for the given flags

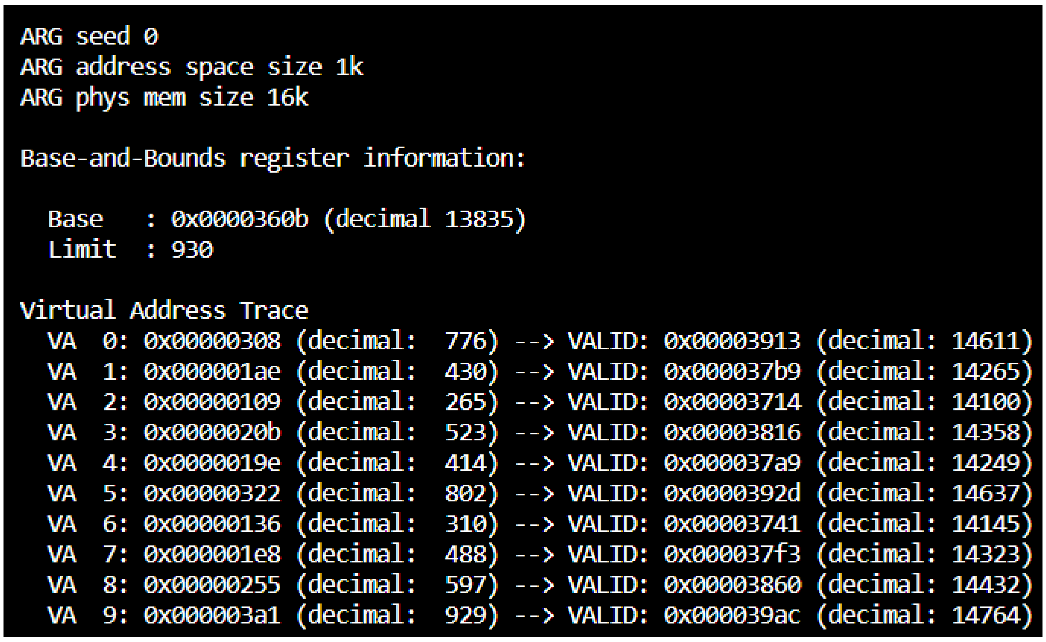
We run the program with the given flags `-s 0 -n 10`



```
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?
```

Figure 6: Results

We get the above values in the figure. We can deduce that we need at least a limit value which is the `Max value of VA + 1` which results in 930. Now, trying out this with the `-c` flag, we get:



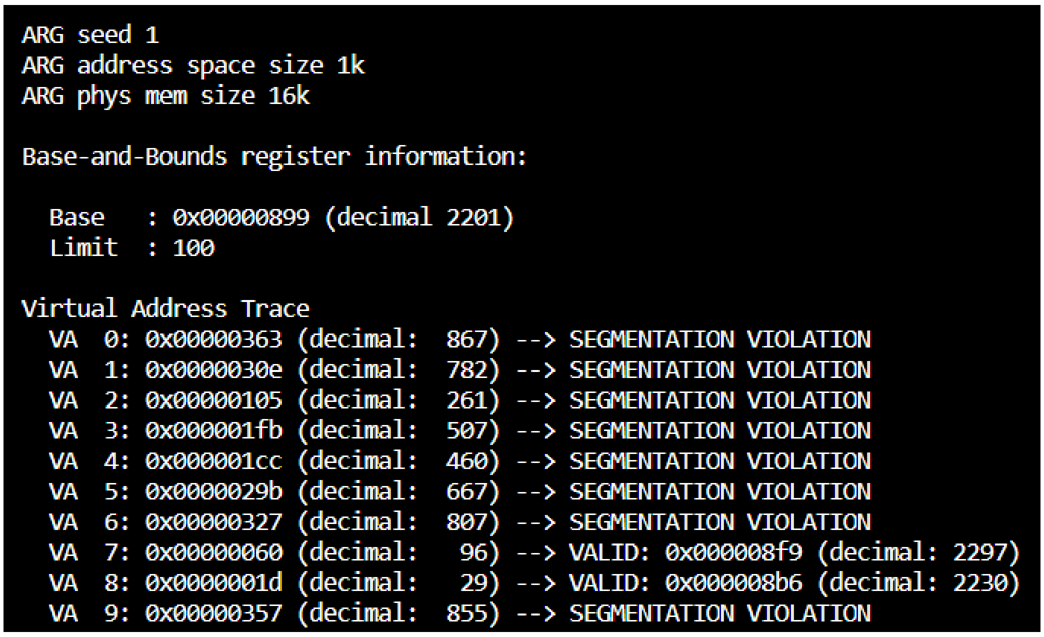
```
ARG seed 0
ARG address space size 1k
ARG phys mem size 16k

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)
```

Figure 7: Results for seed value 1

We can see that all the generated addresses are valid, and therefore our answer is correct. The bound limit to be set is 930.

1.3 Finding maximum base register value with given flags

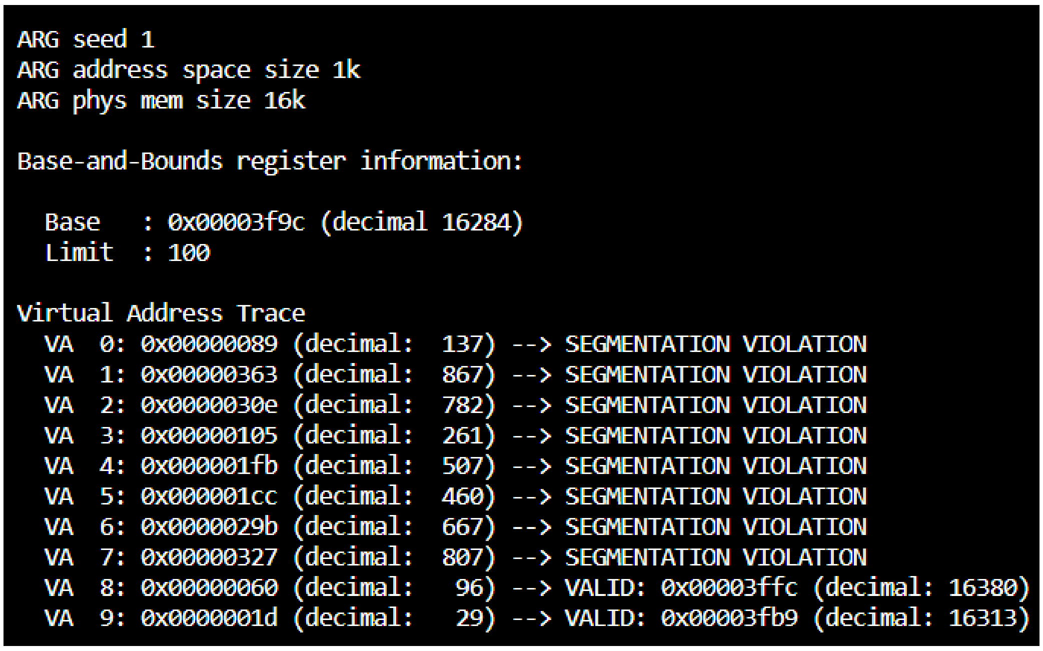
Here, on running with the given flags:



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

Base-and-Bounds register information:

  Base   : 0x00000899 (decimal 2201)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x00000363 (decimal:  867) --> SEGMENTATION VIOLATION
  VA  1: 0x0000030e (decimal:  782) --> SEGMENTATION VIOLATION
  VA  2: 0x00000105 (decimal:  261) --> SEGMENTATION VIOLATION
  VA  3: 0x000001fb (decimal:  507) --> SEGMENTATION VIOLATION
  VA  4: 0x000001cc (decimal:  460) --> SEGMENTATION VIOLATION
  VA  5: 0x0000029b (decimal:  667) --> SEGMENTATION VIOLATION
  VA  6: 0x00000327 (decimal:  807) --> SEGMENTATION VIOLATION
  VA  7: 0x00000060 (decimal:   96) --> VALID: 0x000008f9 (decimal: 2297)
  VA  8: 0x0000001d (decimal:   29) --> VALID: 0x000008b6 (decimal: 2230)
  VA  9: 0x00000357 (decimal:  855) --> SEGMENTATION VIOLATION
```

Figure 8: Results

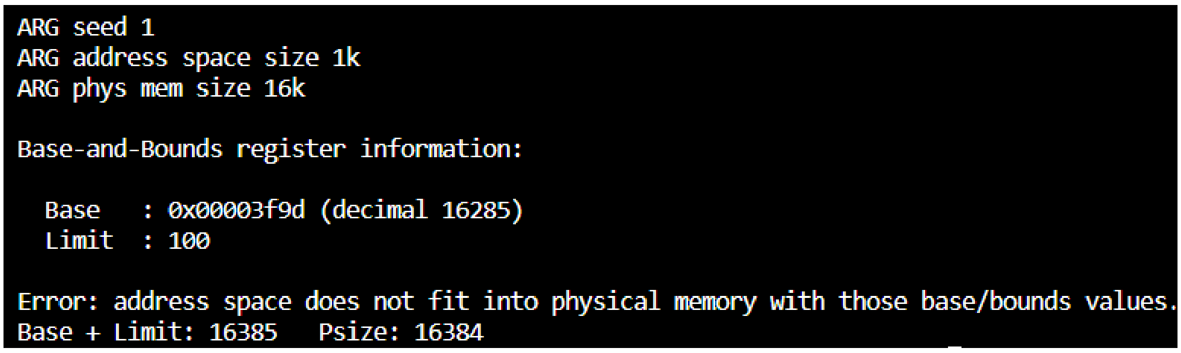
Here, sum of maximum base and limit must not increase more than the physical memory size. Therefore, we can compute max base size as `max base = 16K - 100` which gives us `max base = 16384 - 100` which is 16284. On running with 16284 as base register:



```
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)
```

Figure 9: Results

On running with one more:



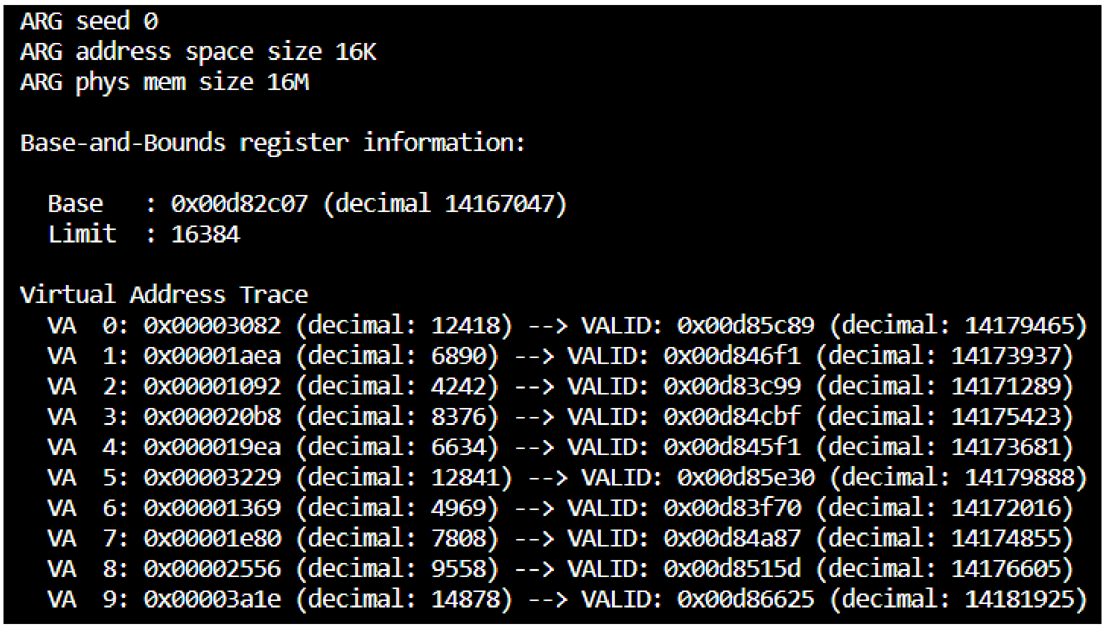
```
ARG seed 1
ARG address space size 1k
ARG phys mem size 16k

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 10: Results

We can deduce that our calculations are correct.

1.4 Using -a and -p flags

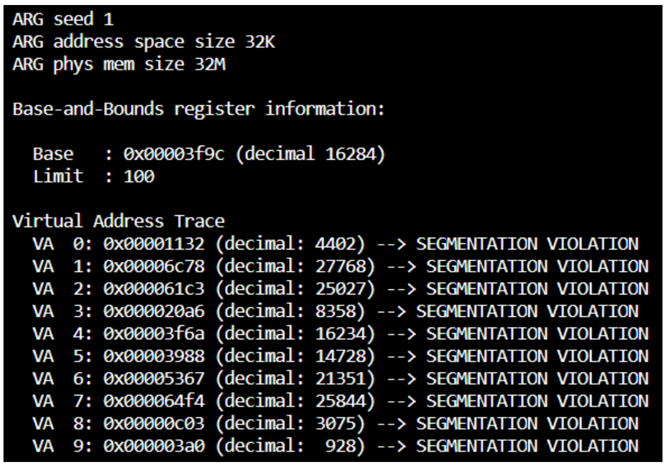
Here, we use `-s 0 -n 10 -a 16K -p 16M -l 16K -c` flags to get a completely valid list of VAs.



```
ARG seed 0
ARG address space size 16K
ARG phys mem size 16M

Base-and-Bounds register information:

  Base   : 0x00d82c07 (decimal 14167047)
  Limit  : 16384

Virtual Address Trace
  VA  0: 0x00003082 (decimal: 12418) --> VALID: 0x00d85c89 (decimal: 14179465)
  VA  1: 0x00001aea (decimal: 6890) --> VALID: 0x00d846f1 (decimal: 14173937)
  VA  2: 0x00001092 (decimal: 4242) --> VALID: 0x00d83c99 (decimal: 14171289)
  VA  3: 0x000020b8 (decimal: 8376) --> VALID: 0x00d84cbf (decimal: 14175423)
  VA  4: 0x000019ea (decimal: 6634) --> VALID: 0x00d845f1 (decimal: 14173681)
  VA  5: 0x00003229 (decimal: 12841) --> VALID: 0x00d85e30 (decimal: 14179888)
  VA  6: 0x00001369 (decimal: 4969) --> VALID: 0x00d83f70 (decimal: 14172016)
  VA  7: 0x00001e80 (decimal: 7808) --> VALID: 0x00d84a87 (decimal: 14174855)
  VA  8: 0x00002556 (decimal: 9558) --> VALID: 0x00d8515d (decimal: 14176605)
  VA  9: 0x00003a1e (decimal: 14878) --> VALID: 0x00d86625 (decimal: 14181925)
```

Figure 11: Results

This time, we use `-s 1 -n 10 -l 100 -c -b 16284 -a 32K -p 32M` flags to see what happens:



```
ARG seed 1
ARG address space size 32K
ARG phys mem size 32M

Base-and-Bounds register information:

  Base   : 0x00003f9c (decimal 16284)
  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
```

Figure 12: Results

We can see that we have a lot of segmentation violations. This is because the limit is much lesser than the limits needed by the generated VAs.

1.5 Graphs needed

Here, I have used three seed values- 0, 1 and 2. I've used a python script to auto-generate the graph, which is included in the submission. This is the output I got:

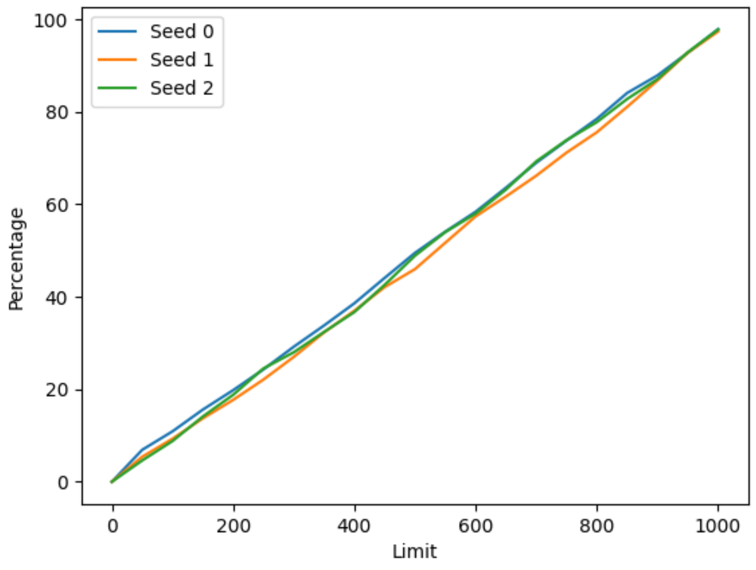


Figure 13: Results

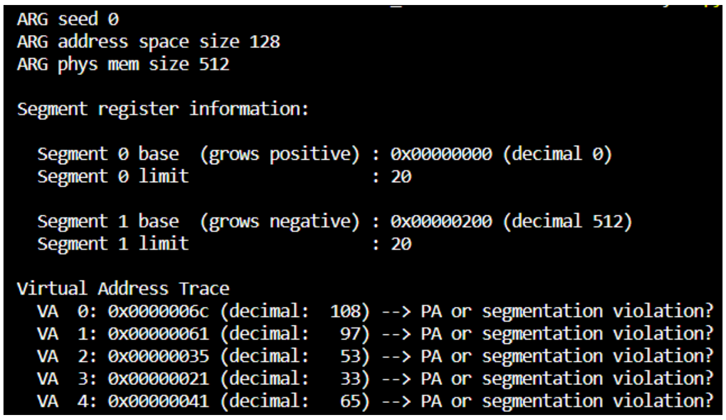
We can see that this is roughly linear, and the percentage of Valid Virtual Addresses increase as the limit increases.

2 Part 2

We have the program `segmentation.py` which allows us to see how addresses are translated in a system with segmentation.

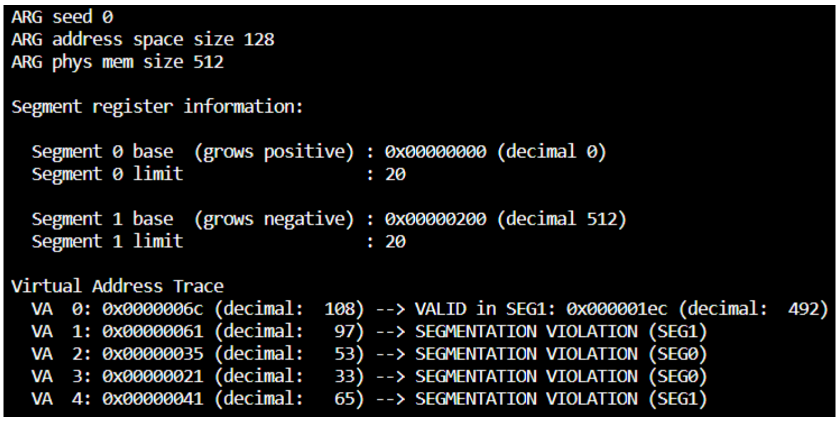
2.1 Translating addresses

In all cases, the start of the address space has to map to segment 0 of the PA, while the last addresses in the address space has to map to segment 1, which has base 512 and limit 20. Therefore, 0 to 19 in VA would map to 0 to 19 in PA while 108 to 127 in VA map to 492-511 in PA. Now, let us run the program for each seed value:
Here, only 108 would have a valid mapping, deriving from our earlier conclusion above. Therefore, 108 would map to 492 in PA.
We can see that we have got the correct answer.



```
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?
```

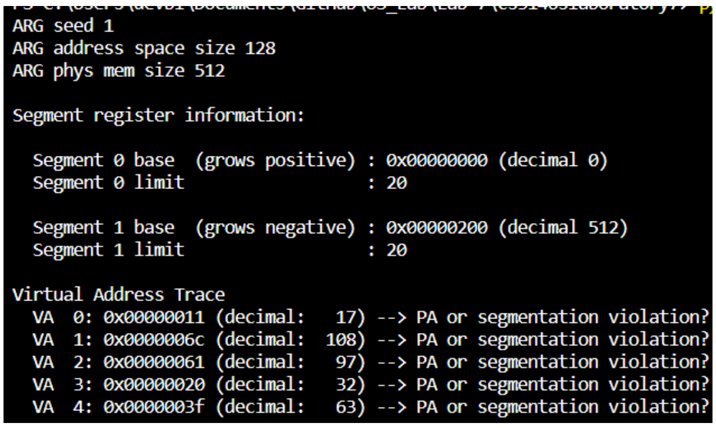
Figure 14: Results



```
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)
```

Figure 15: Results

Similarly, we do the same for seed 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?
```

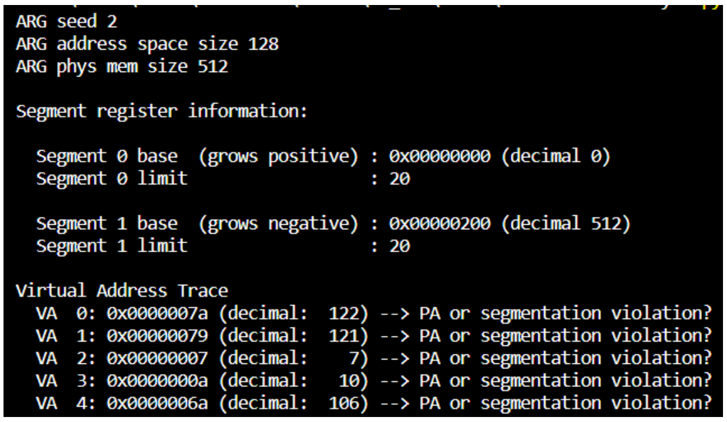
Figure 16: Results

Here, 17 and 108 are valid, and they map to 17 itself and 492 respectively. Now we do the same for seed 2:
Here, 122, 121, 7 and 10 are valid. They translate to 506, 505, 7 and 10 respectively.

2.2 A couple of questions

We can answer these questions based on the observations we made on the last question:

- Highest legal virtual address in segment 0 is 19
- Lowest legal virtual address in segment 1 is 108

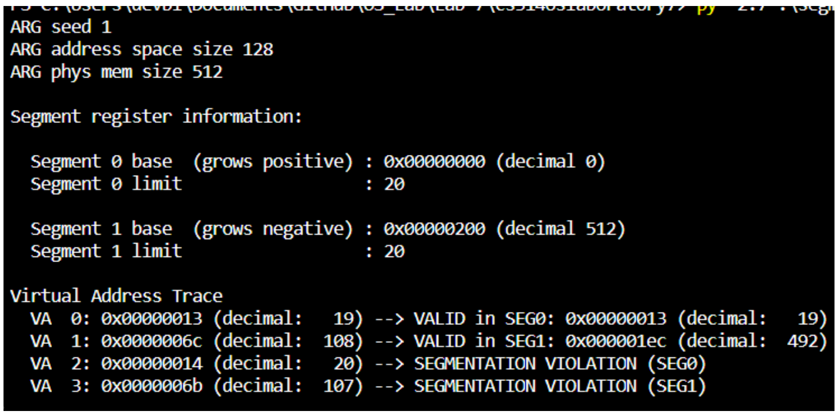


```
ARG seed 2
ARG address space size 128
ARG phys mem size 512

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?
```

Figure 17: Results

- Lowest and highest illegal addresses in this entire address space is 20 and 107 respectively.

Now, to test if we are correct, we can run the program with these flags - `-a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1 -A 19,108,20,107 -c`. We get this:



```
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: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492)
  VA  2: 0x00000014 (decimal:   20) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x0000006b (decimal:  107) --> SEGMENTATION VIOLATION (SEG1)
```

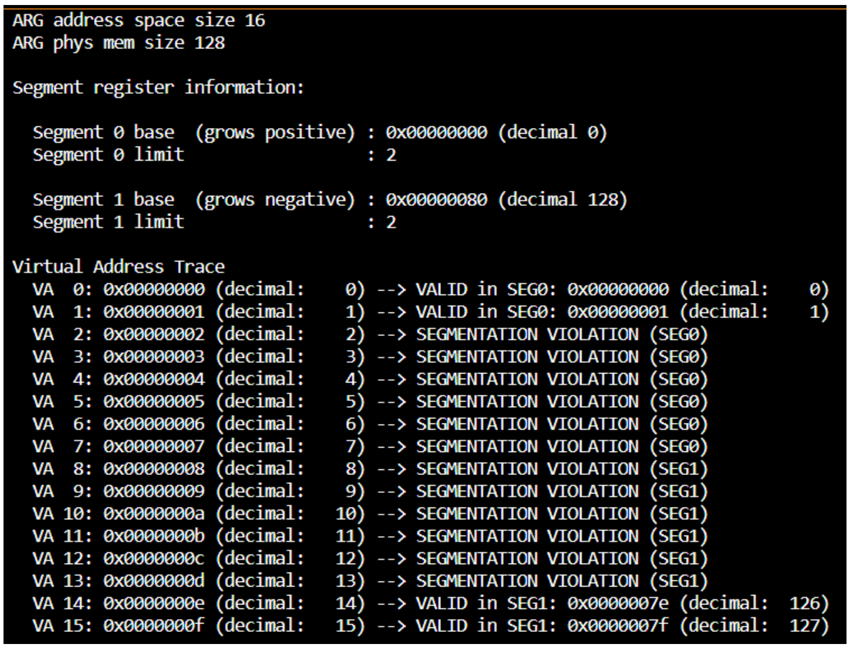
Figure 18: Results

2.3 Let's say you have a tiny 16-byte address space..

In this question, we have to set the base and bounds for both segments. We can set them as:

- Segment 0: Base = 0, Bound = 2
- Segment 1: Base = 128, Bound = 2

Now, we run the program with these flags:
`-a 16 -p 128 -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 --b0 0 --l0 2 --b1 128 --l1 2 -c` We can see that this satisfies given condition:



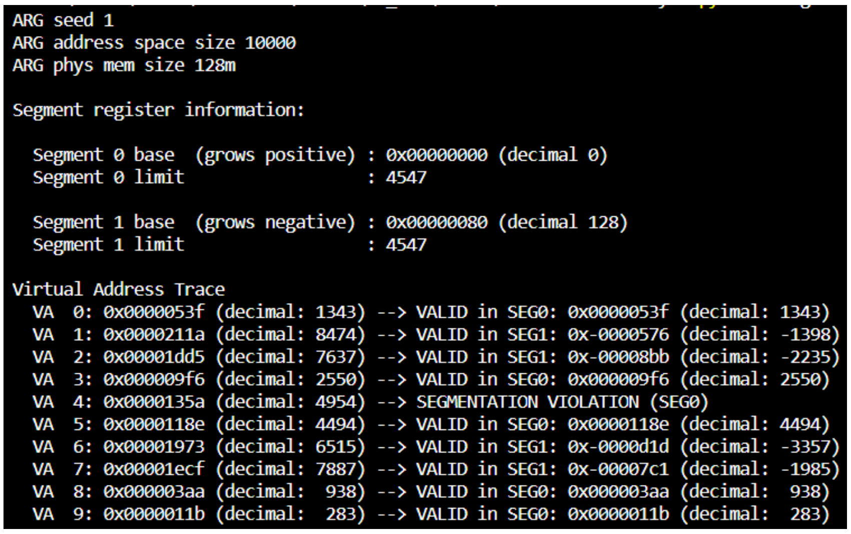
```
ARG address space size 16
ARG phys mem size 128

Segment register information:

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

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

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

Figure 19: Results

2.4 Assume we want to generate a problem..

To ensure that the majority of the virtual addresses generated are valid, we need to set the size of the valid memory to be approximately 90% of the entire virtual memory size. This means that in a system with virtual address space of size 'a', if 'b' and 'B' are the bounds for each of the segments, the sum of 'b' and 'B' should be roughly equal to 0.9 times 'a'. In practice, we may need to set the sum of 'b' and 'B' slightly higher than 0.9 times 'a' to achieve the desired level of validity.

For example, let's say we have a system with virtual address space of size 1000 units. We want to ensure that roughly 90% of the virtual addresses generated are valid. To achieve this, we would need to set the sum of 'b' and 'B' to be approximately equal to 0.9 times 1000, which is 900 units. However, in practice, we may need to set the sum of 'b' and 'B' slightly higher than 900 units to ensure that we achieve the desired level of validity. We use these flags to achieve this:
`-a 10000 -p 128m -s 1 --b0 0 --l0 4547 --b1 128 --l1 4547 -n 10 -c`
We can see that we have got a 90% rate of valid addresses:

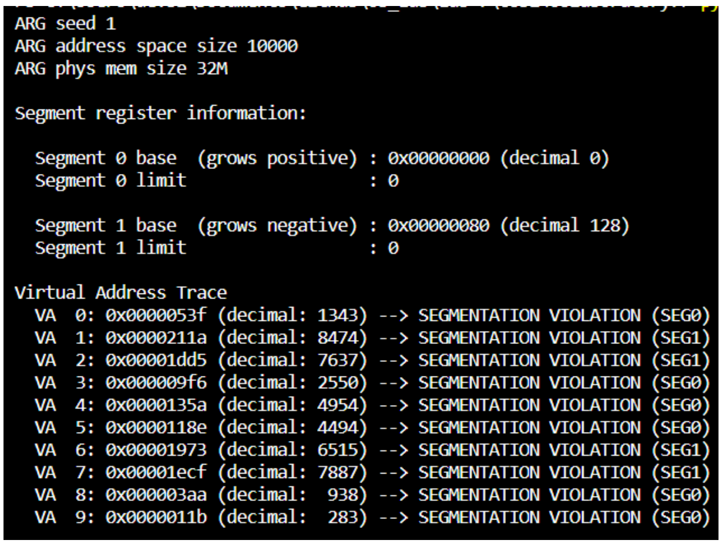


```
ARG seed 1
ARG address space size 10000
ARG phys mem size 128m

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 4547

  Segment 1 base  (grows negative) : 0x00000080 (decimal 128)
  Segment 1 limit                  : 4547

Virtual Address Trace
  VA  0: 0x0000053f (decimal: 1343) --> VALID in SEG0: 0x0000053f (decimal: 1343)
  VA  1: 0x0000211a (decimal: 8474) --> VALID in SEG1: 0x-0000576 (decimal: -1398)
  VA  2: 0x00001dd5 (decimal: 7637) --> VALID in SEG1: 0x-00008bb (decimal: -2235)
  VA  3: 0x000009f6 (decimal: 2550) --> VALID in SEG0: 0x000009f6 (decimal: 2550)
  VA  4: 0x0000135a (decimal: 4954) --> SEGMENTATION VIOLATION (SEG0)
  VA  5: 0x0000118e (decimal: 4494) --> VALID in SEG0: 0x0000118e (decimal: 4494)
  VA  6: 0x00001973 (decimal: 6515) --> VALID in SEG1: 0x-0000d1d (decimal: -3357)
  VA  7: 0x00001ecf (decimal: 7887) --> VALID in SEG1: 0x-00007c1 (decimal: -1985)
  VA  8: 0x000003aa (decimal:  938) --> VALID in SEG0: 0x000003aa (decimal:  938)
  VA  9: 0x0000011b (decimal:  283) --> VALID in SEG0: 0x0000011b (decimal:  283)
```

Figure 20: Results

2.5 Can you run the simulator such that..

Yes, we can just set the bounds of all segments to zero. We set the flags to: `-a 10000 -p 32M -s 1 --b0 0 --l0 0 --b1 128 --l1 0 -n 10 -c` and run the program. We get this output:



```
ARG seed 1
ARG address space size 10000
ARG phys mem size 32M

Segment register information:

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

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

Virtual Address Trace
  VA  0: 0x0000053f (decimal: 1343) --> SEGMENTATION VIOLATION (SEG0)
  VA  1: 0x0000211a (decimal: 8474) --> SEGMENTATION VIOLATION (SEG1)
  VA  2: 0x00001dd5 (decimal: 7637) --> SEGMENTATION VIOLATION (SEG1)
  VA  3: 0x000009f6 (decimal: 2550) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x0000135a (decimal: 4954) --> SEGMENTATION VIOLATION (SEG0)
  VA  5: 0x0000118e (decimal: 4494) --> SEGMENTATION VIOLATION (SEG0)
  VA  6: 0x00001973 (decimal: 6515) --> SEGMENTATION VIOLATION (SEG1)
  VA  7: 0x00001ecf (decimal: 7887) --> SEGMENTATION VIOLATION (SEG1)
  VA  8: 0x000003aa (decimal:  938) --> SEGMENTATION VIOLATION (SEG0)
  VA  9: 0x0000011b (decimal:  283) --> SEGMENTATION VIOLATION (SEG0)
```

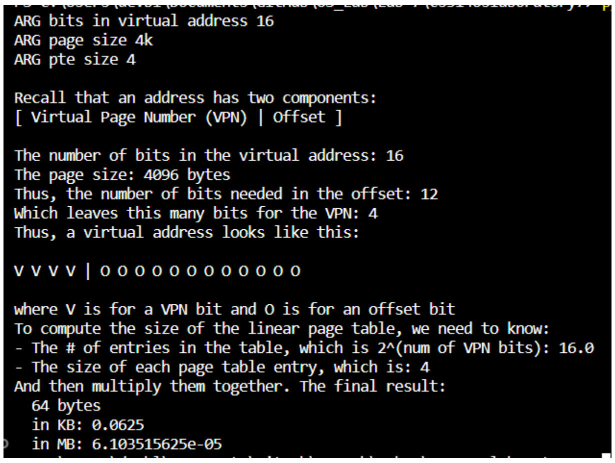
Figure 21: Results

We can see that all VAs generated have faults.

3 Linear Page Tables

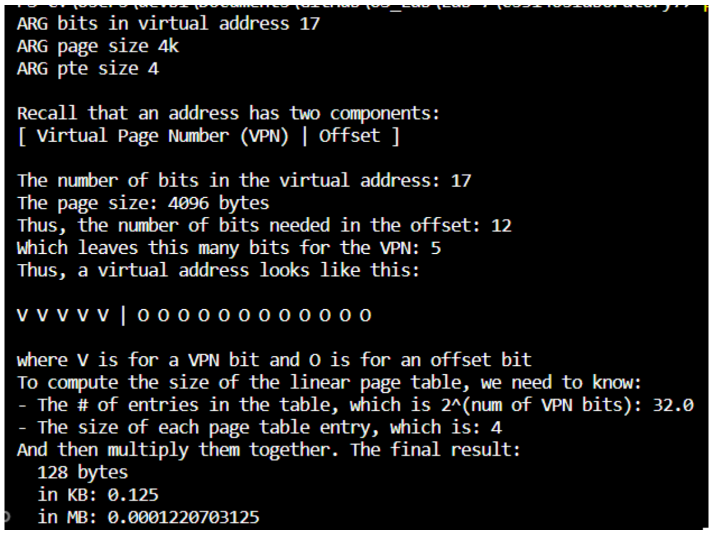
When we increase the address space bit, the page time (time required to access a page) doubles. This means that for every bit increase in the Virtual Page Number (VPN), we will have twice as many Page Table Entries (PTEs). The total number of PTEs can be calculated by multiplying 2 with the number of VPN bits.

For example, if we have a system with 16-bit address space, the page table size would be 6 MB. If we increase the address space to 17 bits, the number of VPN bits increases by 1 and the page table size doubles to 12 MB. Similarly, if we increase the address space to 20 bits, the number of VPN bits increases by 4 and the page table size becomes 6 MB multiplied by 2î6 (i.e., the number of possible VPN values), resulting in a page table size of 96 MB.
These are demonstrated in these screenshots:



```
ARG bits in virtual address 16
ARG page size 4k
ARG pte size 4

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

The number of bits in the virtual address: 16
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 4
Thus, a virtual address looks like this:

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

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 16.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  64 bytes
  in KB: 0.0625
  in MB: 6.103515625e-05
```

Figure 22: Results



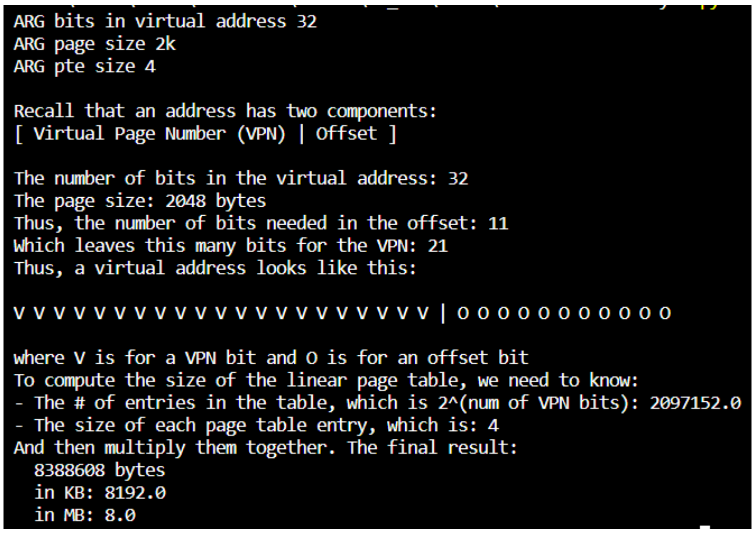
```
ARG bits in virtual address 17
ARG page size 4k
ARG pte size 4

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

The number of bits in the virtual address: 17
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 5
Thus, a virtual address looks like this:

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

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 32.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  128 bytes
  in KB: 0.125
  in MB: 0.0001220703125
```

Figure 23: Results

When we use different page sizes, the number of Page Table Entries (PTEs) changes based on the size of the address space and the page size. The number of PTEs decreases when the page size decreases, but the size of the address space remains the same. The page table size also decreases accordingly with the decrease in page size.

For instance, if we have an address space of size 'A' and a page size of 'P', the number of PTEs can be calculated by dividing the address space size by the page size, i.e., A/P. If we decrease the page size while keeping the address space size constant, the number of PTEs will reduce proportionately.

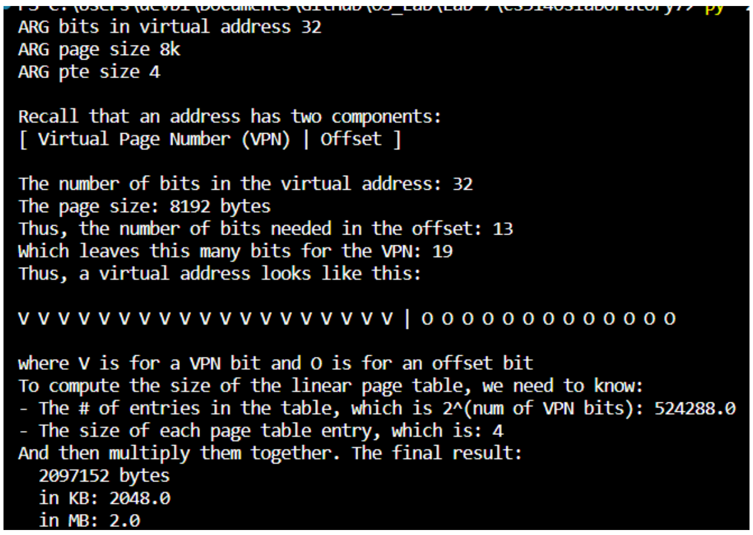
For example, let's say we have an address space of size 16 GB (i.e., $2\hat{3}4$ bytes) and we are using a page size of 2 KB. In this case, the number of PTEs would be $2\hat{3}4/2\hat{1}1$, which is equal to $2\hat{2}3$. This means that the page table size would be $2\hat{2}3$ times the size of a single PTE, which is typically 4 bytes. Hence, the total page table size would be 8 MB. However, if we were to decrease the page size to 1 KB, the number of PTEs would double to $2\hat{2}4$, and the page table size would be reduced accordingly. These screenshots demonstrate what I described:



```
ARG bits in virtual address 32
ARG page size 2k
ARG pte size 4

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

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

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

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

Figure 24: Results

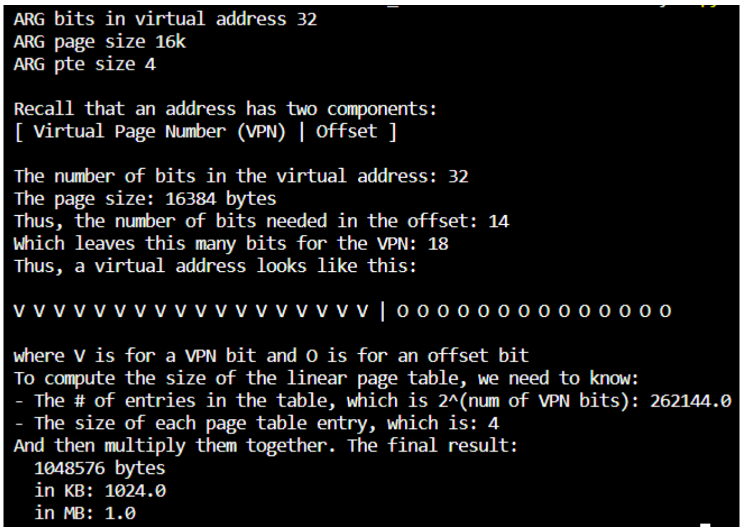
Here, page table size is 8MB for page size 2k.



```
ARG bits in virtual address 32
ARG page size 8k
ARG pte size 4

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

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

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

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

Figure 25: Results

Here, page table size is 2MB is page size is 8k.



```
ARG bits in virtual address 32
ARG page size 16k
ARG pte size 4

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

The number of bits in the virtual address: 32
The page size: 16384 bytes
Thus, the number of bits needed in the offset: 14
Which leaves this many bits for the VPN: 18
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 | O O O O O O O O O O O O O O

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 262144.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  1048576 bytes
  in KB: 1024.0
  in MB: 1.0
```

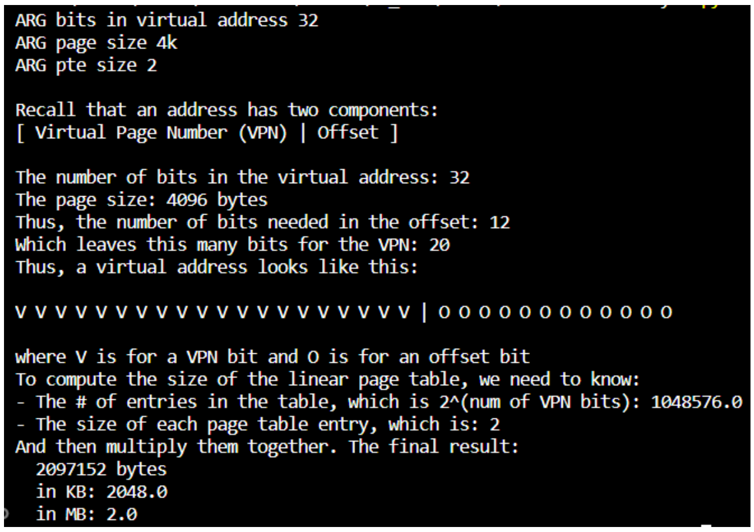
Figure 26: Results

Similarly, here page table size is 1MB for page size 16k.

When we talk about page tables, the size of the table is determined by the product of the size of each Page Table Entry (PTE) and the total number of entries. So, if we increase the size of each PTE, the size of the page table will grow in direct proportion to the increase in PTE size.

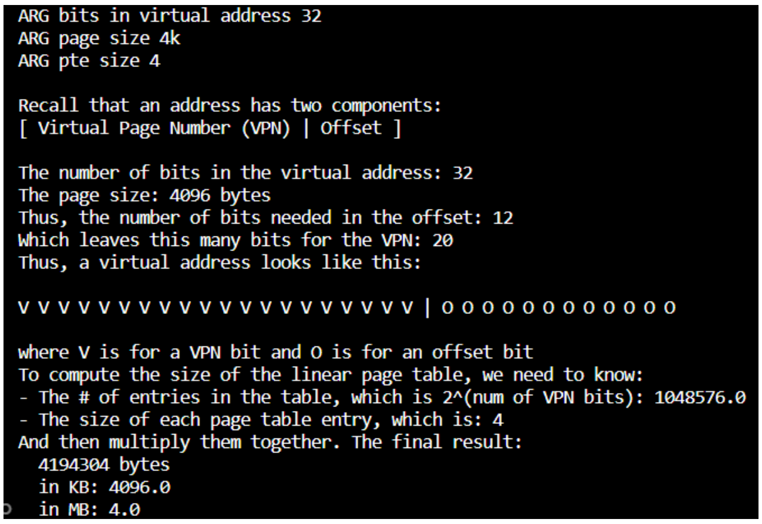
For example, if each PTE is 4 bytes long and the page table has a total of 1000 entries, then the page table size would be 4*1000 = 4 KB. If we increase the size of each PTE to 8 bytes while keeping the number of entries constant, then the size of the page table would double to 8 KB.

In summary, the size of the page table is directly proportional to the size of each PTE and the total number of entries, and increasing the size of each PTE will result in a larger page table. The screenshots in the next page demonstrate what I described.



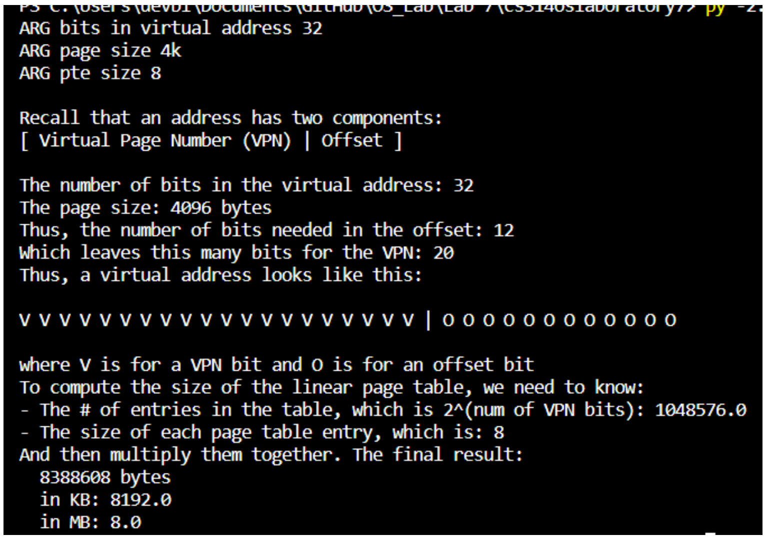
```
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 2

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

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

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

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 1048576.0
- The size of each page table entry, which is: 2
And then multiply them together. The final result:
  2097152 bytes
  in KB: 2048.0
  in MB: 2.0
```

Figure 27: Results



```
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 4

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

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

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

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

Figure 28: Results



```
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 8

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

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

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

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

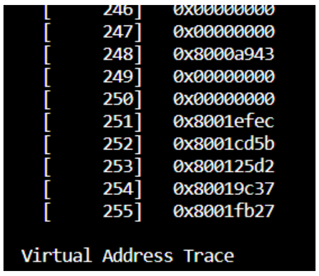
Figure 29: Results

4 Address Translation with linear page tables

4.1 Before doing any translation..

4.1.1 Address space Grows

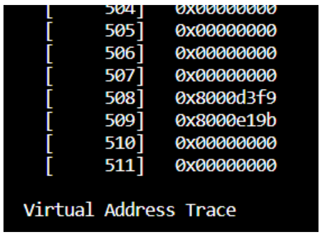
The size of each entry in the page table is the same, but the size of the page table increases as the address space increases.
Running with a as 1M:



```
[      246]   0x00000000
[      247]   0x00000000
[      248]   0x8000a943
[      249]   0x00000000
[      250]   0x00000000
[      251]   0x8001efec
[      252]   0x8001cd5b
[      253]   0x800125d2
[      254]   0x80019c37
[      255]   0x8001fb27

Virtual Address Trace
```

Figure 30: Results

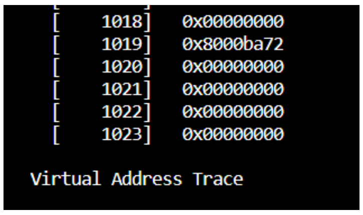
Running with a as 2m:



```
[      504]   0x00000000
[      505]   0x00000000
[      506]   0x00000000
[      507]   0x00000000
[      508]   0x8000d3f9
[      509]   0x8000e19b
[      510]   0x00000000
[      511]   0x00000000

Virtual Address Trace
```

Figure 31: Results

Running with a as 4m:



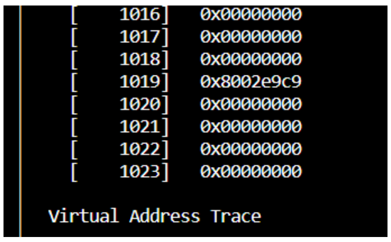
```
[     1018]   0x00000000
[     1019]   0x8000ba72
[     1020]   0x00000000
[     1021]   0x00000000
[     1022]   0x00000000
[     1023]   0x00000000

Virtual Address Trace
```

Figure 32: Results

We can see that the size of the page table has increased from 256 to 1024 while increasing address space from 1M to 4M.

4.1.2 Page size grows

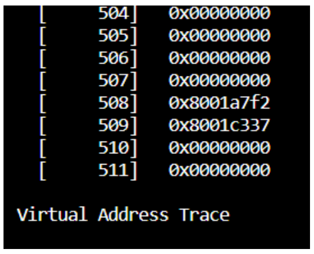
Running with p as 1k:



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

Virtual Address Trace
```

Figure 33: Results

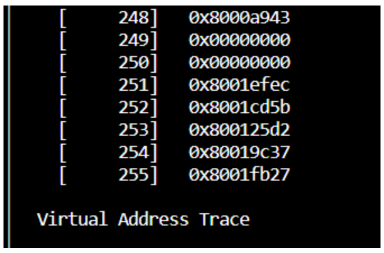
Running with p as 2k:



```
  [      504]   0x00000000
  [      505]   0x00000000
  [      506]   0x00000000
  [      507]   0x00000000
  [      508]   0x8001a7f2
  [      509]   0x8001c337
  [      510]   0x00000000
  [      511]   0x00000000

Virtual Address Trace
```

Figure 34: Results

Running with p as 4k:



```
  [      248]   0x8000a943
  [      249]   0x00000000
  [      250]   0x00000000
  [      251]   0x8001efec
  [      252]   0x8001cd5b
  [      253]   0x800125d2
  [      254]   0x80019c37
  [      255]   0x8001fb27

Virtual Address Trace
```

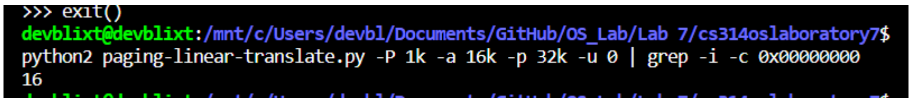
Figure 35: Results

We can see that as the page size grows, fewer pages are needed to cover the address space. Therefore, the size of the page table decreases as we increase the page size.

4.2 Now let's do some translations..

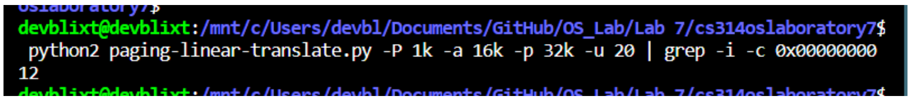
As we increase the value of u, the fraction of the address space used also increases. In the current scenario where each page is 1K in size and the address space (A) is 16K, the page table has 16 entries.

To elaborate further, as we increase the value of u from 0 to 100, more page table entries and physical pages are allocated. To determine the number of entries in the unallocated page table, we can use the grep tool to find the number of `0x00000000`.



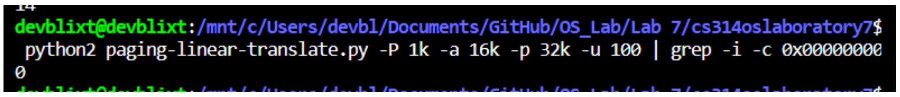
```
>>> exit()
devblixt@devblixt:/mnt/c/Users/devbl/Documents/GitHub/OS_Lab/Lab 7/cs314oslaboratory7$
python2 paging-linear-translate.py -P 1k -a 16k -p 32k -u 0 | grep -i -c 0x00000000
16
```

Figure 36: Results



```
devblixt@devblixt:/mnt/c/Users/devbl/Documents/GitHub/OS_Lab/Lab 7/cs314oslaboratory7$
 python2 paging-linear-translate.py -P 1k -a 16k -p 32k -u 20 | grep -i -c 0x00000000
12
```

Figure 37: Results



```
devblixt@devblixt:/mnt/c/Users/devbl/Documents/GitHub/OS_Lab/Lab 7/cs314oslaboratory7$
 python2 paging-linear-translate.py -P 1k -a 16k -p 32k -u 100 | grep -i -c 0x00000000
0
```

Figure 38: Results

As we can see, as we increase u the number of page table entries that are unallocated drops down.

4.3 Now let's try some different random seeds..

The three sets of flags have different values for the page size, virtual address space, and physical address space, resulting in varying degrees of realism. In the first set, each page is allotted only 8B, and at most 4 pages can be present in the address space, making it unrealistic. The second set is slightly more realistic, but still limited to at most 4 pages in the address space. The third set has a larger page size of 1MB, which is impractical as it would consume too much memory and could lead to underutilization of memory. The order of unrealistic-ness can be ranked as `3 > 1 > 2`. However, it is to be noted that all three of them are definitely unrealistic.

4.4 Use the program to try out..

The program code has certain requirements and limitations that need to be met in order for the program to work correctly. These limitations include:

- Ensuring that the address space size, physical memory size, and limits are all positive values.
- Making sure that physical memory and address space are both multiples of the page size.
- Ensuring that the page size is a power of 2.
- Ensuring that the page size is smaller than the physical memory size, but larger than the address space size.

If any of these conditions are not met, then the program will not function properly.