# CS - 314 OPERATING SYSTEMS LAB-7

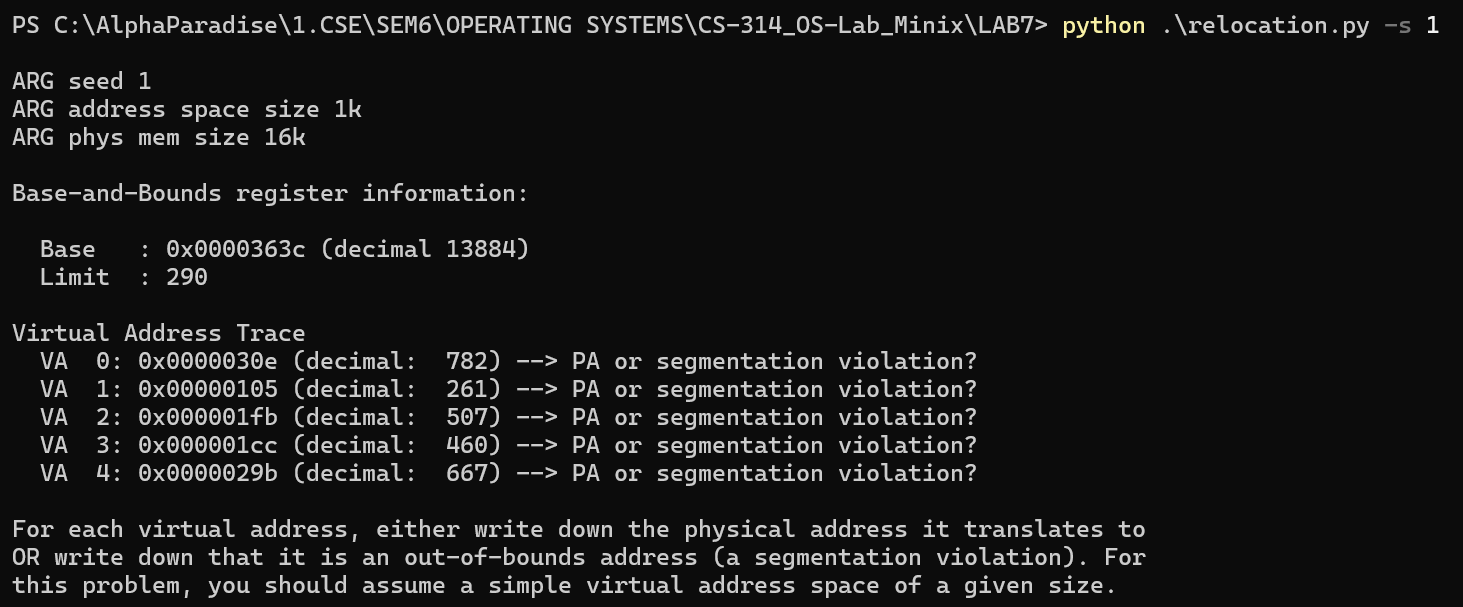
(K.S.N Manikanta, 210010050)

Note: I changed the Python files into Python 3 since there is no python2 installed on my PC. So, I used python instead of py2 in the command while running the files.

**QUESTION – 1**

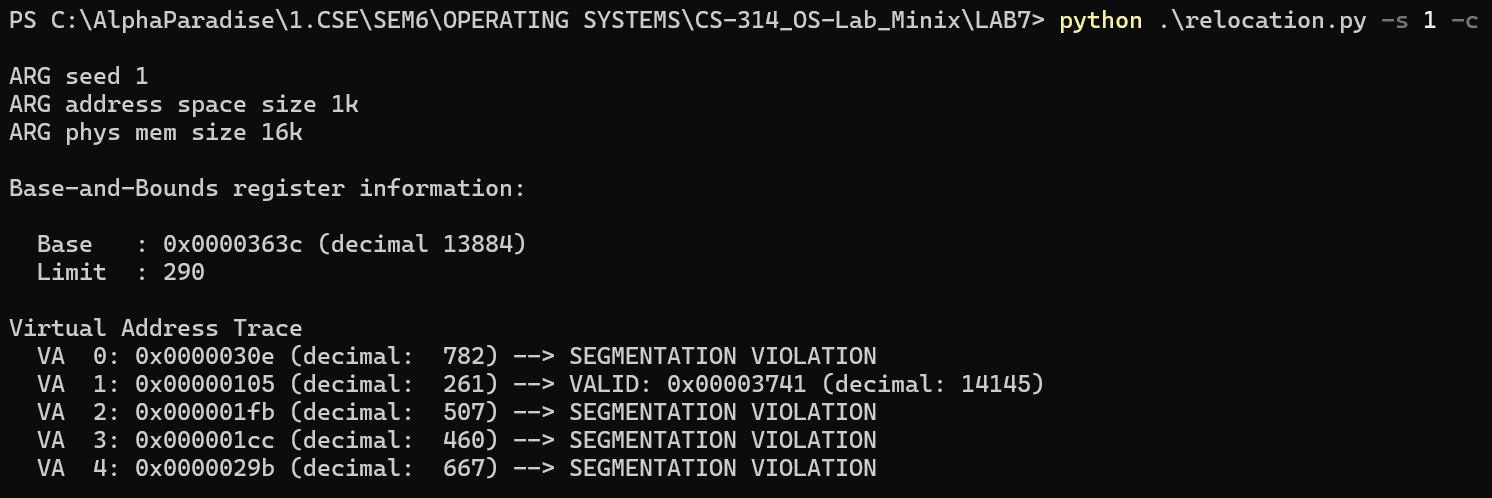
1. [Bound = Base + Limit & PA = Base + VA]. If calculated PA > Bound or VA > Limit, then it is a Segmentation violation, if PA < Bound or VA < Limit, then PA = Base + VA is valid.

**For seed 1: Virtual Address (VA) Trace => Base: 13884, Limit: 290**

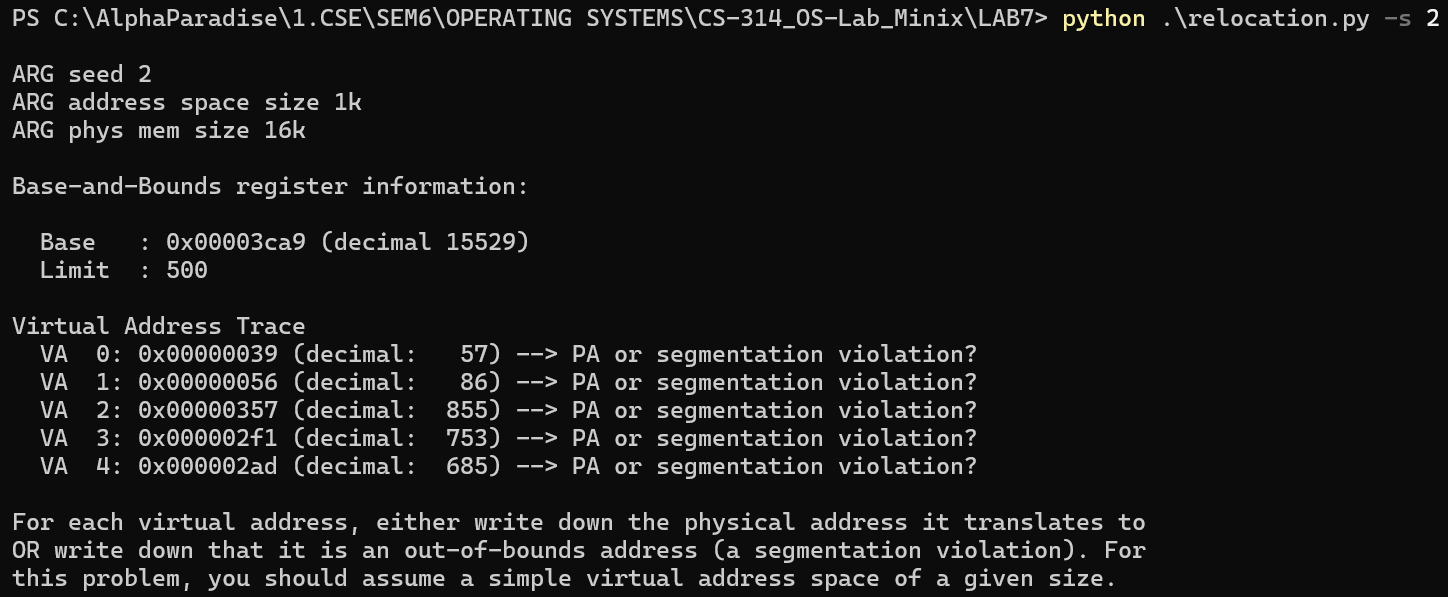


|  |  |  |  |
| --- | --- | --- | --- |
| VA No. | VA | In or out of Bounds | PA or Segmentation Violation |
| 0 | 782 | Out | Segmentation Violation (782 >290) |
| 1 | 261 | In | PA = 13884 + 261 = 14145 (0x00003741) |
| 2 | 507 | Out | Segmentation Violation (507 >290) |
| 3 | 460 | Out | Segmentation Violation (460 >290) |
| 4 | 667 | Out | Segmentation Violation (667 >290) |

This can be verified as below:

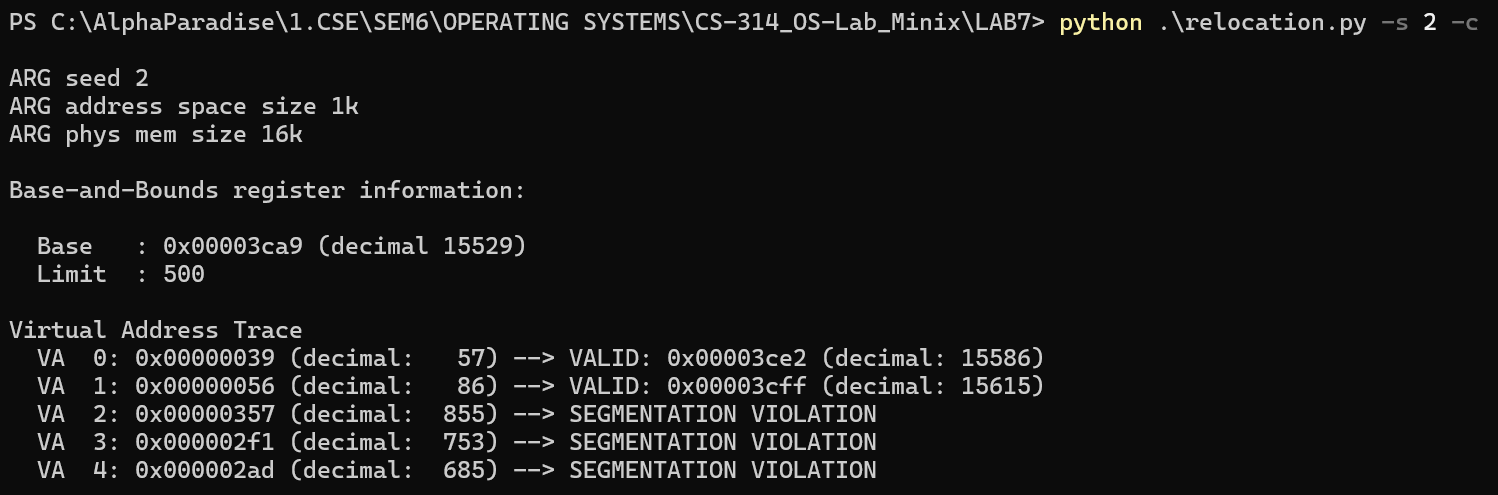


**For seed 2: Virtual Address Trace => Base: 15529, Limit: 500**

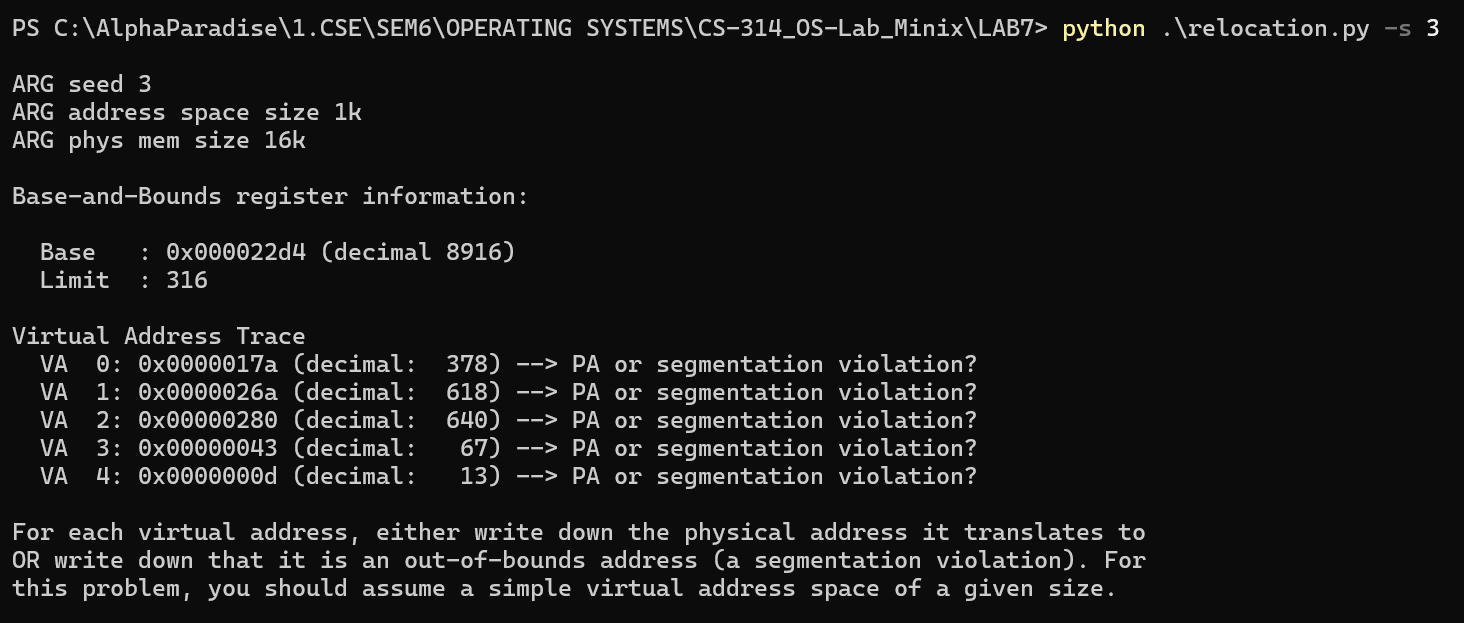


|  |  |  |  |
| --- | --- | --- | --- |
| VA No. | VA | In or out of Bounds | PA or Segmentation Violation |
| 0 | 57 | In | PA = 15529 + 57 = 15586 (0x00003ce2) |
| 1 | 86 | In | PA = 15529 + 86 = 15615 (0x00003cff) |
| 2 | 855 | Out | Segmentation Violation (855>500) |
| 3 | 753 | Out | Segmentation Violation (753 >500) |
| 4 | 685 | Out | Segmentation Violation (685 >500) |

This can be verified as below:

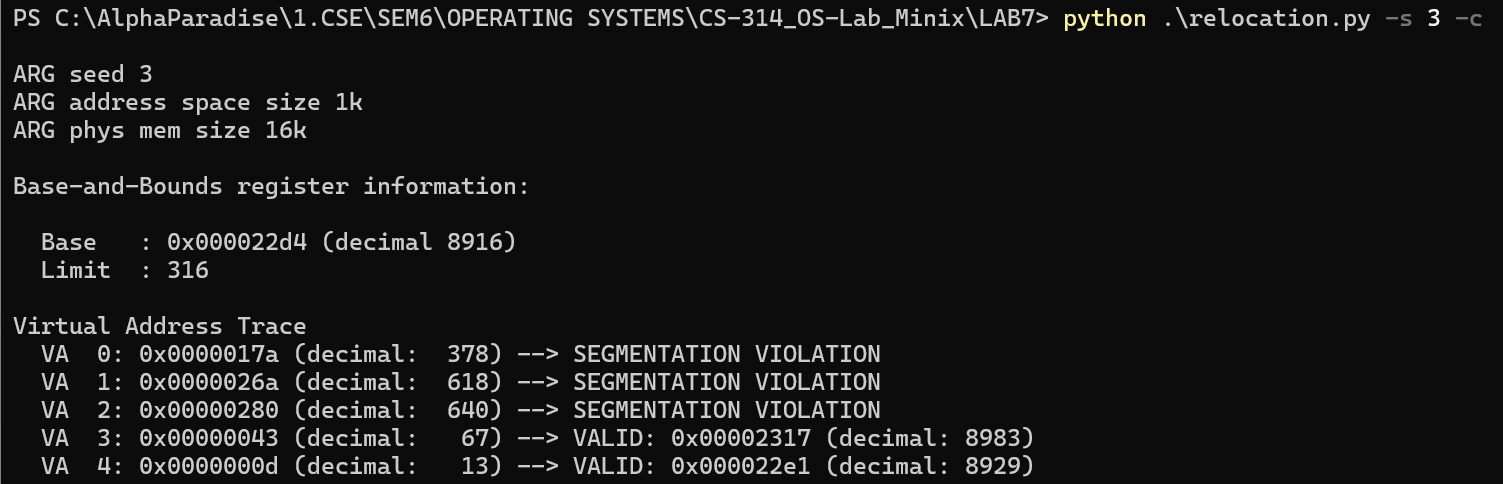
****

**For seed 3: Virtual Address Trace => Base: 8916, Limit: 316**

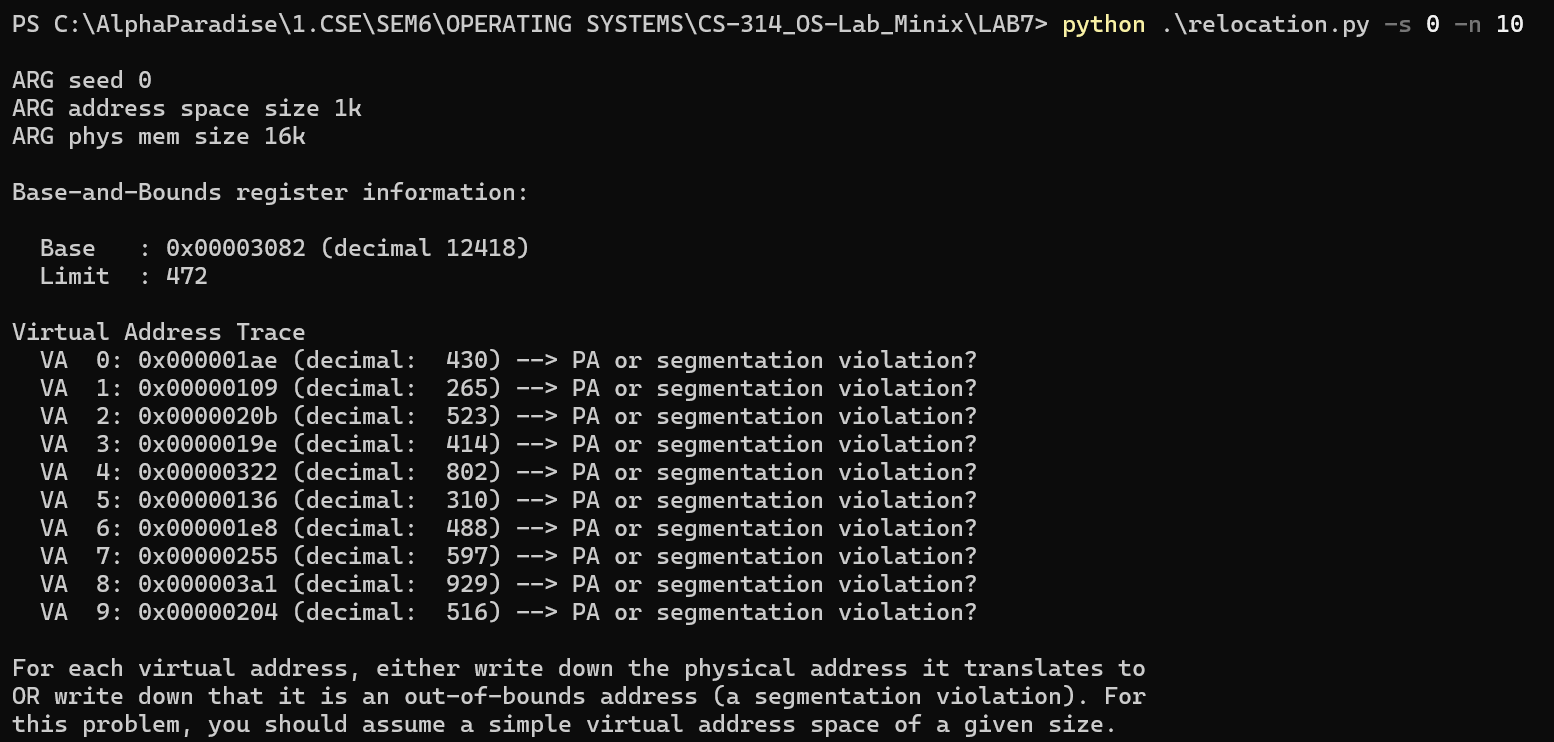


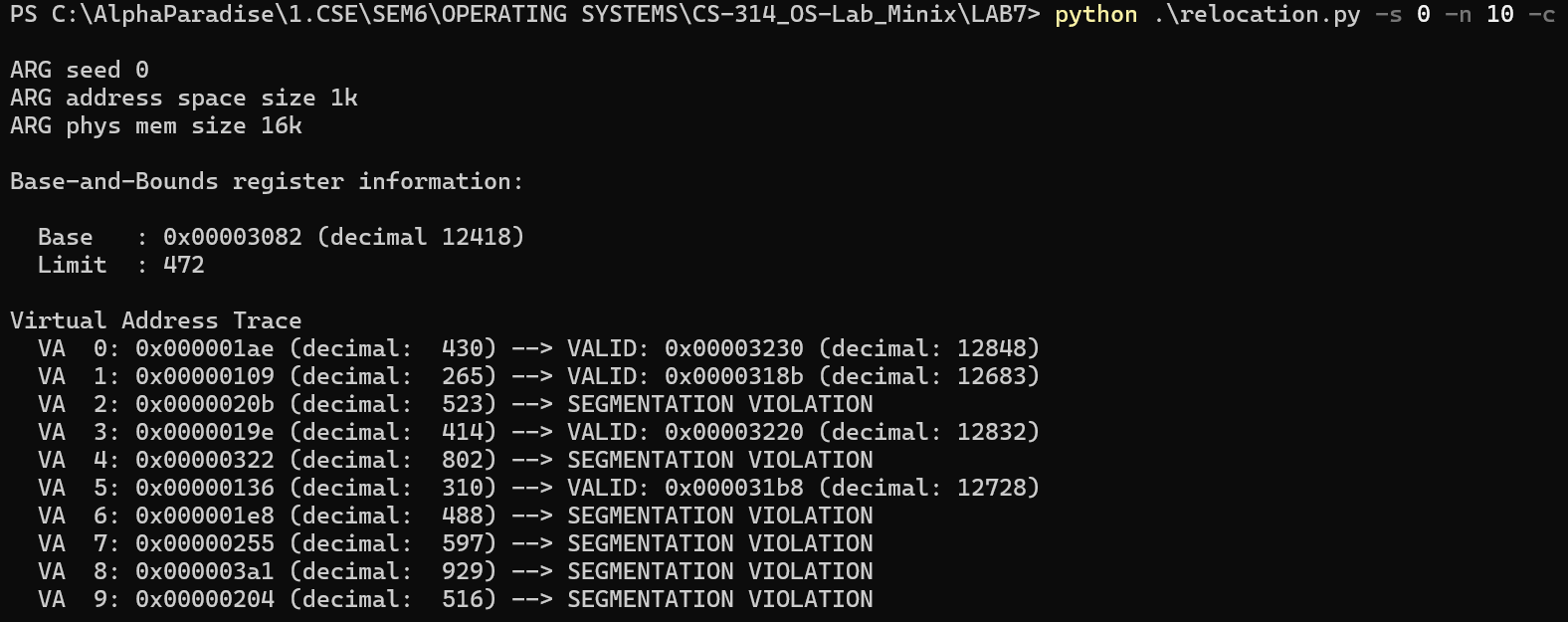
|  |  |  |  |
| --- | --- | --- | --- |
| VA No. | VA | In or Out of bounds | PA or Segmentation Violation |
| 0 | 378 | Out | Segmentation Violation (378>316) |
| 1 | 618 | Out | Segmentation Violation (618>316) |
| 2 | 640 | Out | Segmentation Violation (640>316) |
| 3 | 67 | In | PA = 8916 + 67 = 8983 (0x00002317) |
| 4 | 13 | In | PA = 8916 + 13 = 8929 (0x000022e1) |

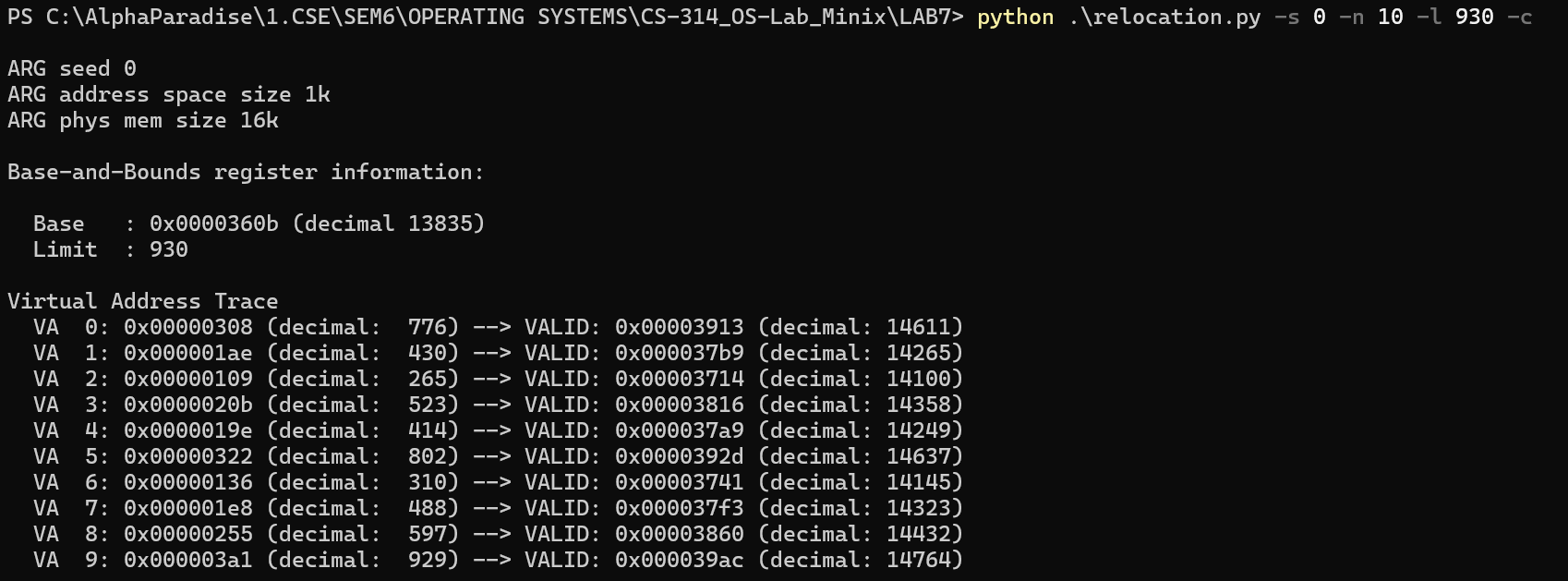
This can be verified as below:



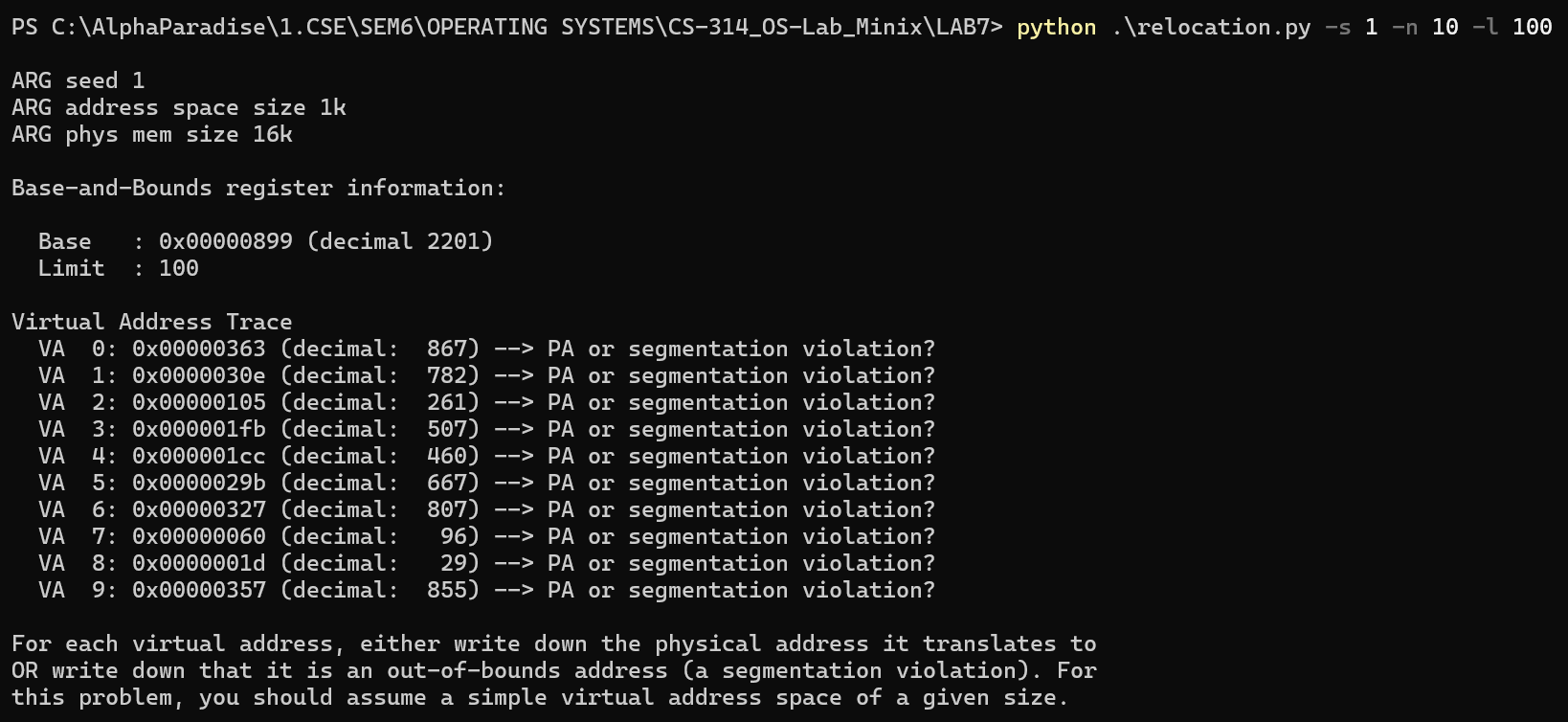
1. We can see from the screenshot below, that the Base is 12418, the Limit is 472 and the VA accesses are 430, 265, 523, 414, 802, 310, 488, 597, 929, 516. The highest of them being 929, We can set the **bounds register value to 930** so that all generated VAs are within the bounds.



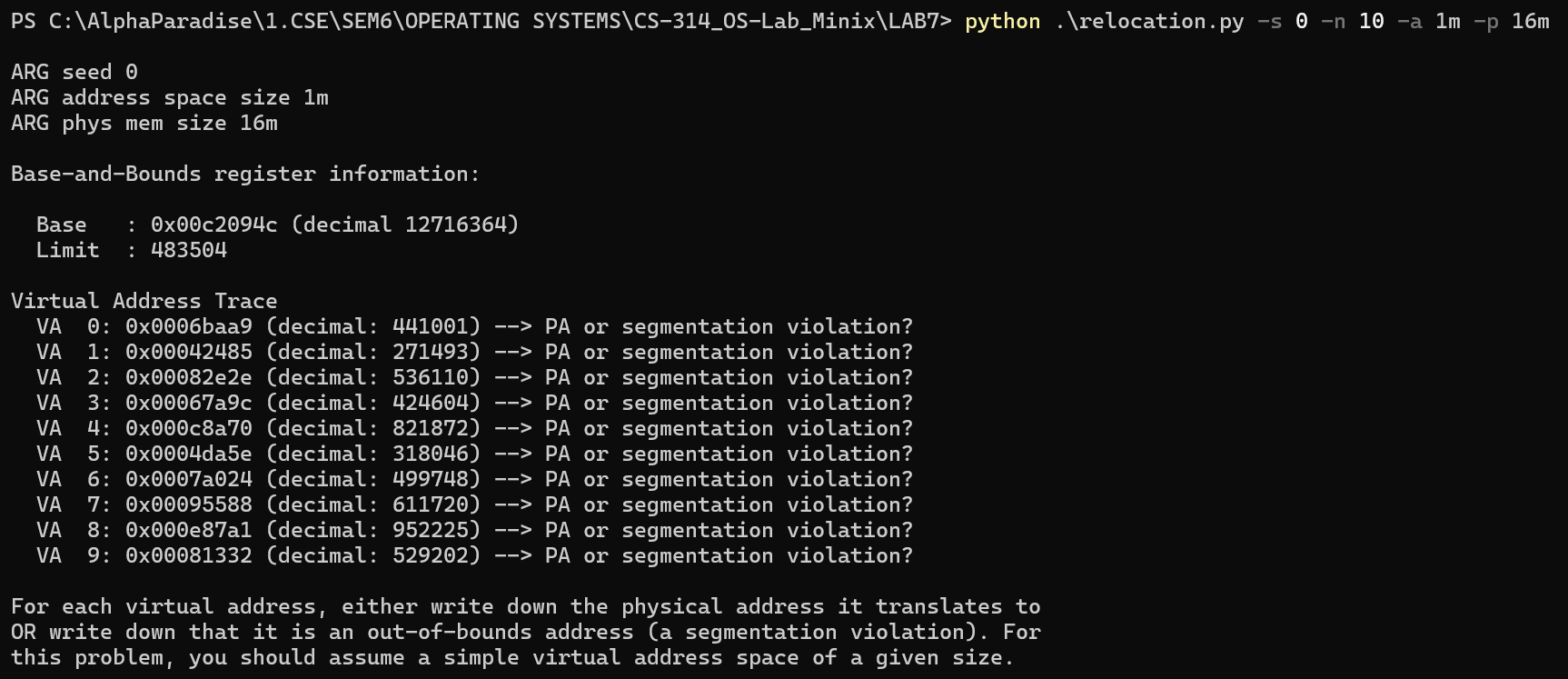


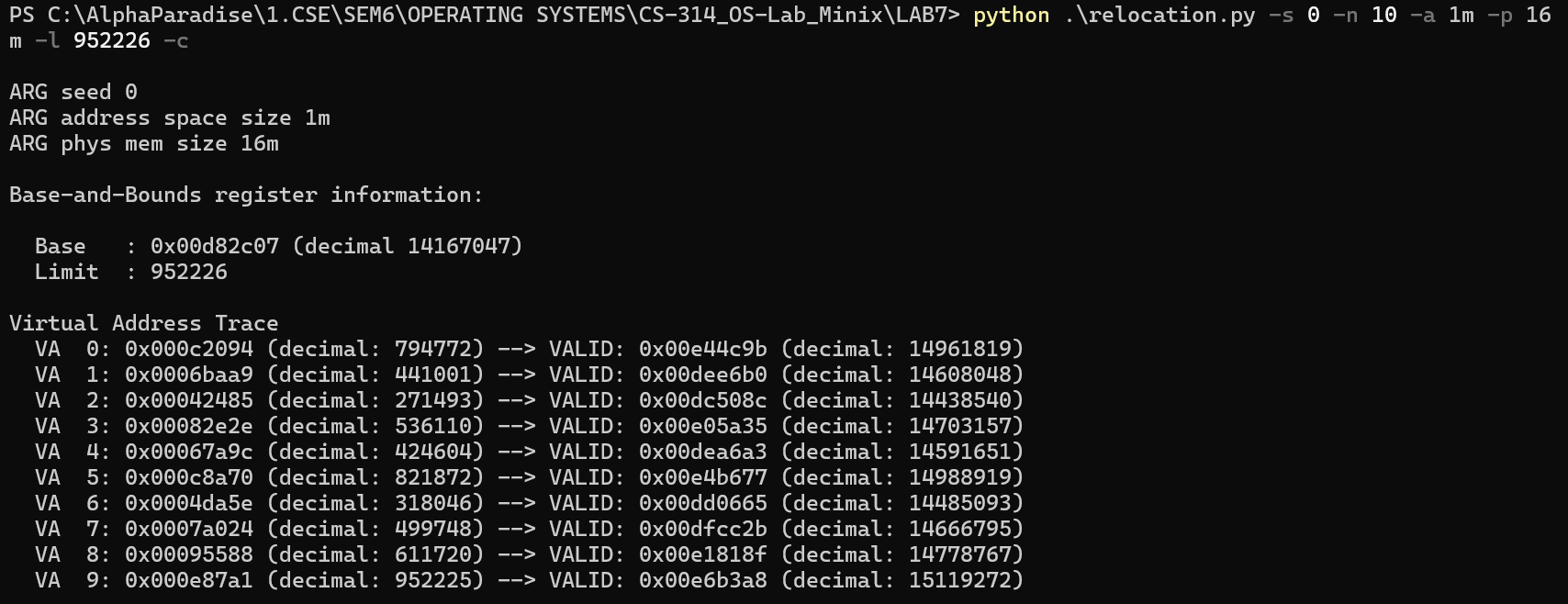


1. When we run with the given flags (seed = 1, no of VA accesses = 10, limit = 100), Size of physical memory = 16kB = 16384 bytes. Since the limit is 100, the **maximum value** that the **base** can be set to will be 16384 – 100 = **16284**.

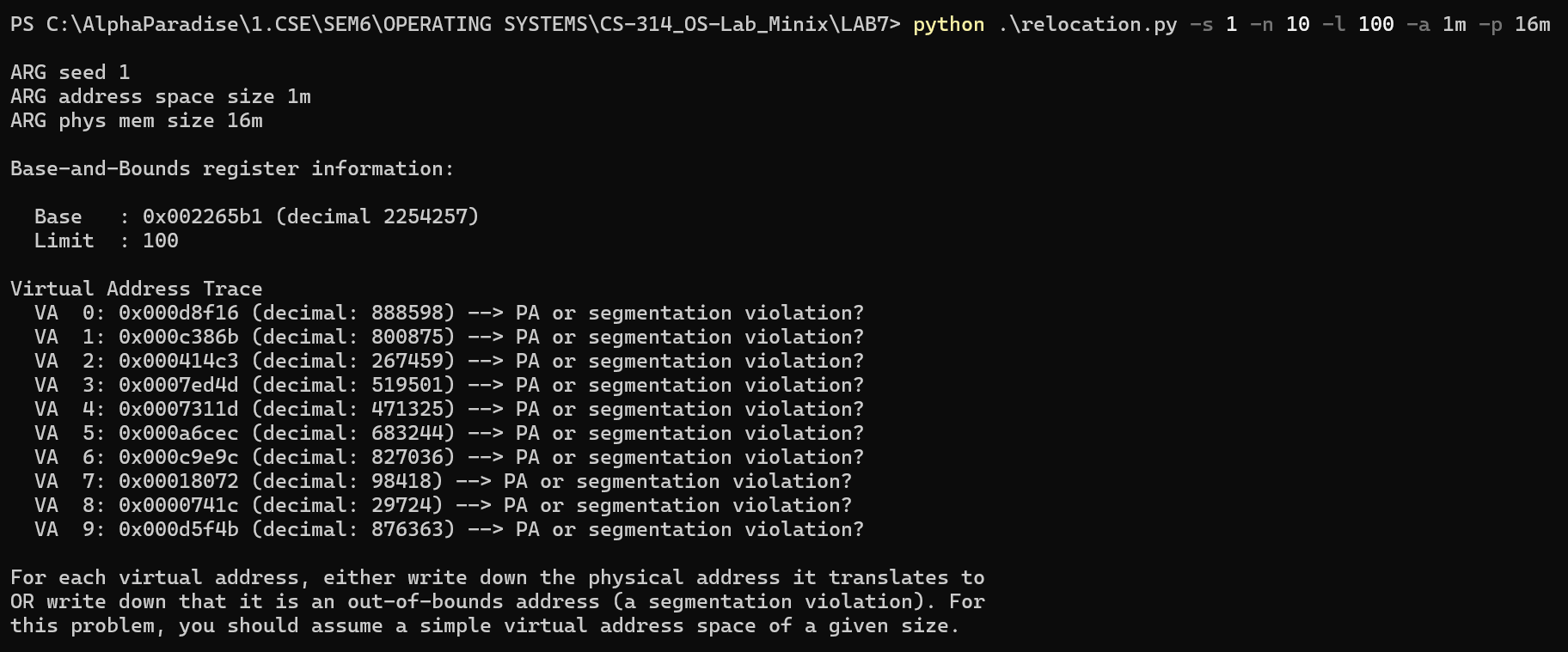


1. **Case 1**: Let's consider an address space of 1 MB and a Physical memory size of 16 MB.   
   Let’s run for part 2 and part 3 of this question 1 again. First, let's check for part 2 of this question 1: run with flags -s 0 -n 10 -a 1m -p 16m. Base is 12716364, Limit is 483504. Of all 10 VA accesses, 952225 is the highest/largest. So, if we set the **Limit to 952226,** all VAs will be within the bounds.

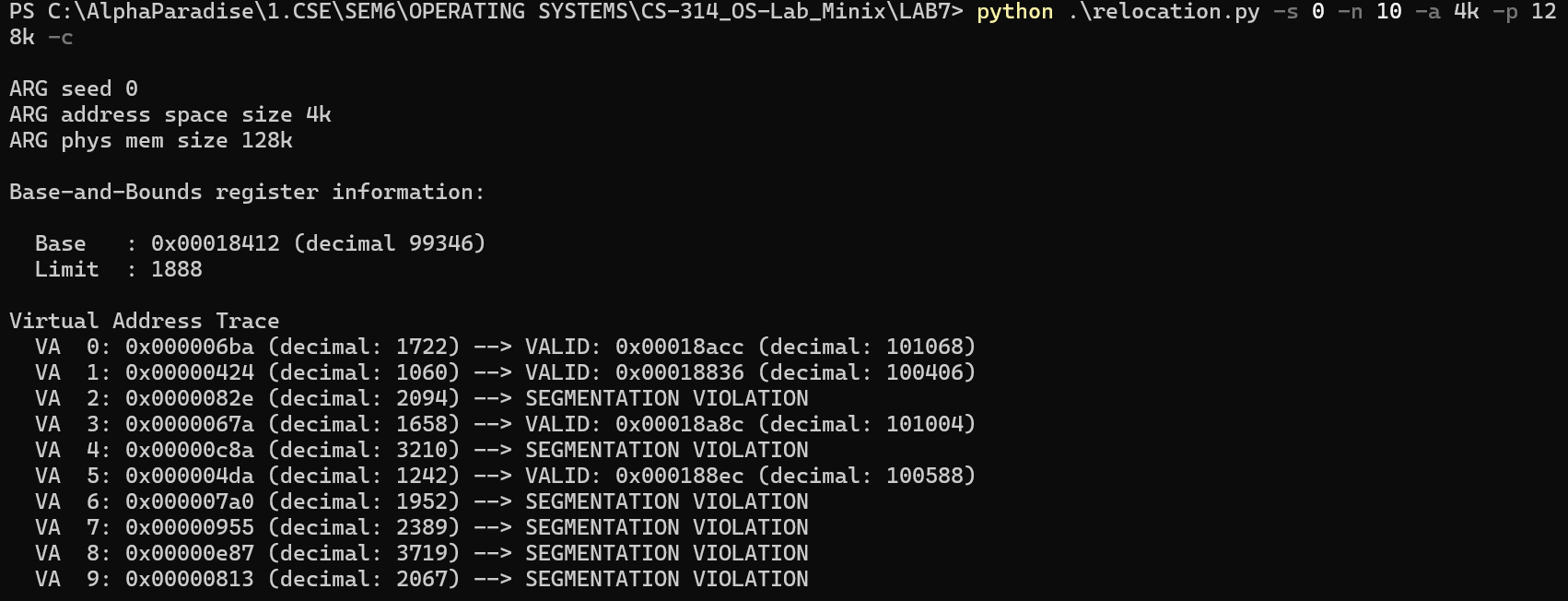


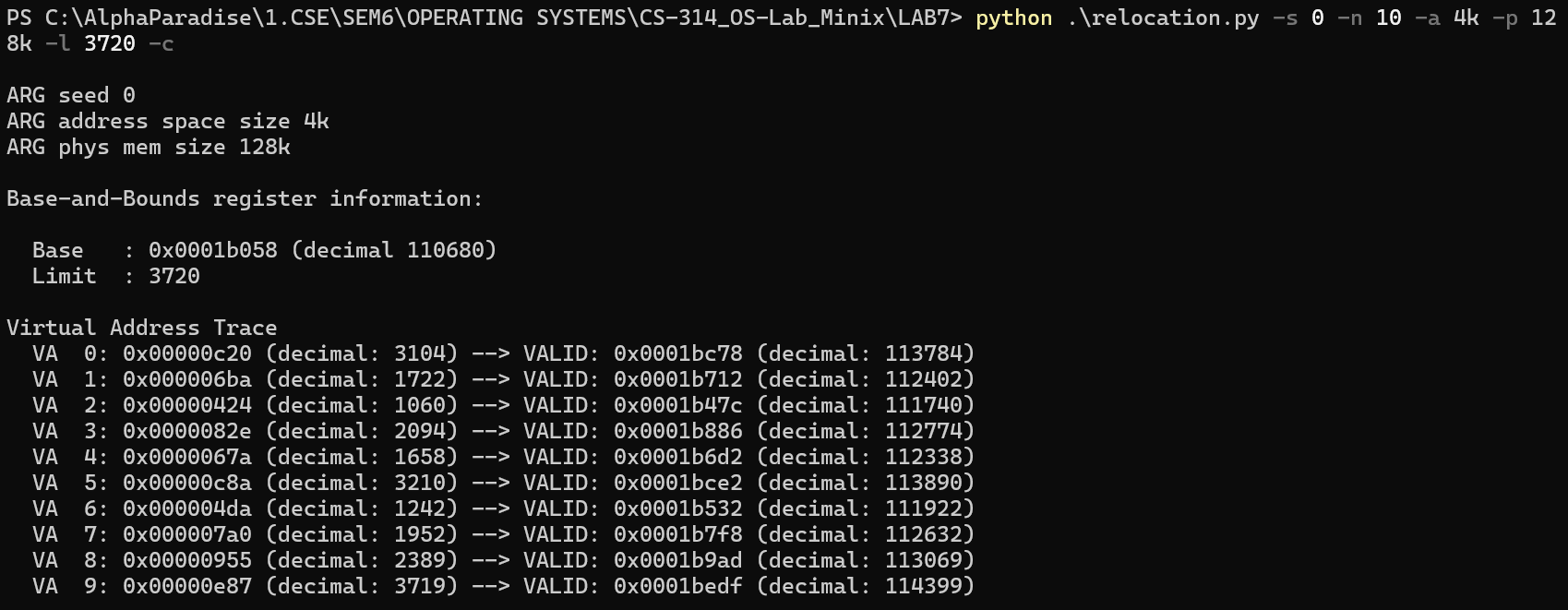


Next, let's check for part 3 of question 1: run with flags -s 1 -n 10 -l 100 -a 1m -p 16m. Physical memory size is 16 MB which is 16777216 bytes. Since the Limit is 100, the **maximum value** that the **base** can be set to will be 16777216 – 100 = **16777116.**

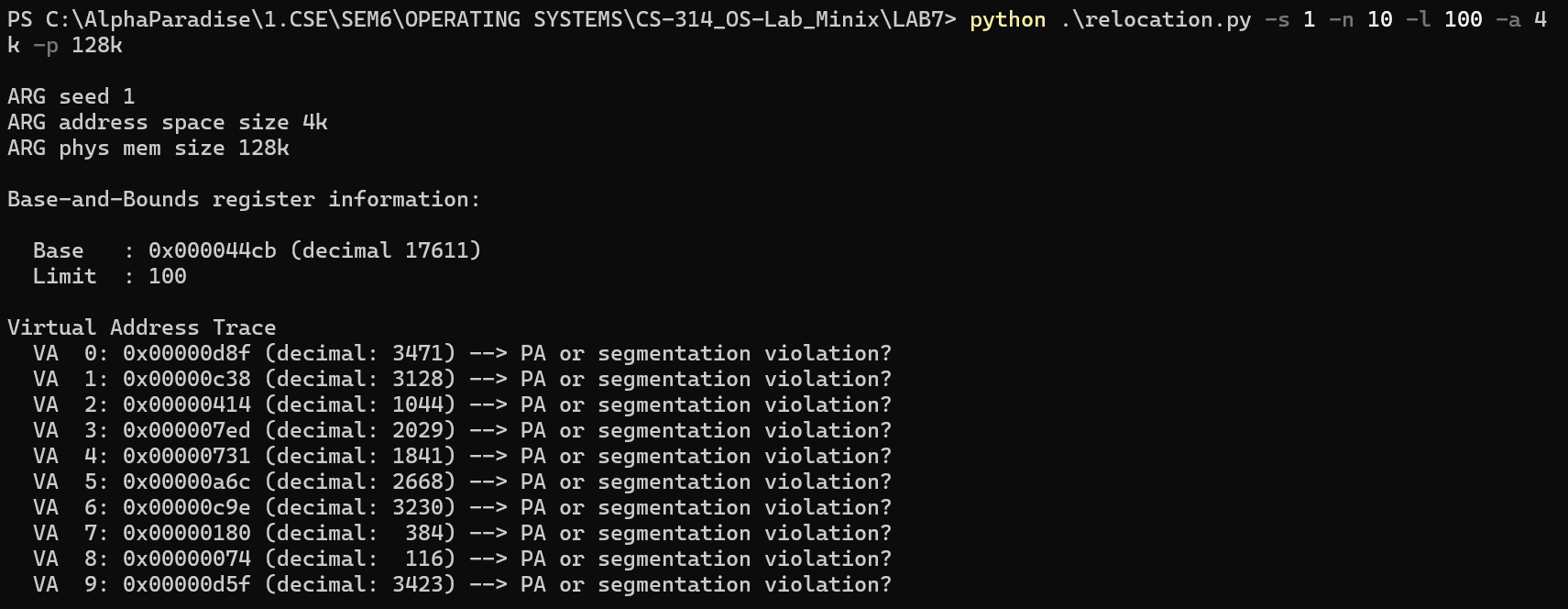


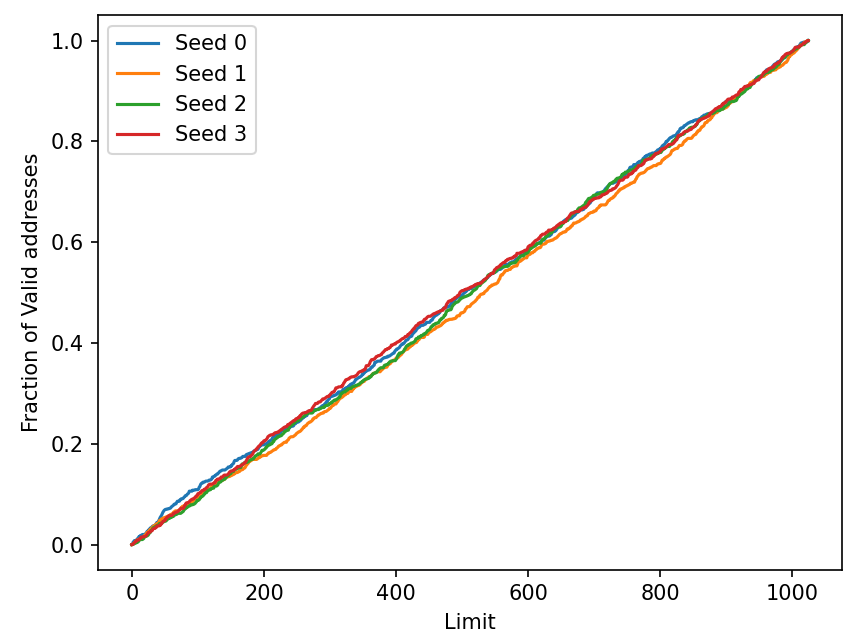
**Case 2:** Let's consider an address space of 4 KB and a Physical memory size of 128 KB.

Let’s run for part 2 and part 3 of this question 1 again. First, let's check for part 2 of this question 1: run with flags -s 0 -n 10 -a 4k -p 128k. Base is 18412 and Limit is 1888. Of all the accesses, 3719 is the highest. So, if we set the limit to 3720, all Vas will be within the bound.  




Next, let's check for part 3 of question 1: run with flags -s 1 -n 10 -l 100 -a 4k -p 128k. Physical memory size is 128 KB which is 131072 bytes. Since the Limit is 100, the **maximum value** that the **base** can be set to will be 131072 – 100 = **130972.**



1. The Graph can be found below: We can see an almost linear graph for all seeds as it is obvious that more addresses become valid with increasing value of the limit.

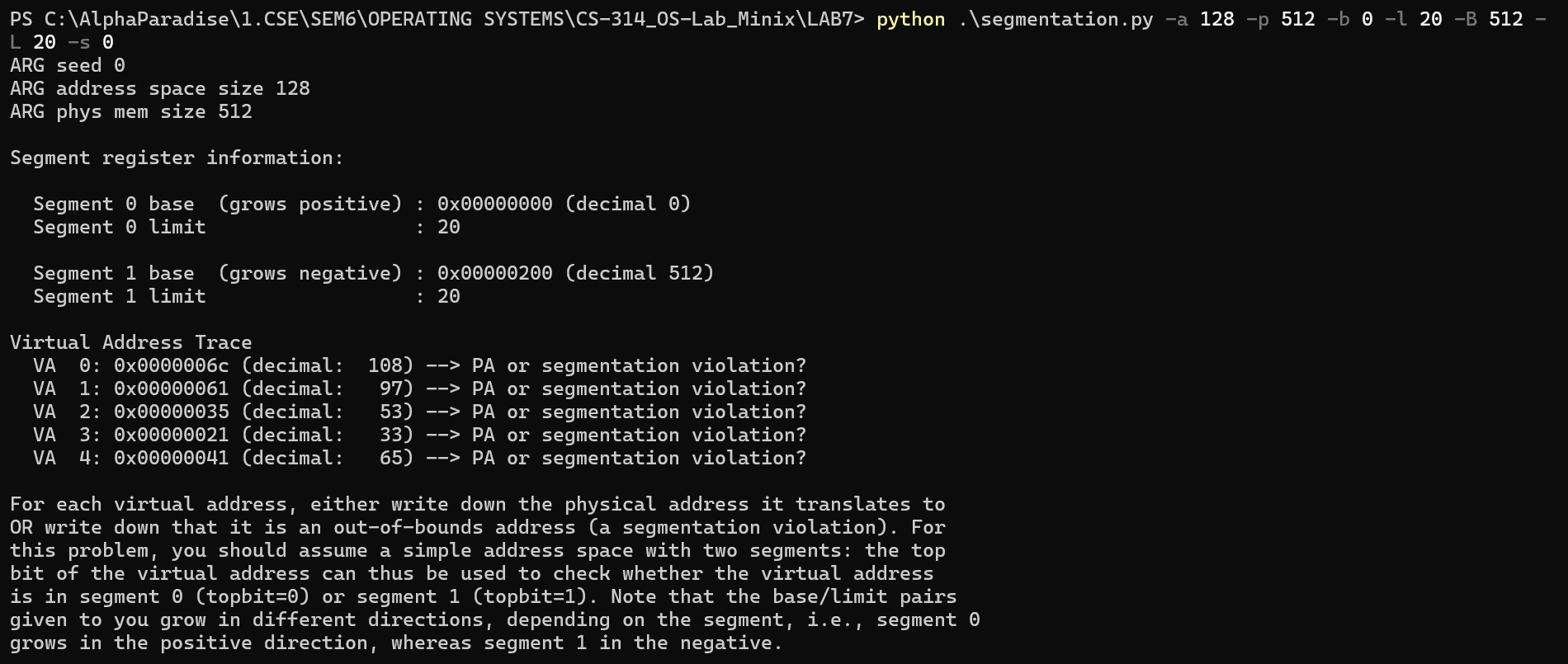
**Question – 2**

1. Ideally, all addresses up to half the value of address space size will be in segment 0 and the rest in segment 1. For segment 0, If VA > segment\_0\_limit, there will be a segmentation violation but if VA < segment\_0\_limit, then the PA mapping would be valid and PA = segment\_0\_base + VA. For segment 1, if VA < address\_space\_size – segment\_1\_limit, then there will be a segmentation violation but if VA >= address\_space\_size – segment\_1\_limit, then the corresponding PA mapping would be valid and PA = segment\_1\_base – (address\_space\_size – VA).

**For seed 0** => From the screenshot below, we can see that

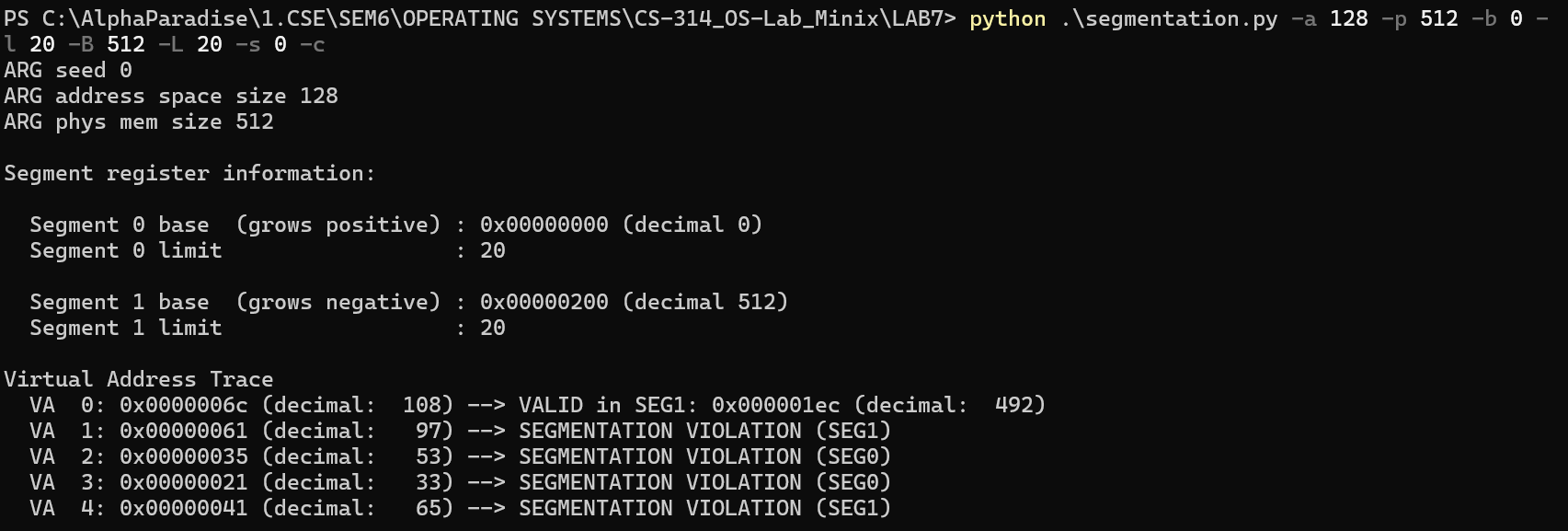
Segment 0 base: 0 and Segment 0 limit: 20 => VA 0-19 would map to 0-19 in PA

Segment 1 base: 512 and Segment 1 limit: 20 => VA 108-127 would map to 492-511 in PA



|  |  |  |
| --- | --- | --- |
| VA No. | VA | PA or Segmentation Violation |
| 0 | 108 | PA = 512 – (128-108) = 492 (0x000001ec) |
| 1 | 97 | Segmentation Violation (97 < 128-20) |
| 2 | 53 | Segmentation Violation (53 > 20) |
| 3 | 33 | Segmentation Violation (33 > 20) |
| 4 | 65 | Segmentation Violation (65 < 128-20) |

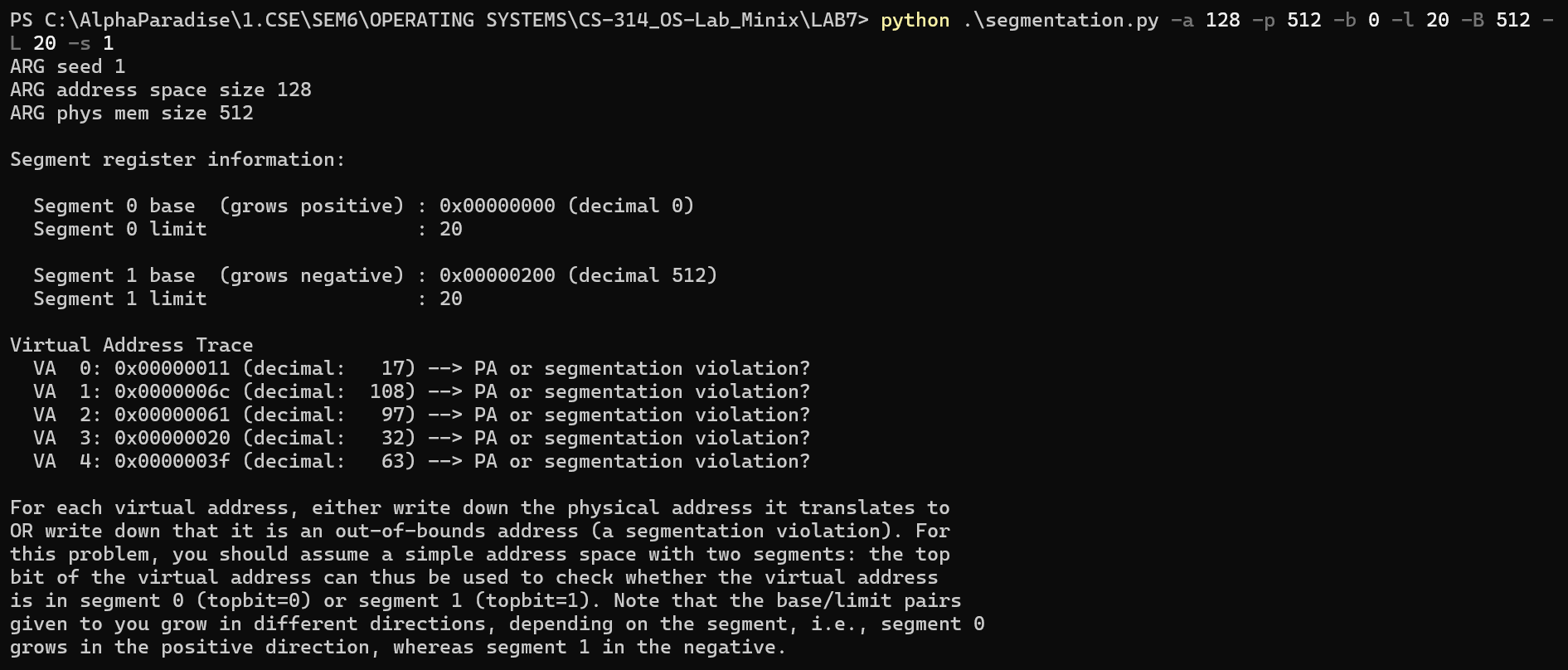
This can be verified as below:



**For seed 1** => From the screenshot below, we can see that:

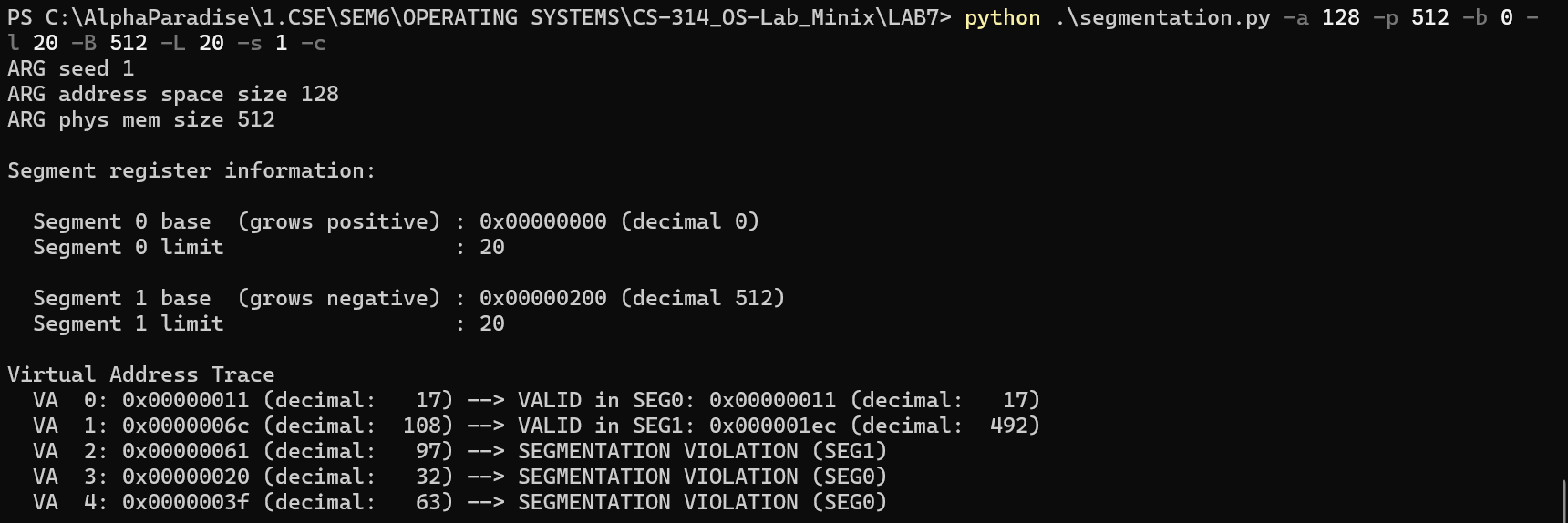
Segment 0 base: 0 and Segment 0 limit: 20 => VA 0-19 would map to 0-19 in PA

Segment 1 base: 512 and Segment 1 limit: 20 => VA 108-127 would map to 492-511 in PA



|  |  |  |
| --- | --- | --- |
| VA No. | VA | PA or Segmentation Violation |
| 0 | 17 | PA = 0 + 17 = 17 (0x00000011) |
| 1 | 108 | PA = 512 – (128-108) = 492 (0x000001ec) |
| 2 | 97 | Segmentation Violation (97 < 128 – 20) |
| 3 | 32 | Segmentation Violation (32 > 20) |
| 4 | 63 | Segmentation Violation (63 >20) |

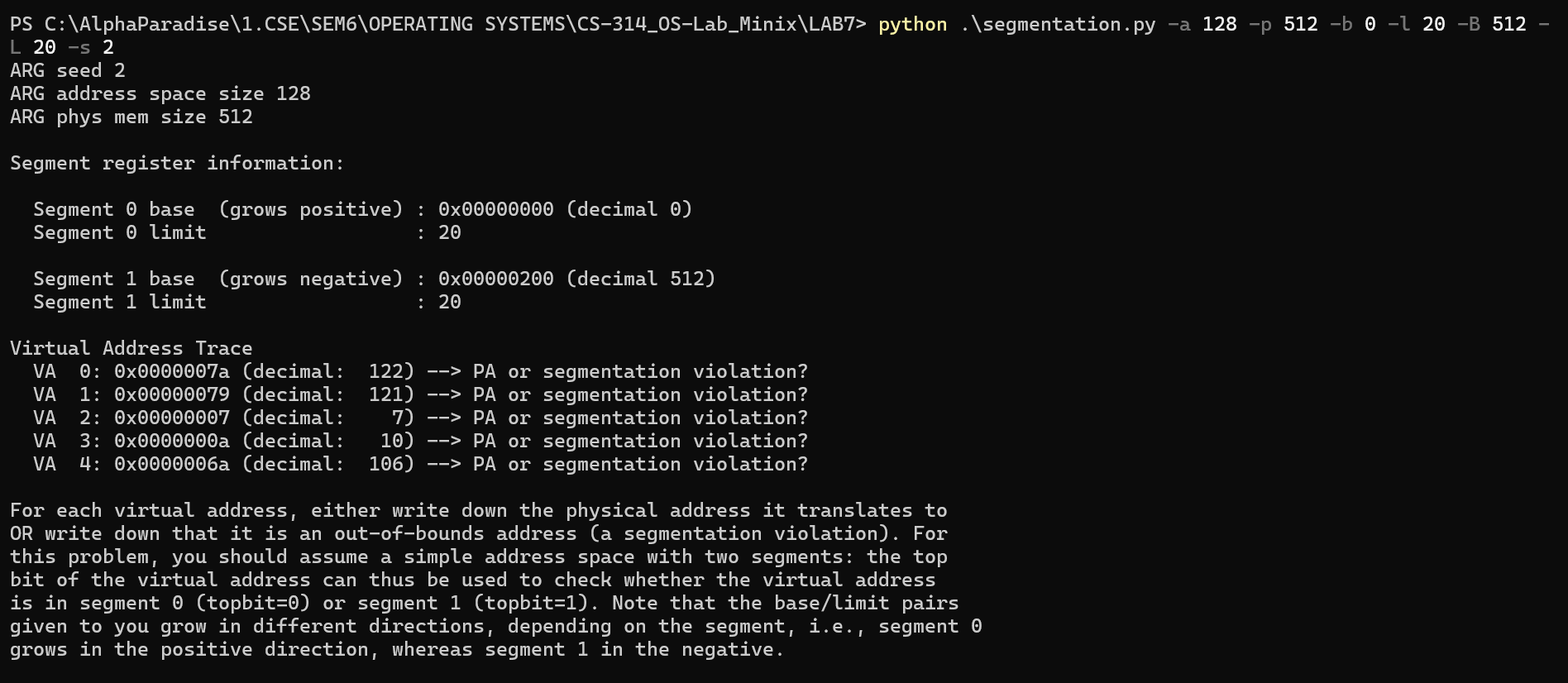
This can be verified as below:



**For seed 2**: From the screenshot below, we can see that:

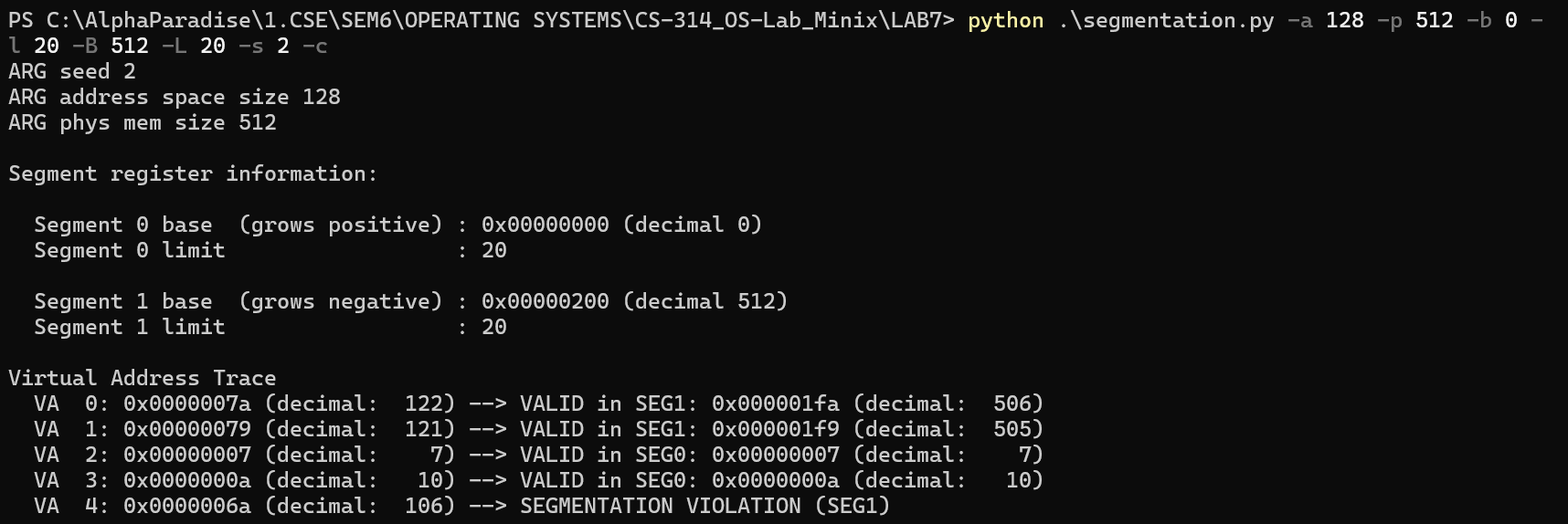
Segment 0 base: 0 and Segment 0 limit: 20 => VA 0-19 would map to 0-19 in PA

Segment 1 base: 512 and Segment 1 limit: 20 => VA 108-127 would map to 492-511 in PA



|  |  |  |
| --- | --- | --- |
| VA No. | VA | PA or Segmentation Violation |
| 0 | 122 | PA = 512 – (128-122) = 506 (0x000001fa) |
| 1 | 121 | PA = 512 – (128-121) = 505 (0x000001f9) |
| 2 | 7 | PA = 0 + 7 = 7 (0x00000007) |
| 3 | 10 | PA = 0 + 10 = 10 (0x0000000a) |
| 4 | 106 | Segmentation Violation (106 < 128 – 20) |

This can be verified as below:



1. The highest legal virtual address in segment 0 would be 19.

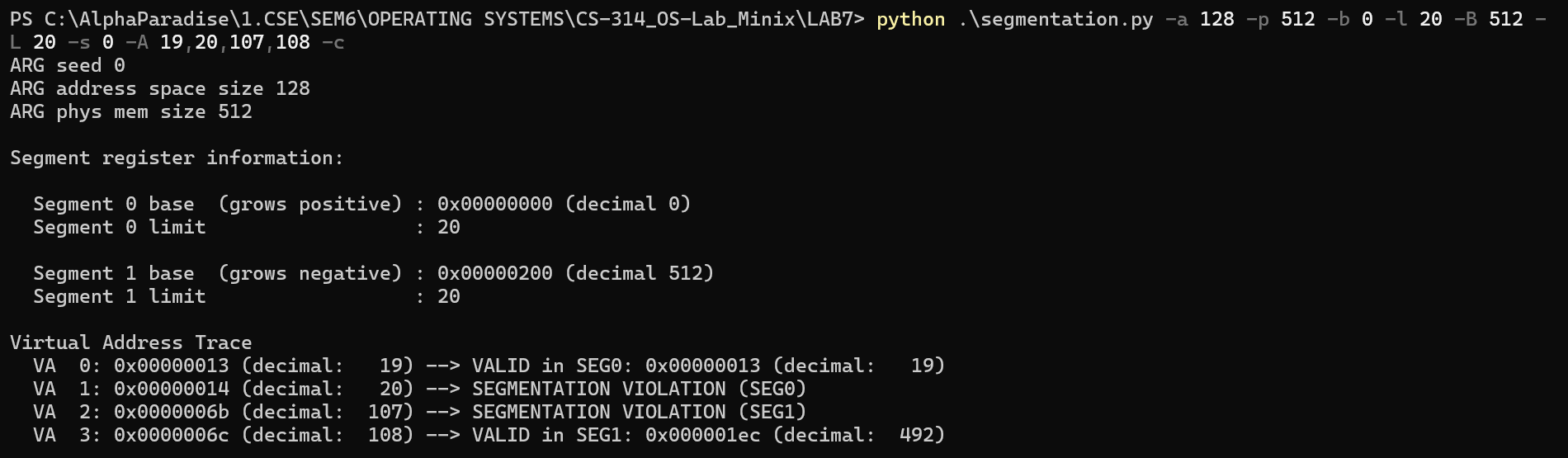
The lowest legal virtual address in segment 1 would be 108.

The highest illegal virtual address would be 107.

The lowest illegal virtual address would be 20.

To check if we are right, run the command (using the -A flag):

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



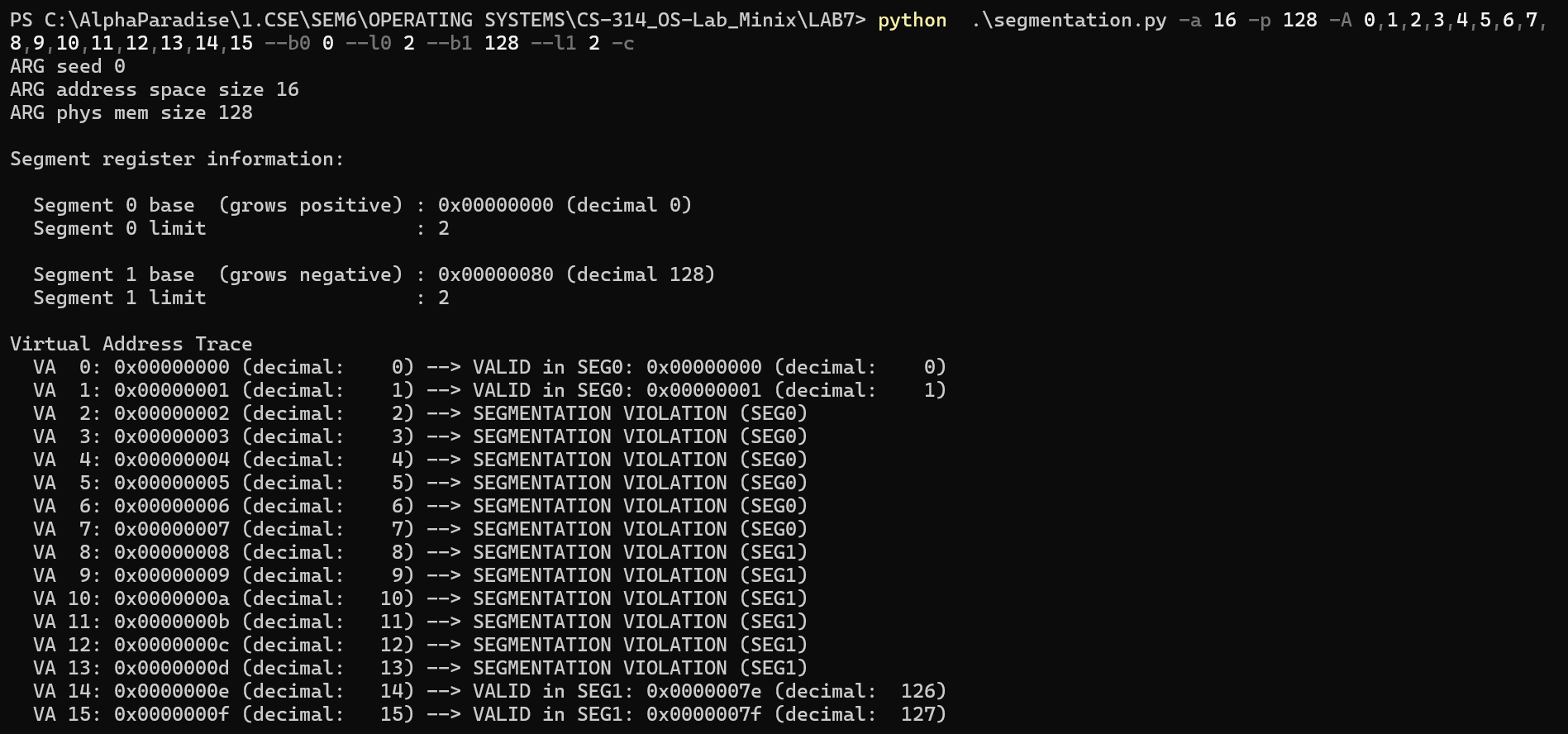
1. According to the question, we have a 16-byte address space and 128-byte physical memory and the valid virtual addresses are 0,1,14 and 15. For this to happen, we need to have:

Segment 0 base: 0, Limit: 2

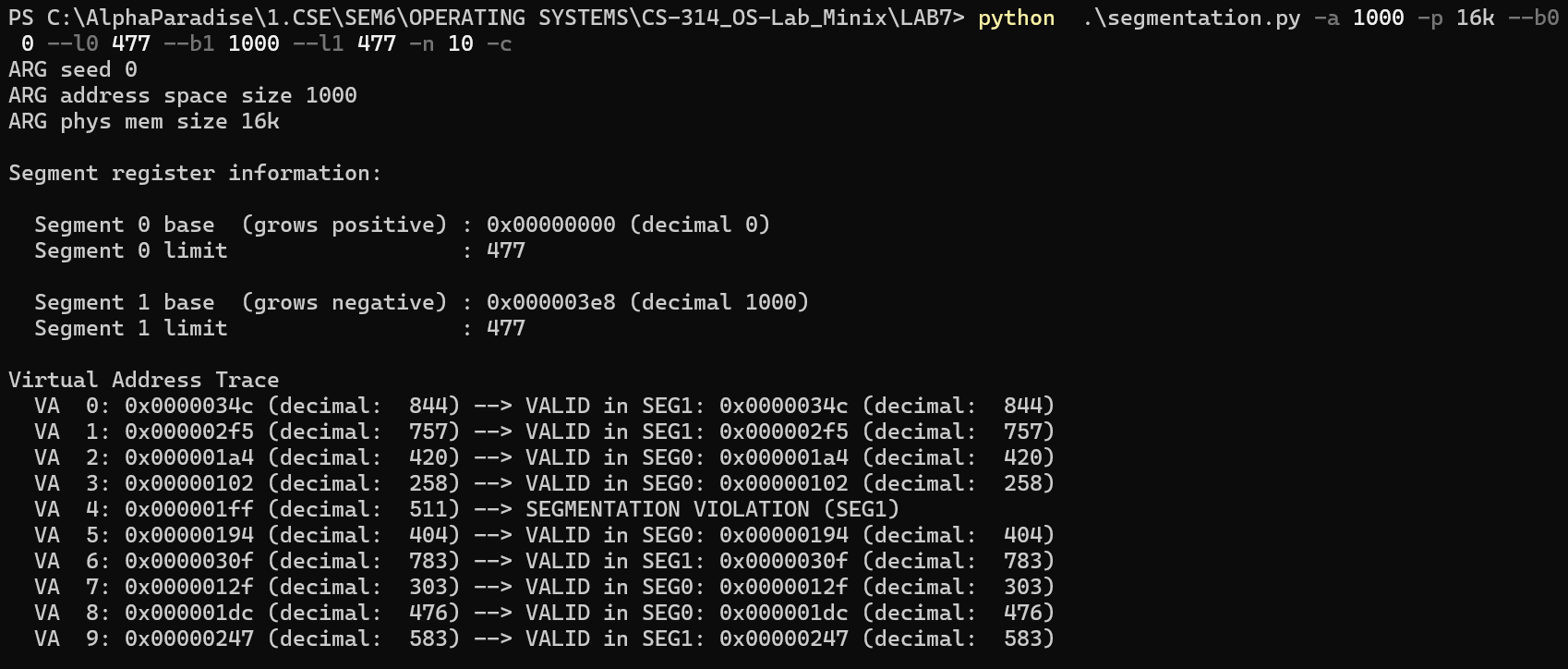
Segment 1 base: 128, Limit: 2

Therefore, **b0 would be 0, l0 would be 2, b1 would be 128 and l1 would be 2**. We can check this by running the command:

python 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 128 --l1 2 -c



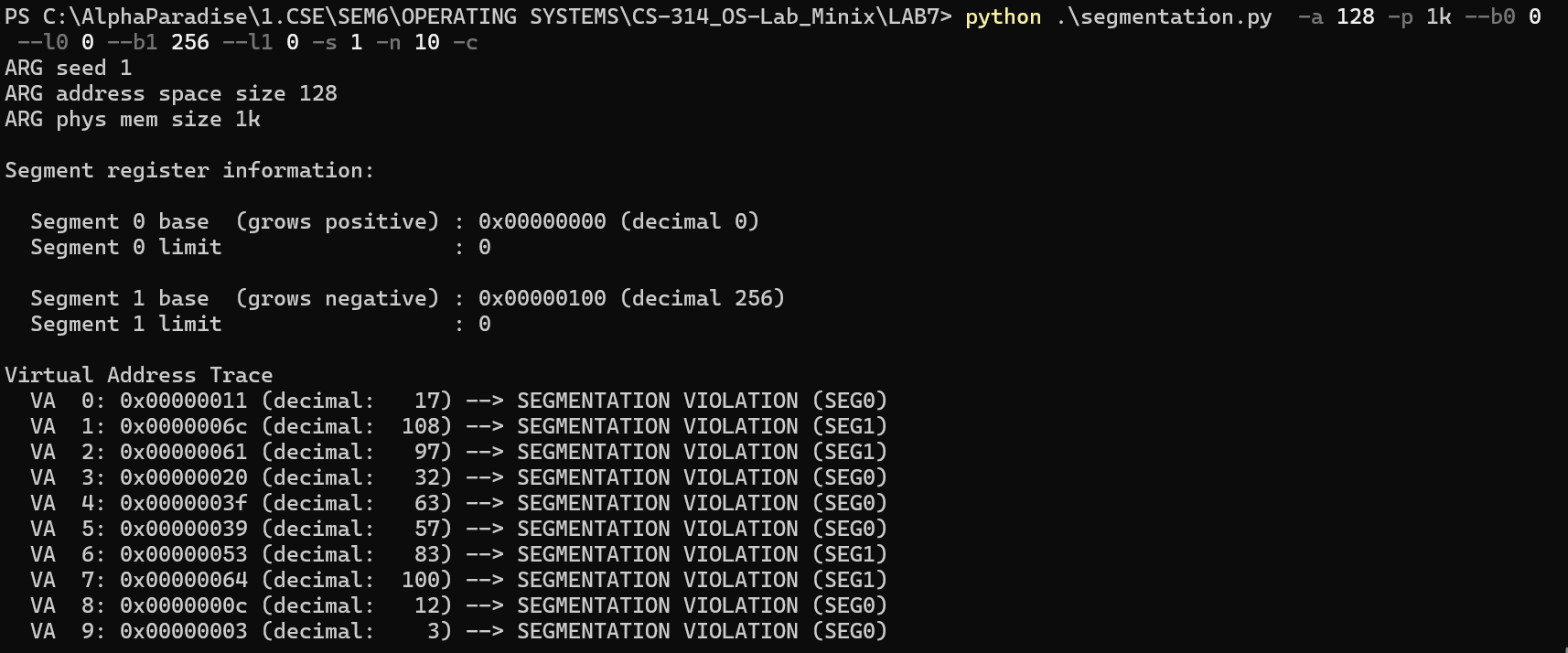
1. To configure the simulator such that 90% of the virtual addresses are valid, we can set both limits in such a way that their sum is more than 90% of the address space size. Parameters l0(segment 0 limit) and l1(segment1 limit) along with the address space size are important to get this outcome. Below’s an example:



We have an address space of 1000 bytes and both l1 and l2 are set to 477 bytes, hence giving 90% or more valid addresses.

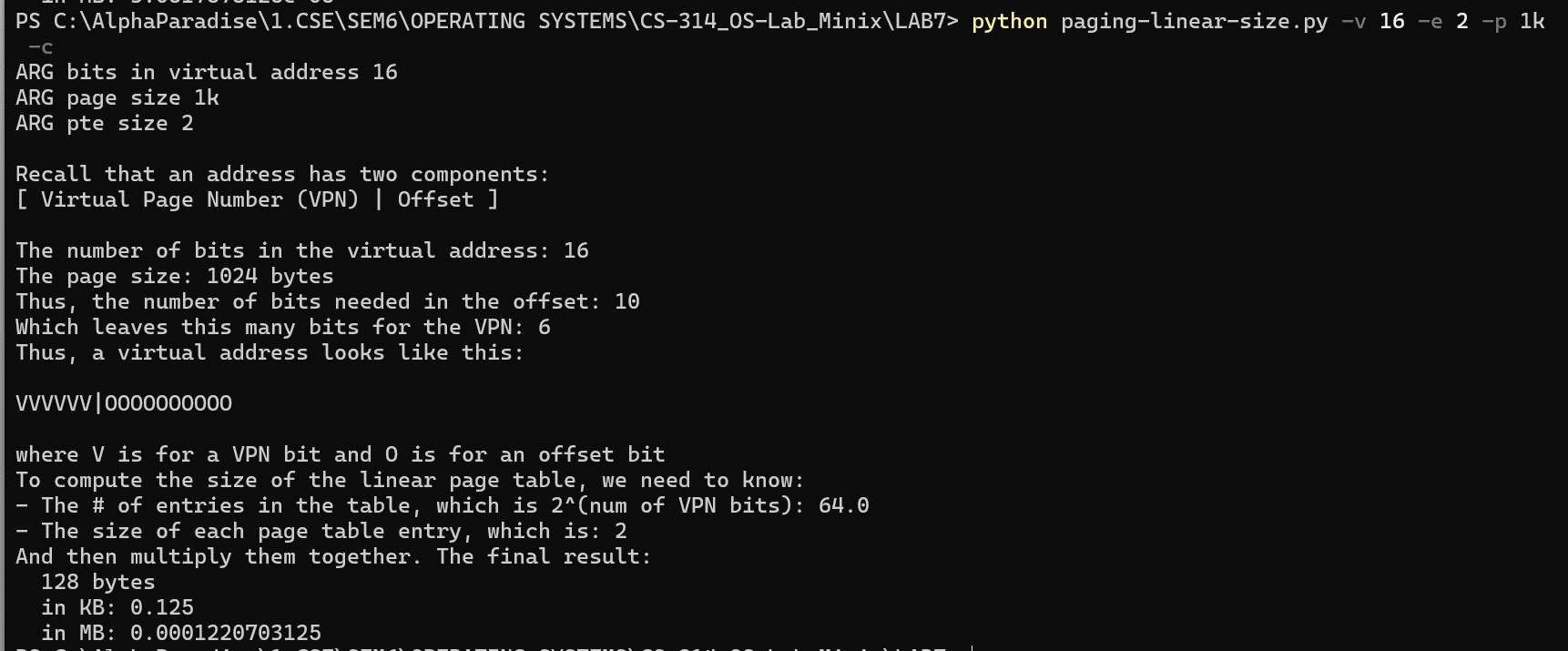
1. Yes, we can run the simulator such that no virtual addresses are valid by having the (both l0 and l1) segmentation limits equal to zero. We can test this by running the command:

python segmentation.py -a 128 -p 1k --b0 0 --l0 0 --b1 256 --l1 0 -s 1 -n 10 -c



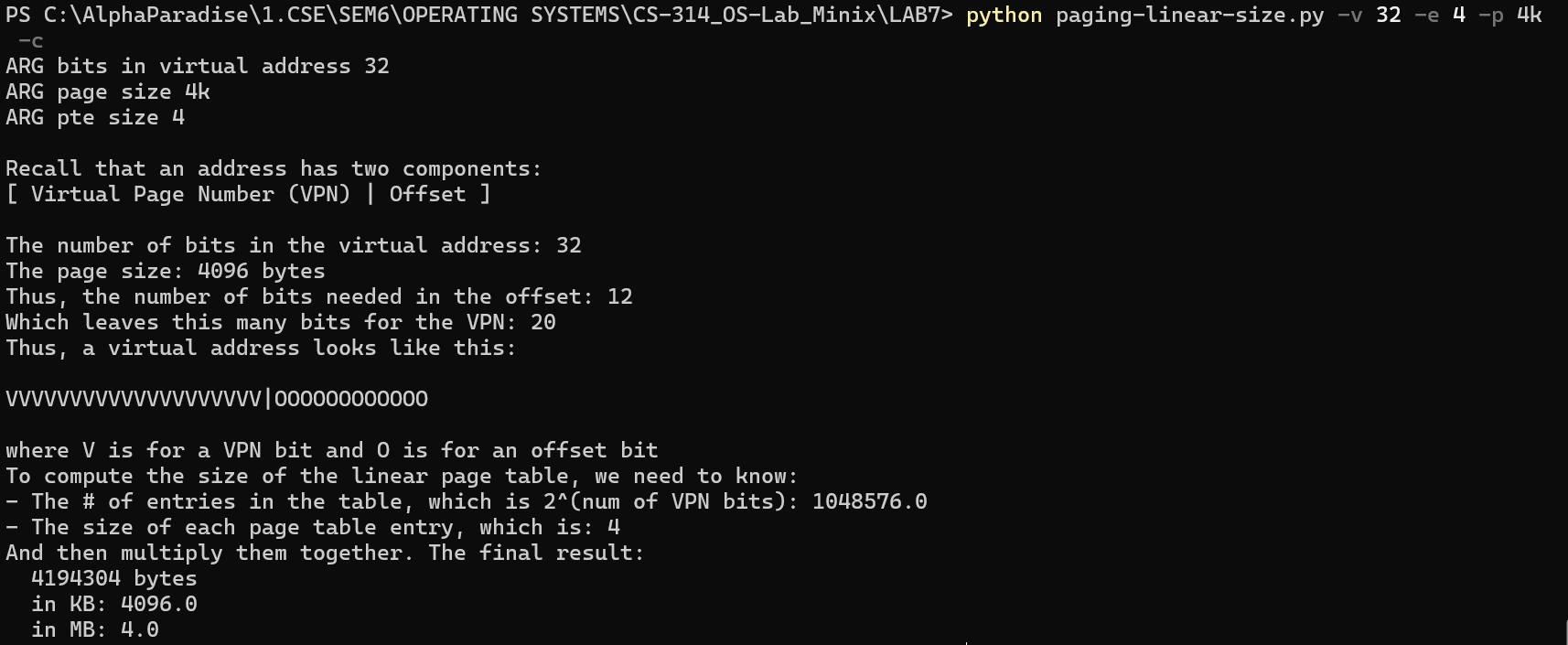
**Question-3**

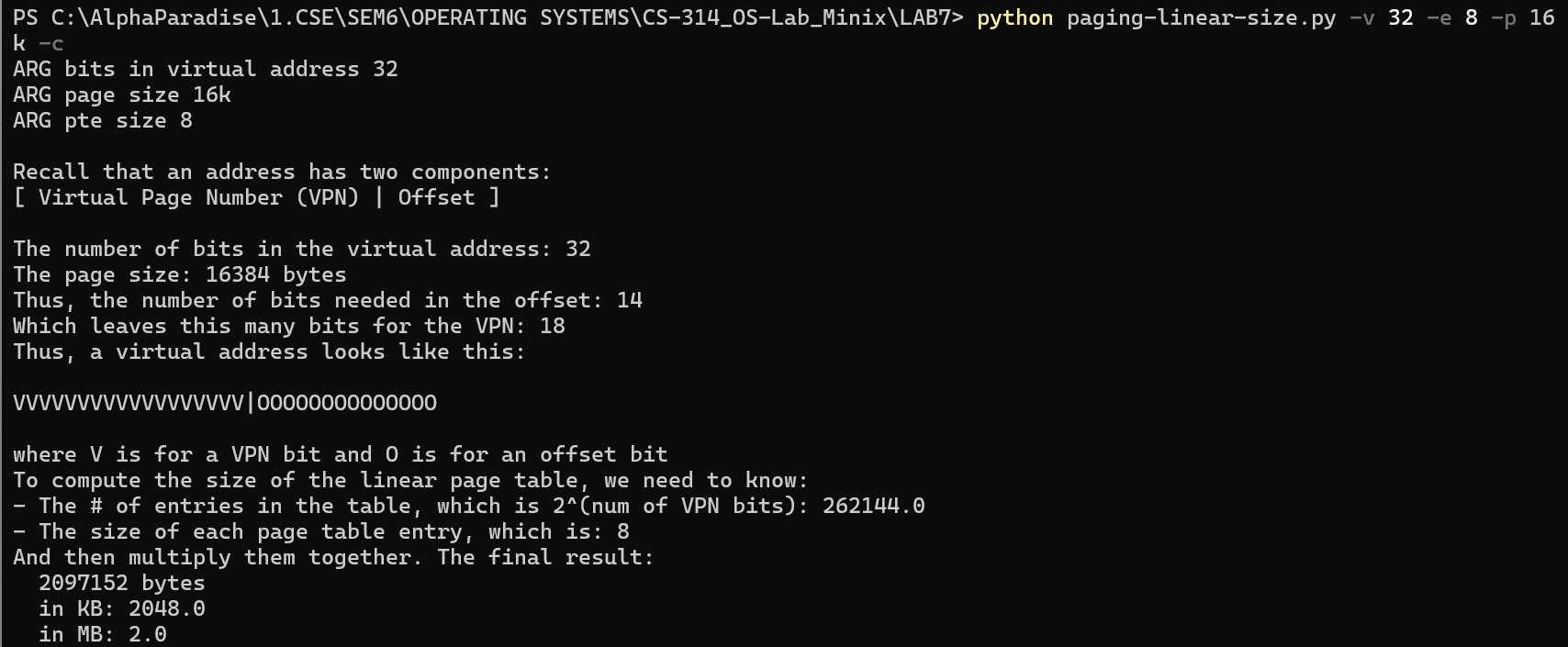
Case 1: **python paging-linear-size.py -v 16 -e 2 -p 1k** In this case, we have  
 no. of bits in virtual address = 16  
 size of page table entry = 2 bytes  
 page size = 1k = 2^10 bytes => which means we require  
 10 bits for offset.  
 size of memory = 2^16  
 no. of pages = no. of page table entries = 2^16 / 2^10 = 2^6 = 64   
 So, **size of page table = 64 \* 2 = 128 bytes**



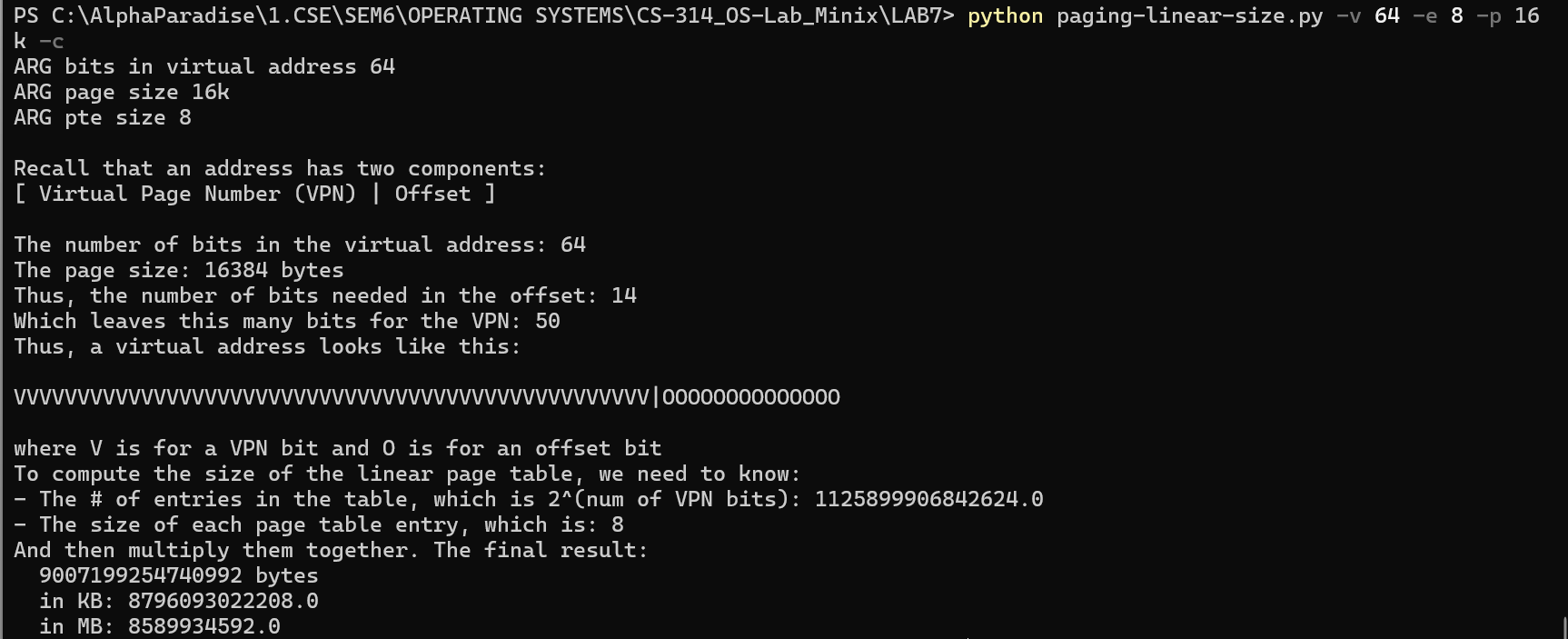
Case 2: **python paging-linear-size.py -v 16 -e 4 -p 4k** In this case, we have  
 no. of bits in virtual address = 16  
 size of page table entry = 4 bytes  
 page size = 4k = 2^12 bytes => which means we require  
 12 bits for offset.  
 size of memory = 2^16  
 no. of pages = no. of page table entries = 2^16 / 2^12 = 2^4 = 16   
 So, **size of page table = 16 \* 4 = 64 bytes**  
  


Case 3: **python paging-linear-size.py -v 32 -e 4 -p 4k** [default case]  
 In this case, we have  
 no of bits in virtual address = 32  
 size of page table entry = 4 bytes  
 page size = 4k = 4096 bytes = 2^12 => which means we require 12  
 bits for offset.  
 size of physical memory = 2^32  
 no. of pages = no. of page table entries = 2^32 / 2^12 = 2^20  
 So, **size of page table = 2^20 \* 4 = 2^22 = 4194304 bytes = 4MB**



Case 4: **python paging-linear-size.py -v 32 -e 8 -p 16k** In this case, we have  
 no of bits in virtual address = 32  
 size of page table entry = 8 bytes  
 page size = 16k = 2^14 bytes => which means we require  
 14 bits for offset.  
 size of memory = 2^32  
 no. of pages = no. of page table entries = 2^32 / 2^14 = 2^18  
 So, **size of page table** = 2^18 \* 8 = 2^21 = **2097152 bytes = 2 MB**  
 

Case 5: **python paging-linear-size.py -v 64 -e 8 -p 16k** In this case, we have  
 no of bits in virtual address = 64  
 size of page table entry = 8 bytes  
 page size = 16k = 2^14 bytes => which means we require  
 14 bits for offset.  
 size of memory = 2^64  
 no. of pages = no. of page table entries = 2^64 / 2^14 = 2^50  
 So, **size of page table** = 2^50 \* 8 = 2^53 = **9007199254740992 bytes = 8589934592 MB**



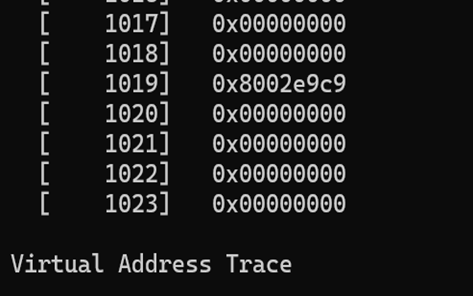
From all the above cases we can conclude that the page table size is directly proportional to the size of the page table entry and indirectly proportional to the page size. It is exponentially proportional to the no. of bits in the virtual address, i.e., directly proportional to 2 ^ (no. of bits).

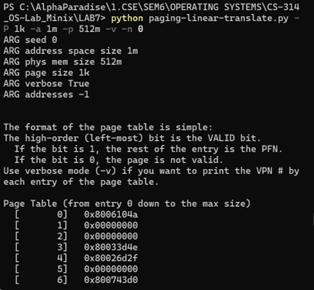
**Question-4**

1. **Before translation:**Using default seed 0
2. Increasing address space size:

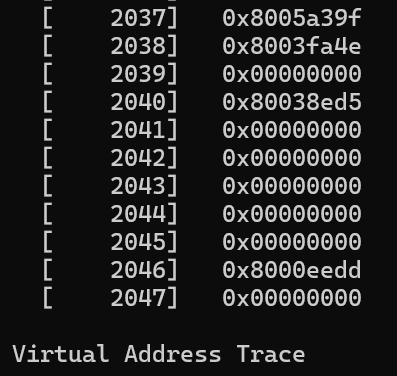
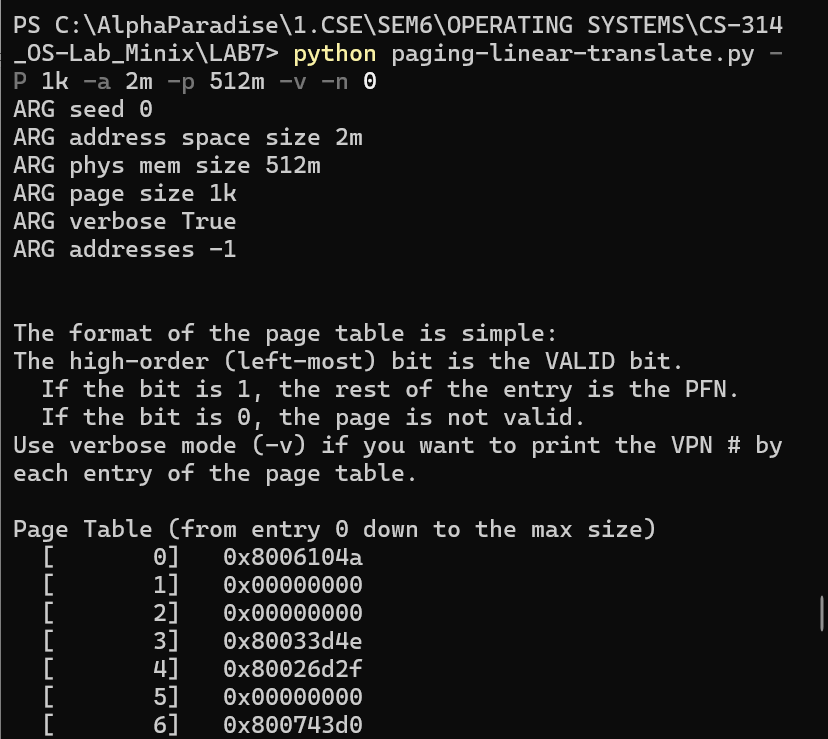
python paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0

> page size = 1k = 2^10 bytes

> address space size = 1m = 2^20 bytes  
> no. of pages = **no. of page table entries** = 2^20 / 2^10 = 2^10 = **1024**, i.e., from 0 to 1023  
> size of page table entry = 4  
> **size of page table** = 2^10\*4 = 2^12 = 4096 bytes = **4KB**



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

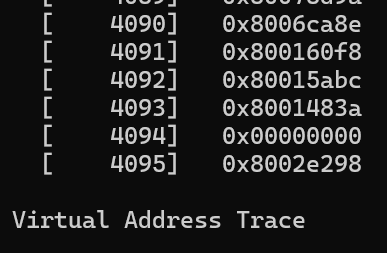
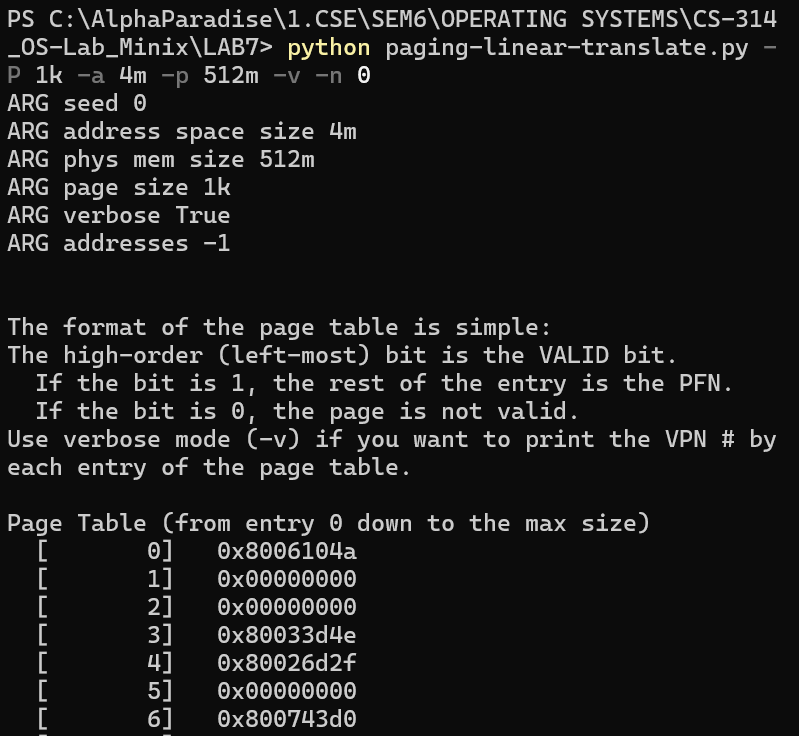
> page size = 1k = 2^10 bytes

> address space size = 2m = 2\* 2^20 bytes = 2^21 bytes  
> no. of pages = **no. of page table entries** = 2^21 / 2^10 = 2^11 = **2048**, i.e., from 0 to 2047  
> size of page table entry = 4  
> **size of page table** = 2^11\*4 = 2^13 = 8192 bytes = **8KB**

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

> page size = 1k = 2^10 bytes

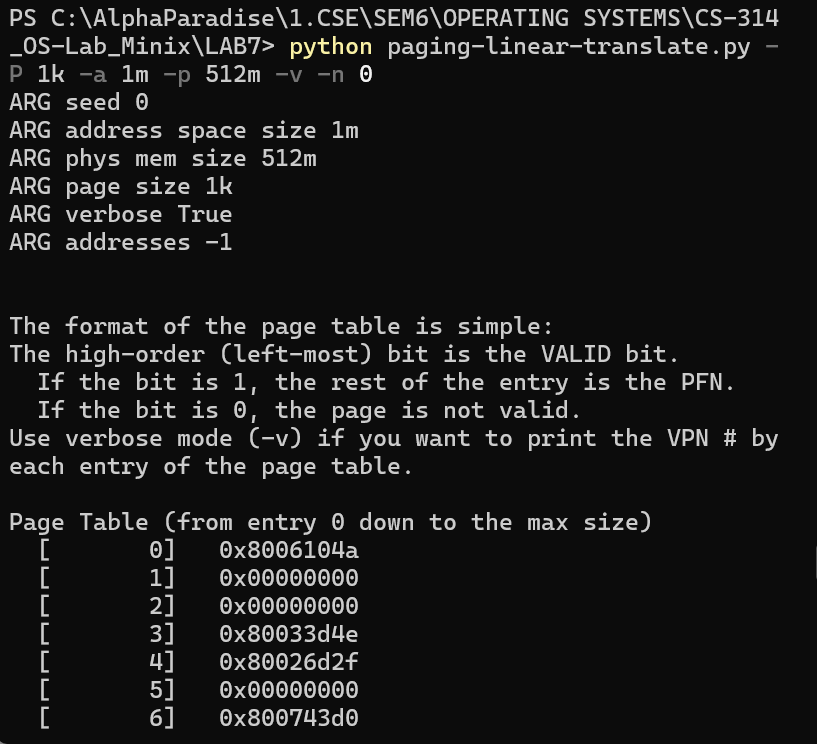
> address space size = 4m = 4\* 2^20 bytes = 2^22 bytes  
> no. of pages = **no. of page table entries** = 2^22 / 2^10 = 2^12 = **4096**, i.e., from 0 to 4095  
> size of page table entry = 4  
> **size of page table** = 2^12\*4 = 2^14 = 16384 bytes = **16KB**



Thus, we can conclude that the size of the page table is directly proportional to the address space size because, when the address space size doubles, the size of the page table also doubles. each entry in the page table corresponds to a portion of the address space. Therefore, as the address space size increases, more entries are needed in the page table to map each virtual address to its corresponding physical address.

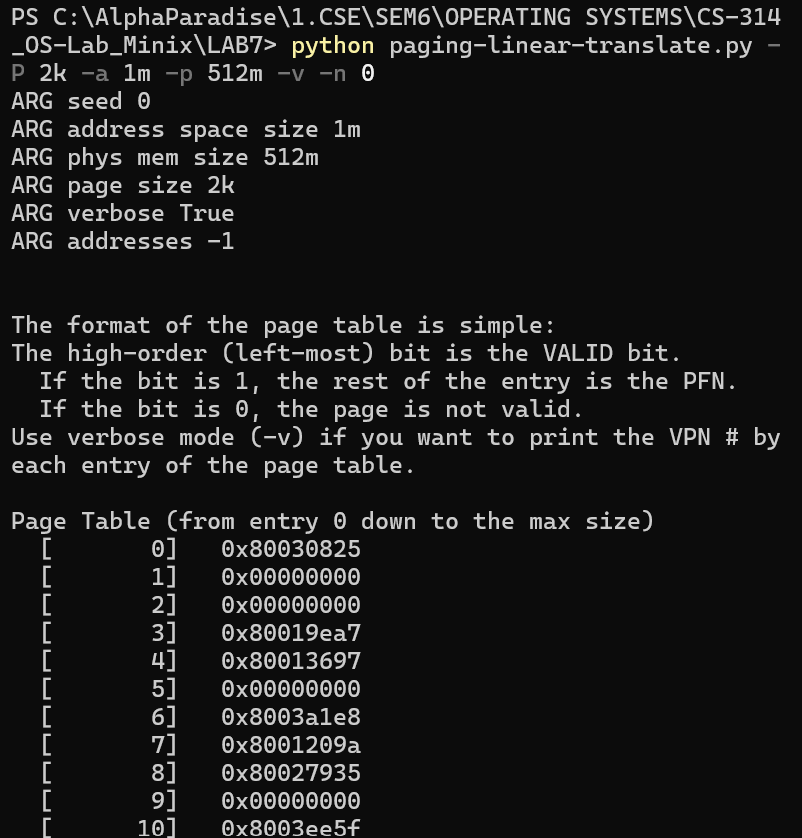
1. Increasing page size

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

> page size = 1k = 2^10 bytes  
> address space size = 1m = 2^20 bytes  
> no. of pages = **no. of page table entries** = 2^20 / 2^10 = 2^10 = **1024**, i.e., from 0 to 1023  
> size of page table entry = 4  
> **size of page table** = 2^10\*4 = 2^12 = 4096 bytes = **4KB**  
  


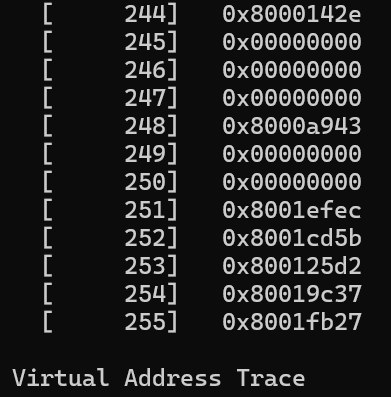
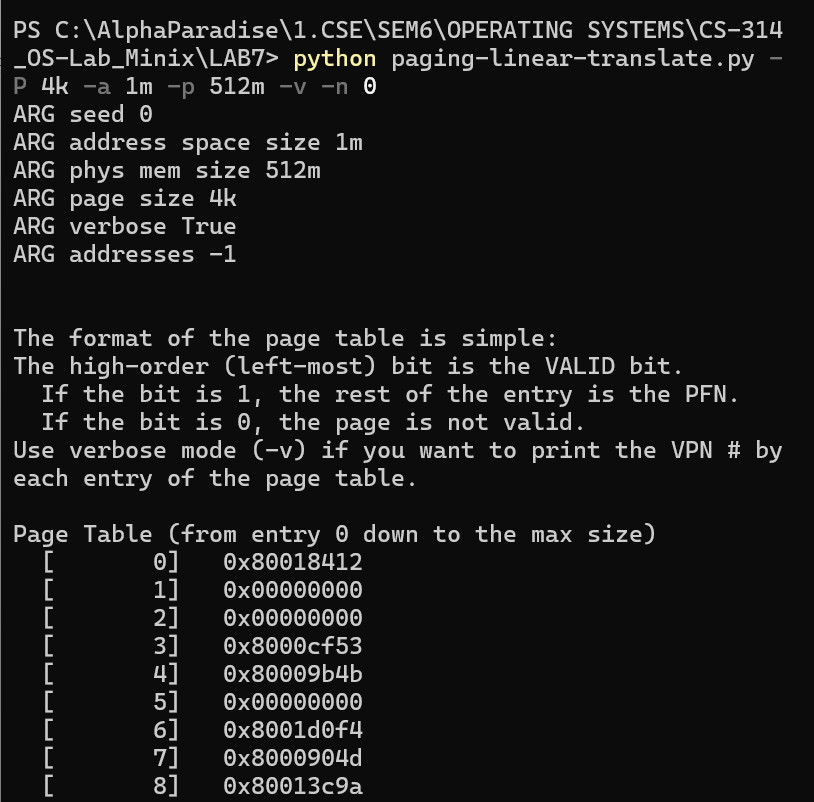
python paging-linear-translate.py -P 2k -a 1m -p 512m -v -n 0

> page size = 2k = 2^11 bytes  
> address space size = 1m = 2^20 bytes  
> no. of pages = **no. of page table entries** = 2^20 / 2^11 = 2^9 = **512**, i.e., from 0 to 511  
> size of page table entry = 4  
> **size of page table** = 2^9\*4 = 2^11 = 2048 bytes = **2KB**



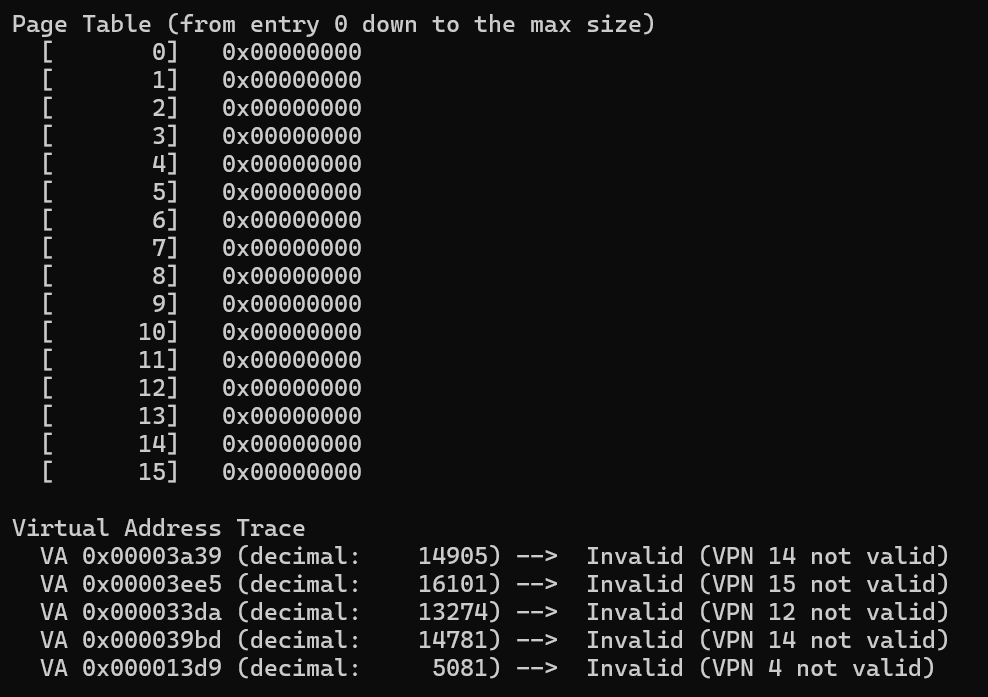
python paging-linear-translate.py -P 4k -a 1m -p 512m -v -n 0

> page size = 4k = 2^12 bytes  
> address space size = 1m = 2^20 bytes  
> no. of pages = **no. of page table entries** = 2^20 / 2^12 = 2^8 = **256**, i.e., from 0 to 255  
> size of page table entry = 4  
> **size of page table** = 2^8\*4 = 2^10 = 1024 bytes = **1KB**



Thus, we can conclude that the size of the page table is inversely proportional to the page size because, when the page size doubles, the size of the page table becomes half. When the page size increases, fewer pages are needed to cover the entire address space. This means that each entry in the page table covers a larger range of addresses. Consequently, the page table can have fewer entries since each entry represents a larger chunk of the address space.  
 We should not use big pages in general because it can lead to internal fragmentation, i.e., the entire big page is always allocated but only a fraction of it is used, while a lot of unused memory space remains. With larger page sizes, the TLB can hold fewer entries due to the larger amount of memory each translation occupies. This increases the likelihood of TLB misses, leading to slower memory access times. If the working set size of a process is smaller than the page size, using large pages may lead to unnecessary memory overhead.

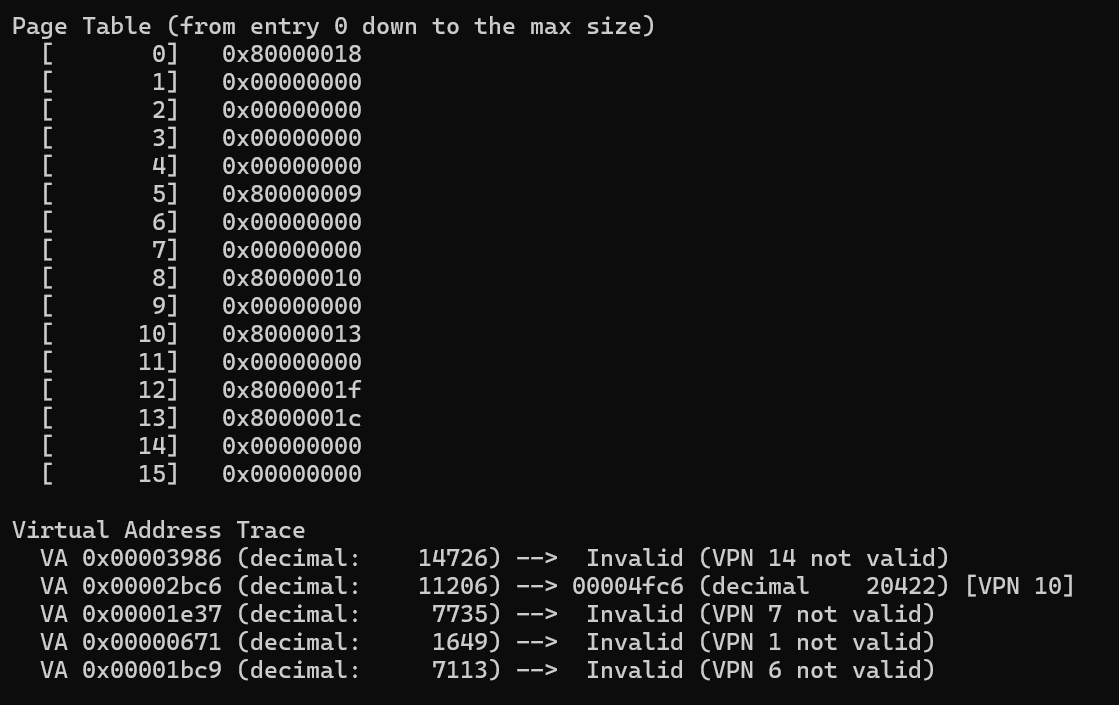
1. Since the address space size is 16k or 2^14 bytes, the required no. of bits for the virtual address is 14 and the page size is 1k or 2^10, so the no. of bits required for offset is 10. Hence, no of bits for VPN will be 14 – 10 = 4 bits. No. of pages will be 2^14 / 2^10 = 2^4 or 16.  
   **As we increase the percentage of virtual address space used (percentage of pages allocated), more pages are allocated, i.e., no. of page table entries increase. Hence, no. of page hits would increase over page faults, i.e., more no. of memory accesses (VA) become valid, as seen in the screenshots below.**  
   **As more of the address space is utilized by processes, there is a higher demand for memory resources to store the data and instructions needed by those processes. This leads to more frequent access to virtual memory addresses to fetch or store data.**  
   python paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 0



Here, we can see that all addresses are deemed invalid since no pages are allocated.

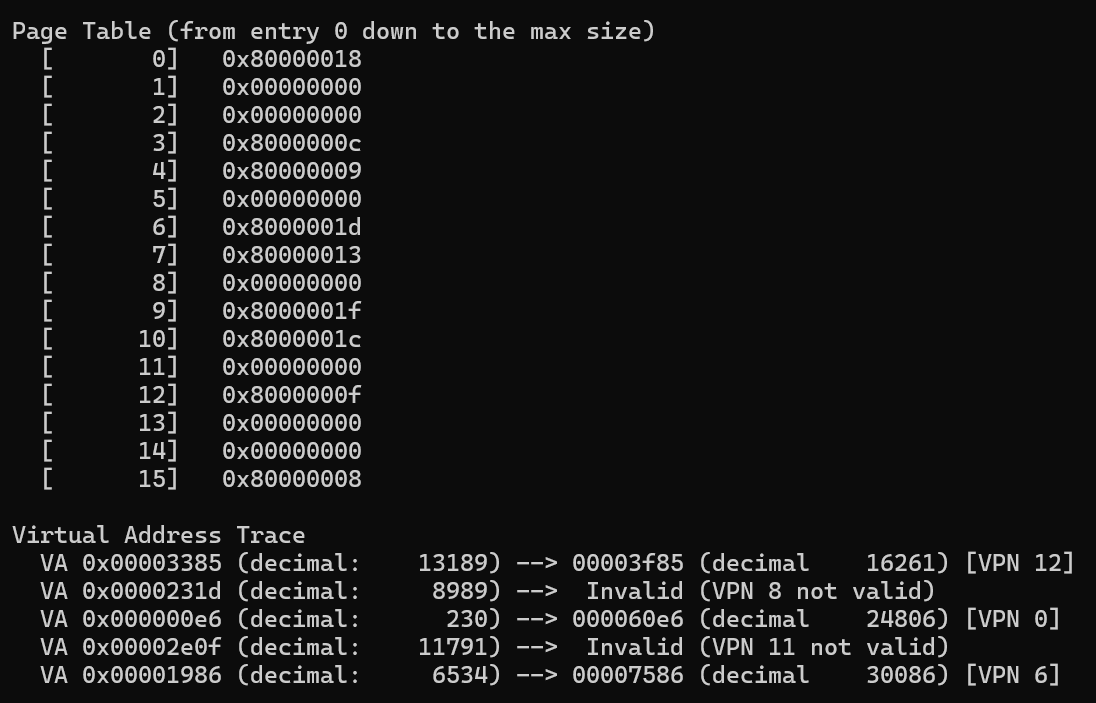
python paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 25

We can see 6 entries are filled in the page table.



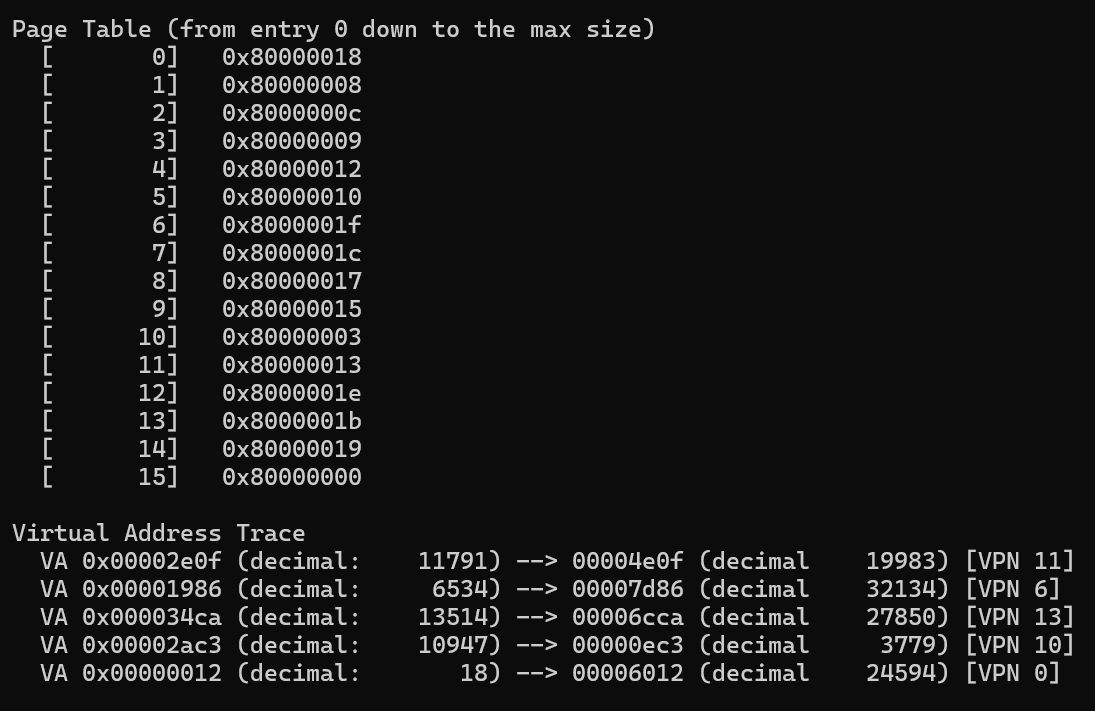
python paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 50

We can see 9 entries are filled in the page table.



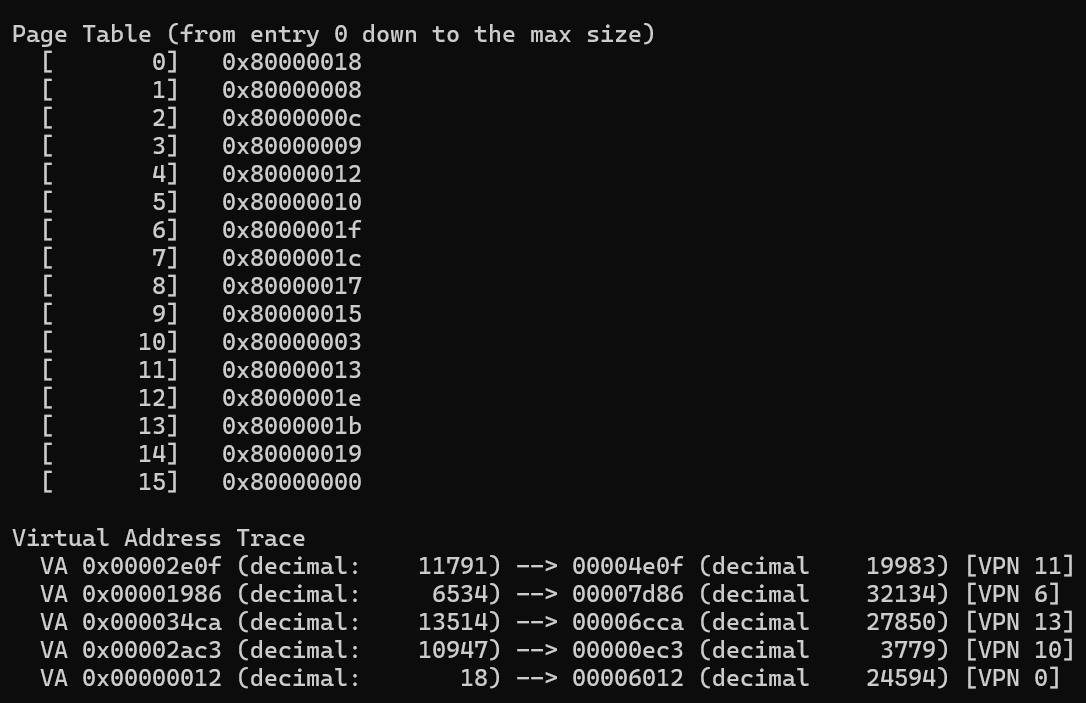
python paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 75

We can see all the entries are filled in the page table.

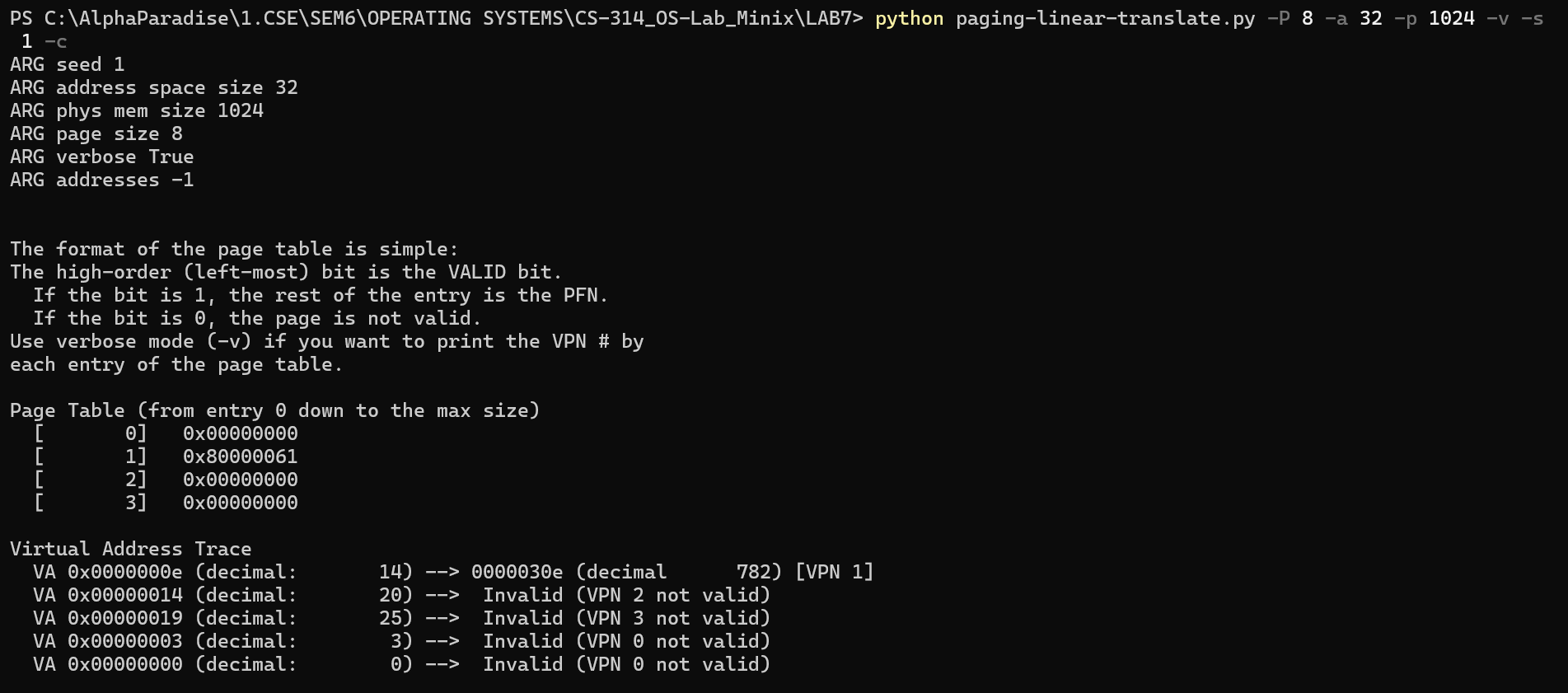


python paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 100

We can see all the entries are filled in the page table.

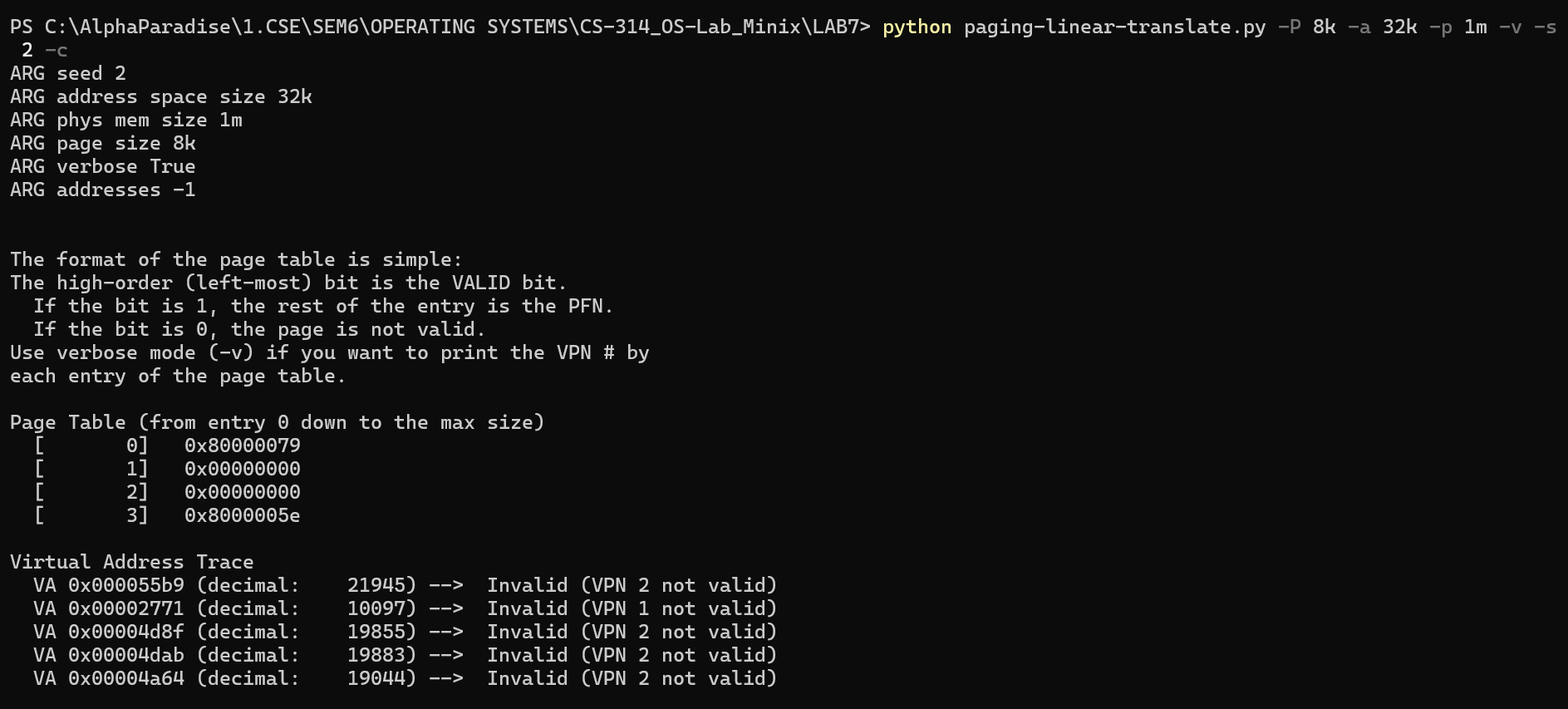


1. 1) python paging-linear-translate.py -P 8 -a 32 -p 1024 -v -s 1



This Parameter combination seems unrealistic because the address space and physical memory size are too small to be practically used. Also, the page size of 8 bytes is too small and we can accommodate only a maximum of 4 pages, which won’t be efficient, making it unrealistic.

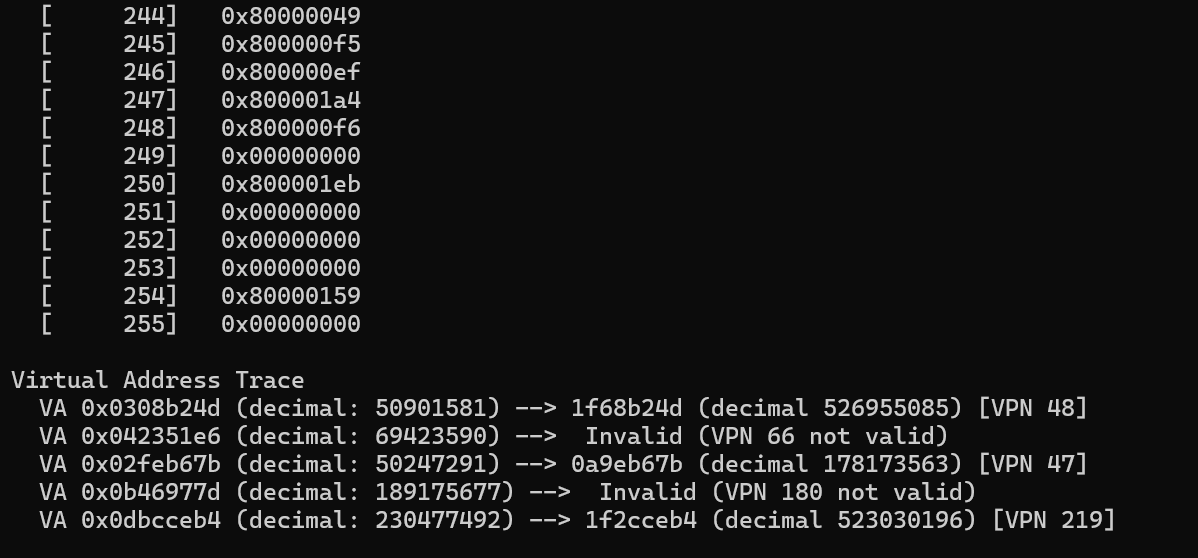
2) python paging-linear-translate.py -P 8k -a 32k -p 1m -v -s 2



This case also seems a little unrealistic. Though the page size is realistic (8 KB), with an address space of 32 KB, we can accommodate only 4 pages. This case (case 2) is better compared to case 1 in terms of the address space, physical memory and pages being large enough in size. But, even then we accommodate only a maximum of 4 pages, thus making it a little unrealistic.

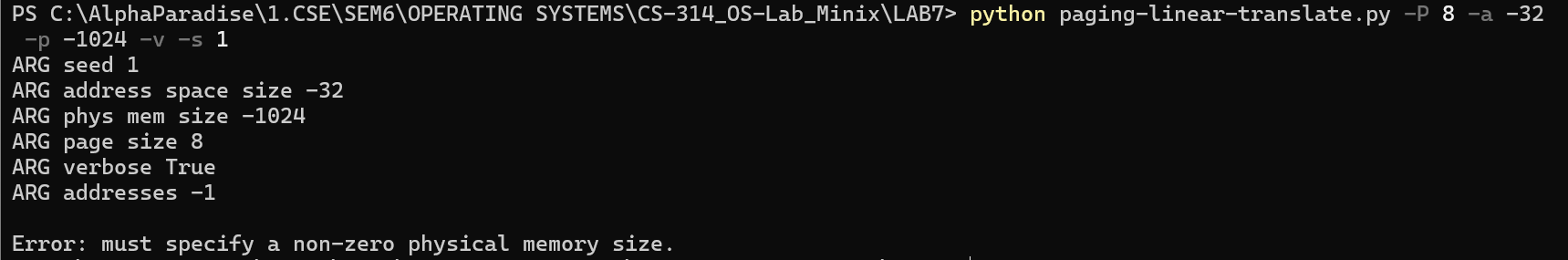
3) python paging-linear-translate.py -P 1m -a 256m -p 512m -v -s 3 -c

This case is also unrealistic because the page size (1 MB) is too much as most of the page would be unused or empty and can also lead to internal fragmentation. Also, having large pages eats up memory and thus reducing the memory available for other processes running.

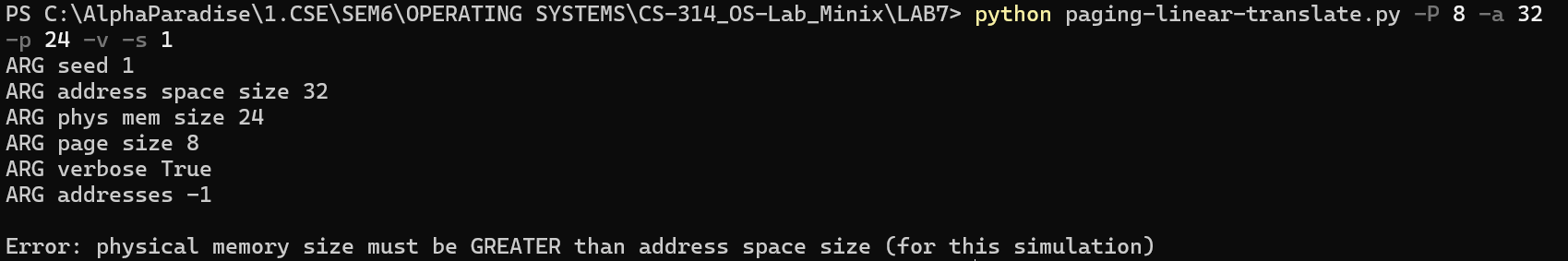


Thus, overall case 3 seems the most unrealistic combination followed by case 1. Though case 2 seems reasonable, still it is a little unrealistic. (**Unrealistic-ness order: 3>>1>>2**).

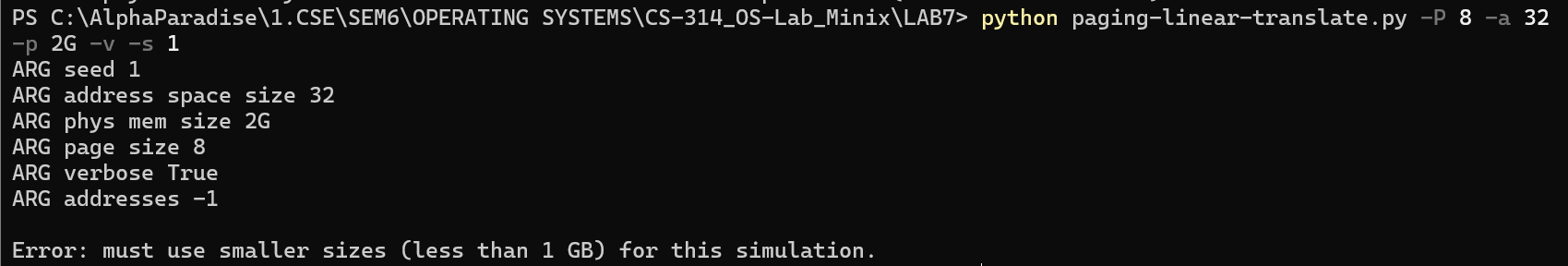
1. The program code has some requirements like the physical memory size being greater than the address space size for the program to work correctly as expected. Some of the requirements/limitations of the program are:
2. All the values such as address space size, physical memory size, limit etc., should be positive.



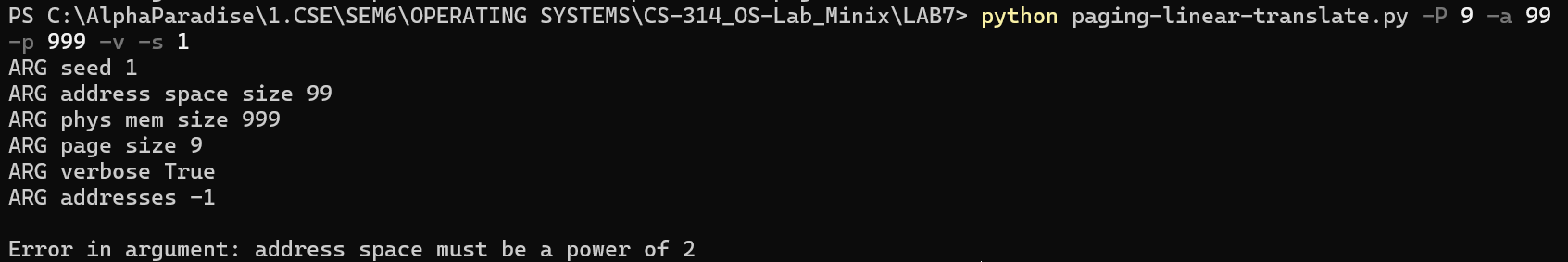
1. Physical memory size should be greater than or equal to the address space size. Otherwise, it would not be possible to accommodate all available addresses.



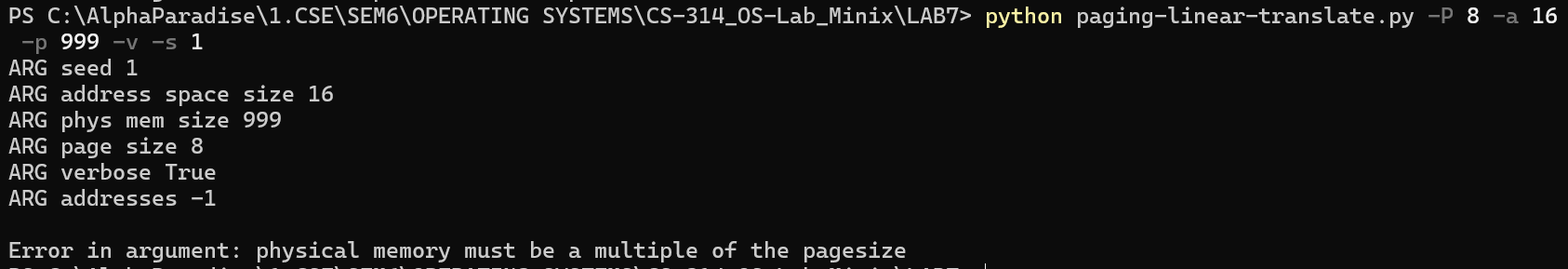
1. Both Physical memory size and address space size should be less than 1 GB.



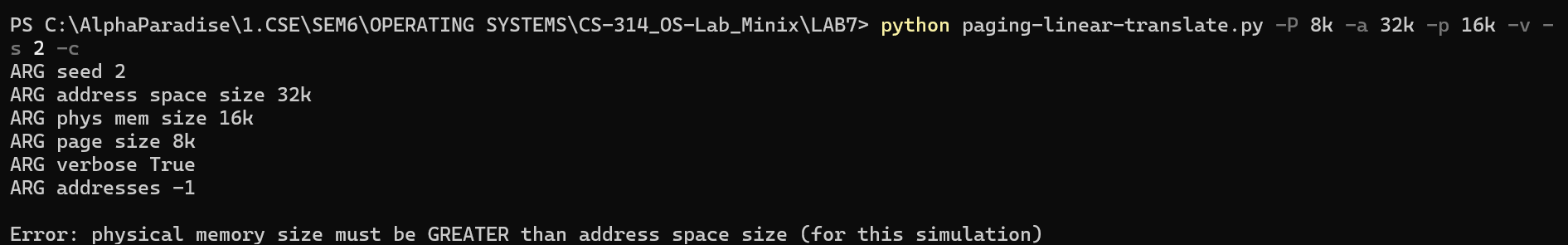
1. The address space size and physical memory size must be in powers of 2 so that there won’t be any discontinuities in memory addressing.



1. The page size must also be in powers of 2 as both the physical memory size and address space size must be multiples of page size so that there would be an integer no. of pages.



If the address space size is more than the physical memory size, the program returns an error:



This is because, if the address space size is more than the physical memory size, we cannot fit/accommodate all the available addresses in the address space in the physical memory. The mapping from VA to PA would be erroneous.