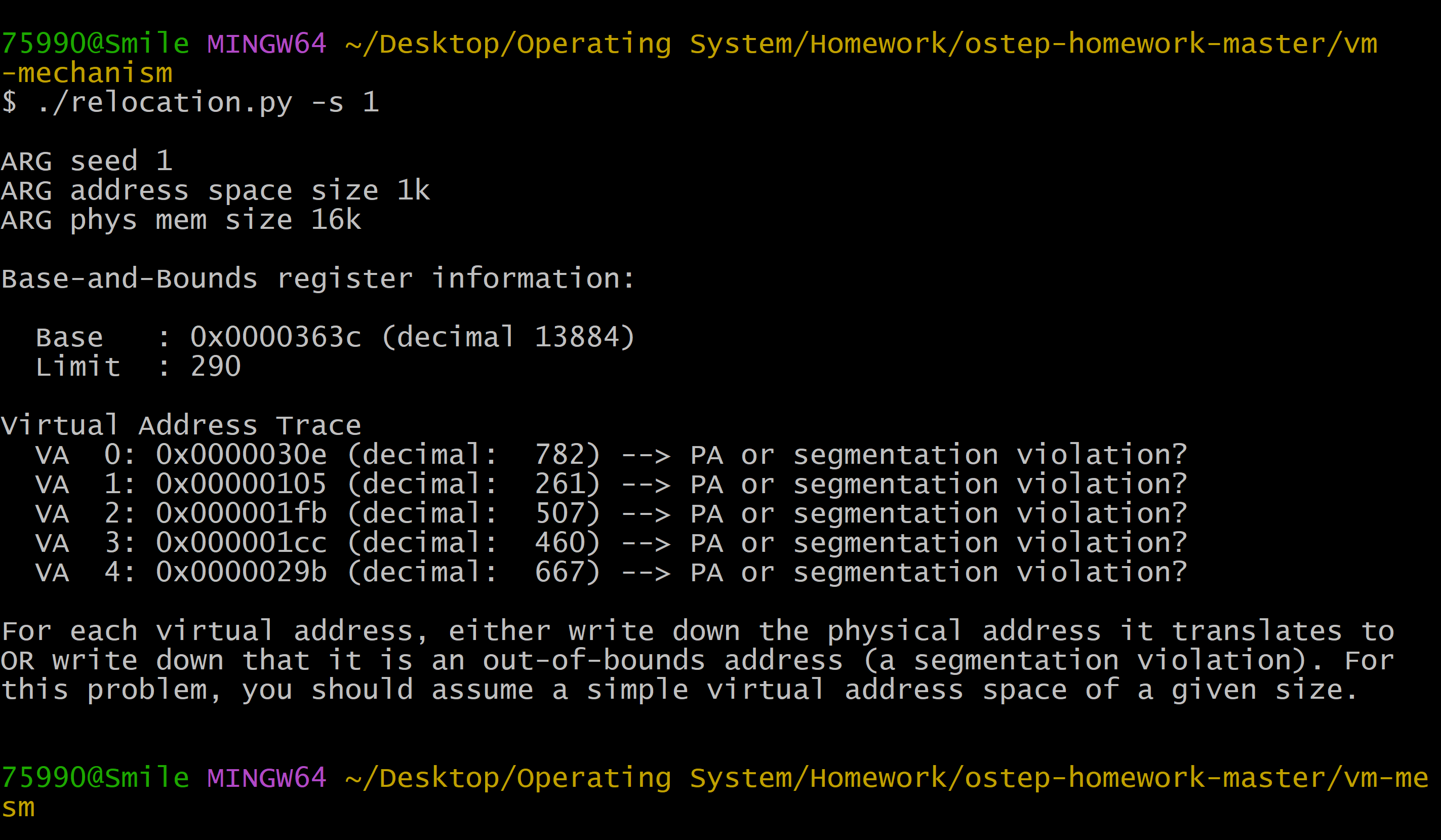
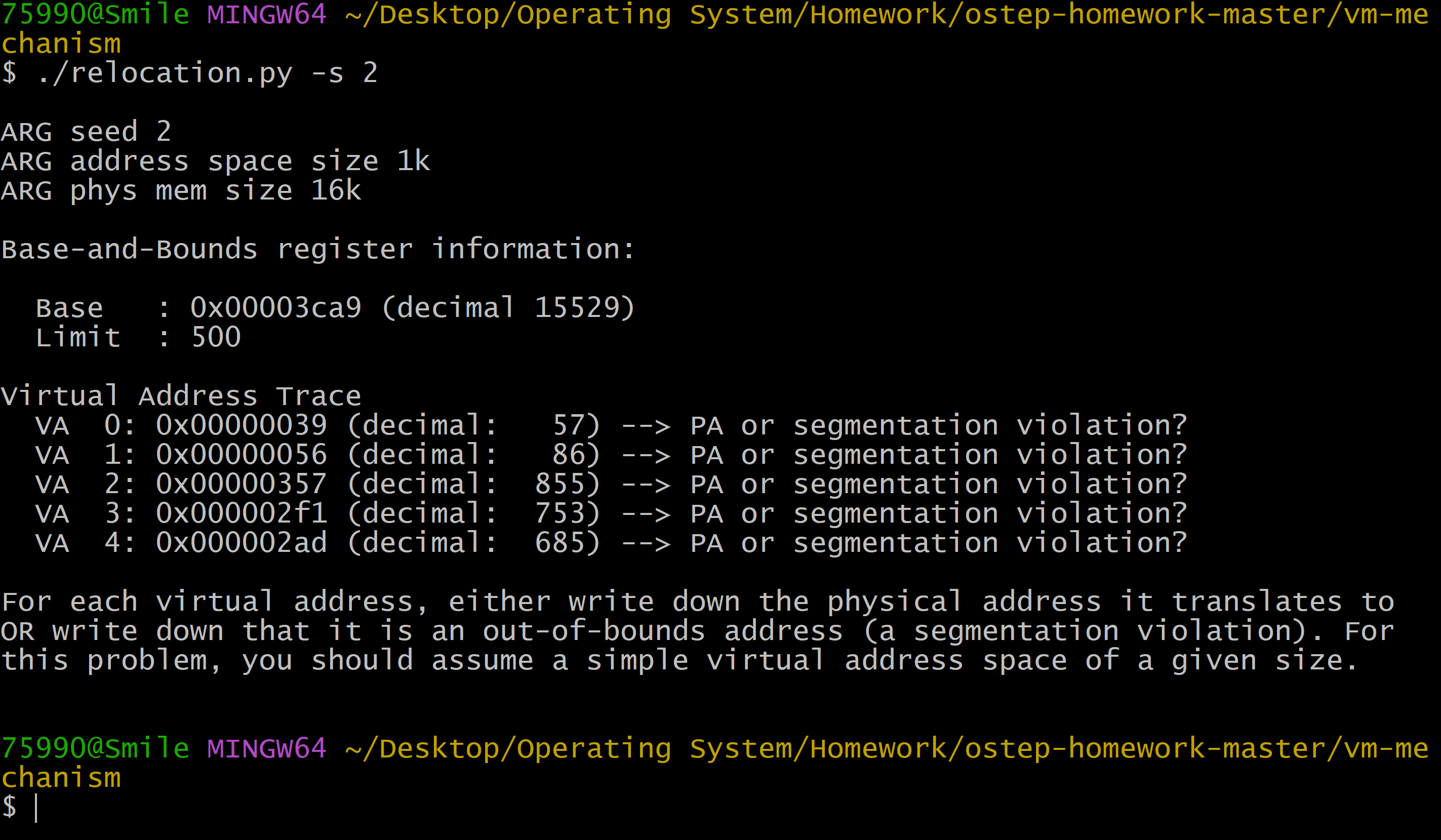
# Chp 15

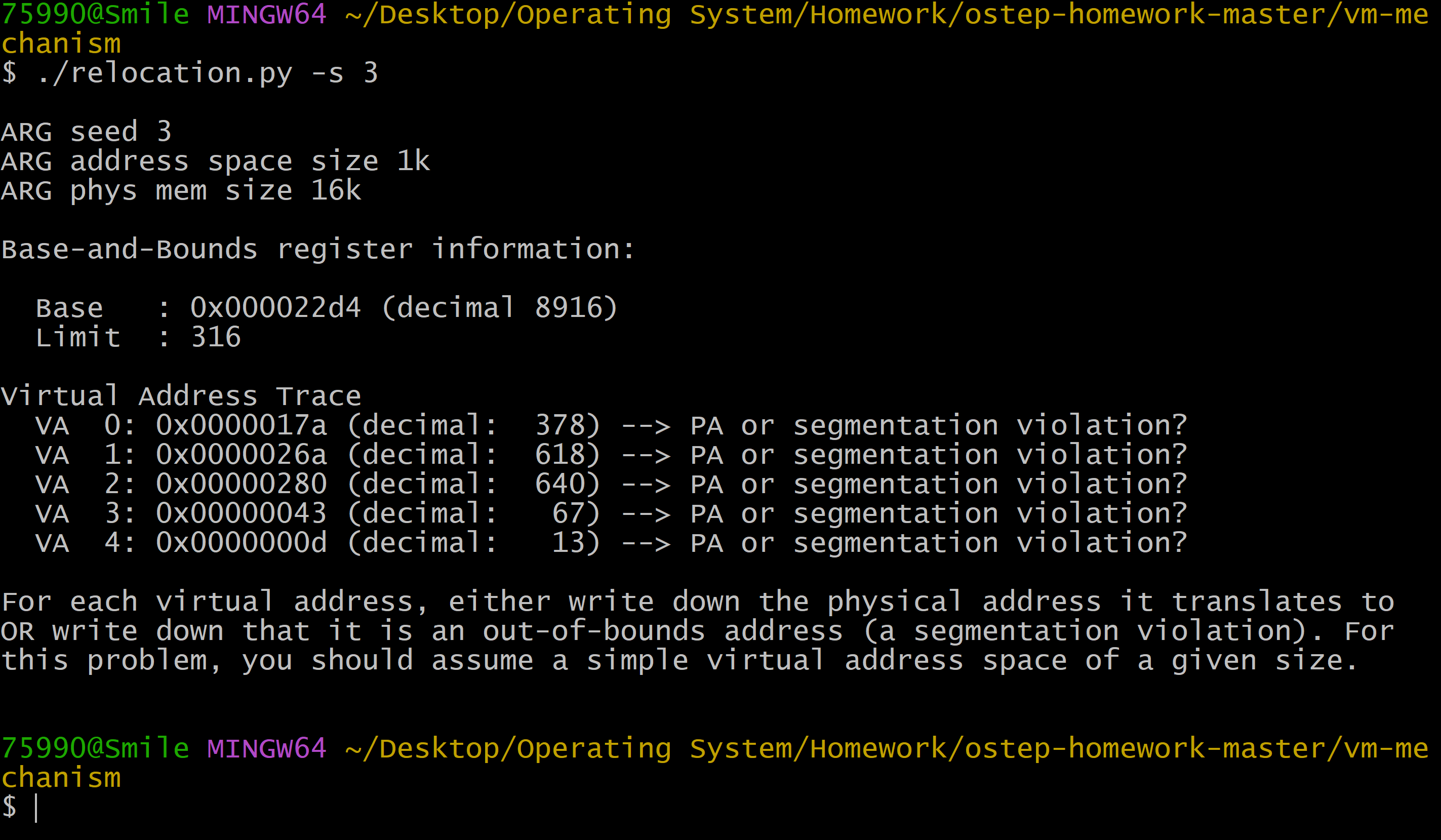
1.



1. Violation
2. PA，13884+261=14145
3. Violation
4. Violation
5. Violation

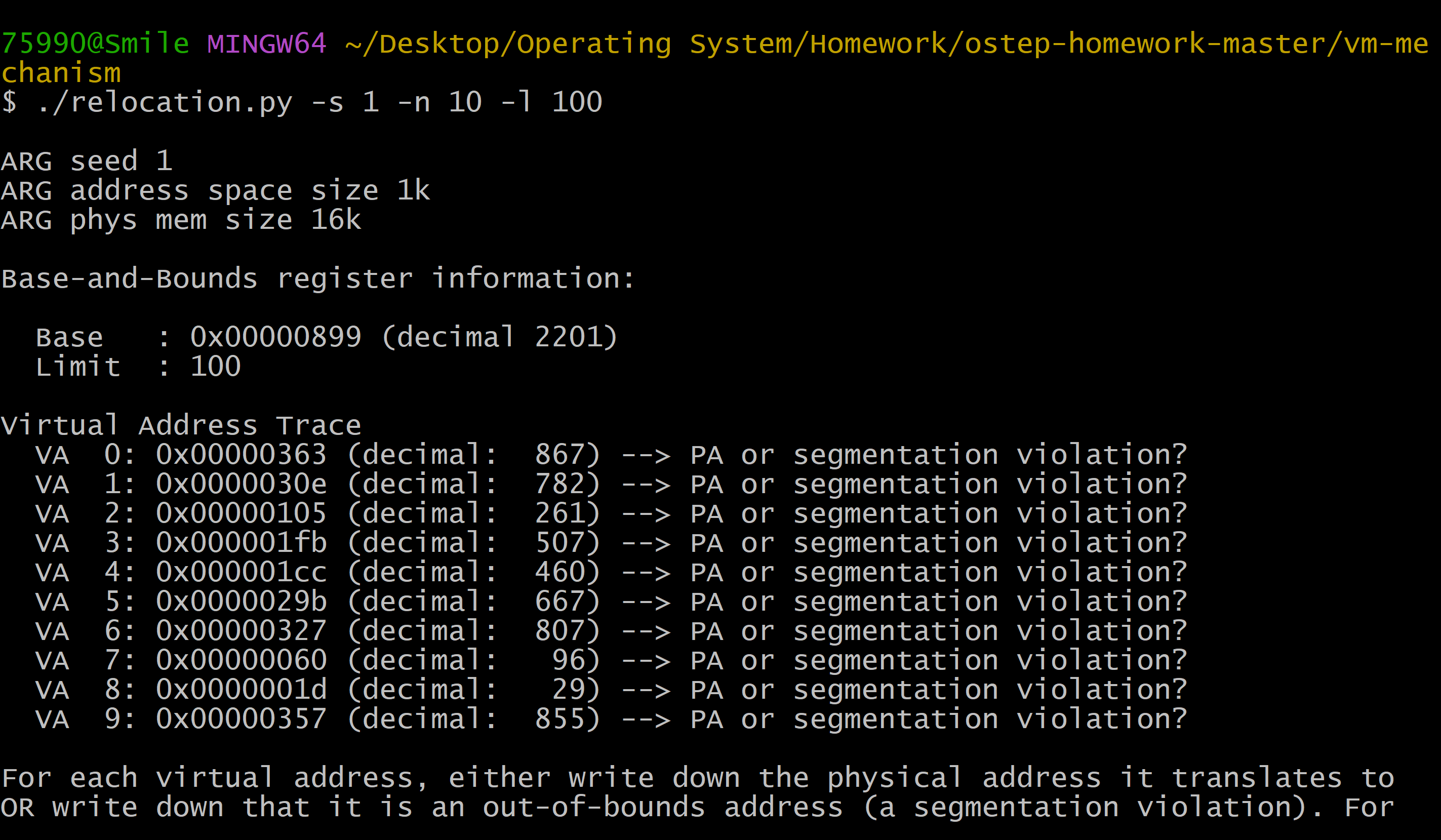


1. PA，15529+57=15586
2. PA，15615
3. Violation
4. Violation
5. Violation



1. Violation
2. Violation
3. Violation
4. PA，8923
5. PA，8929

3.

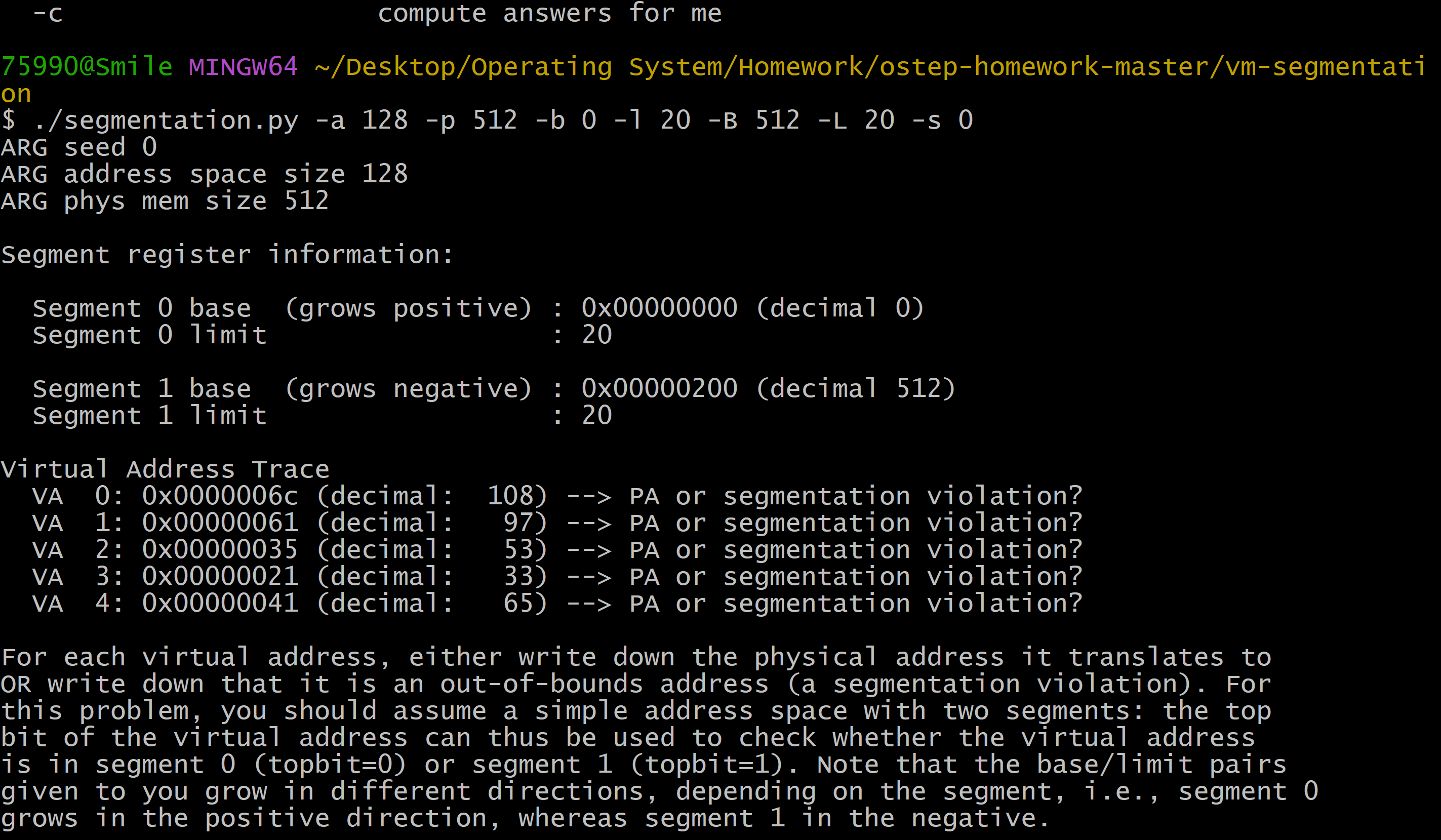


物理内存大小是16K，所以地址范围是0~16\*1024-1，即0~16383

最大的界限值为16383-899= 15485

# Chp 16

## 1.



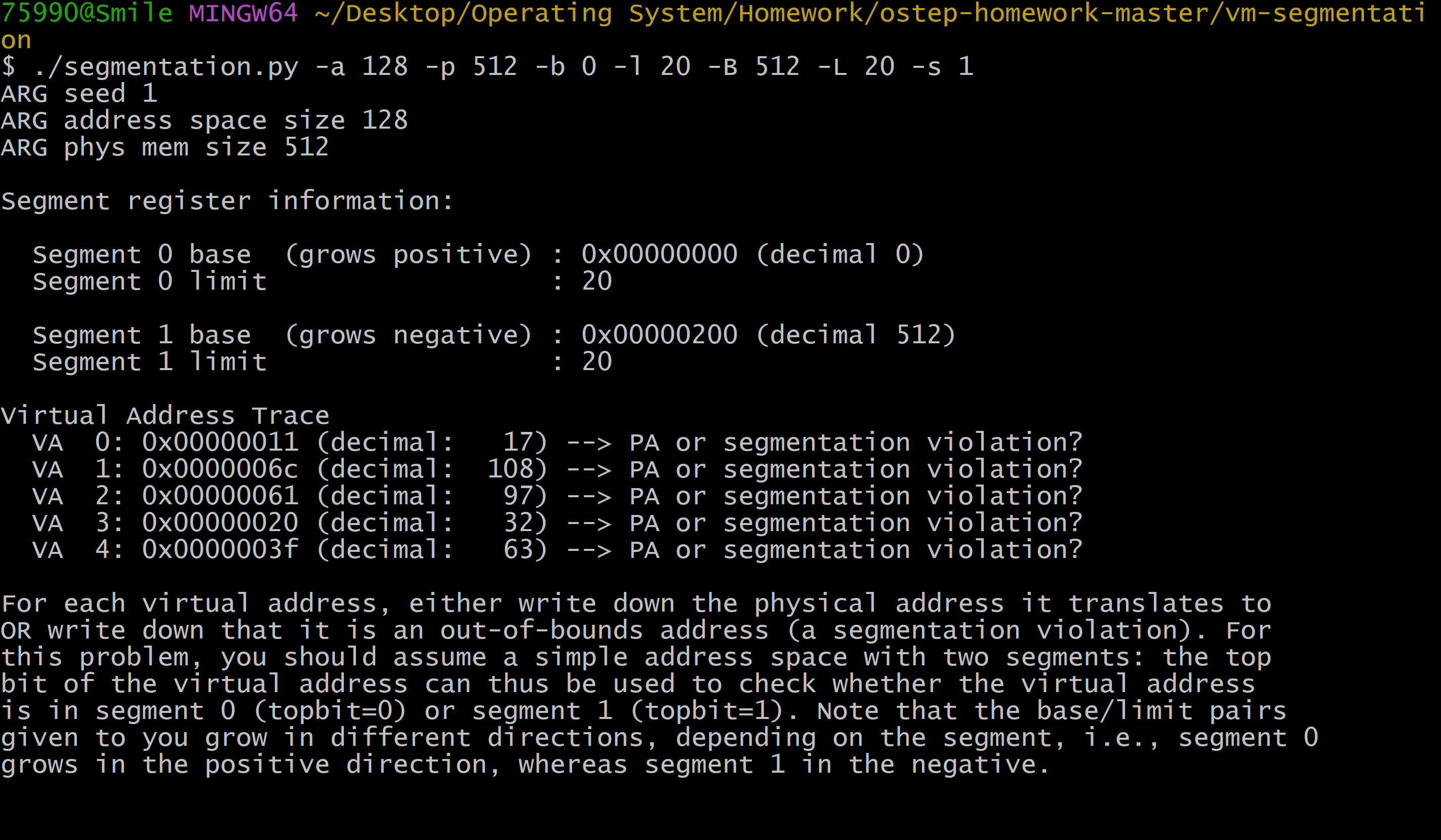
PA:492

Violation

Violation

Violation

Violation



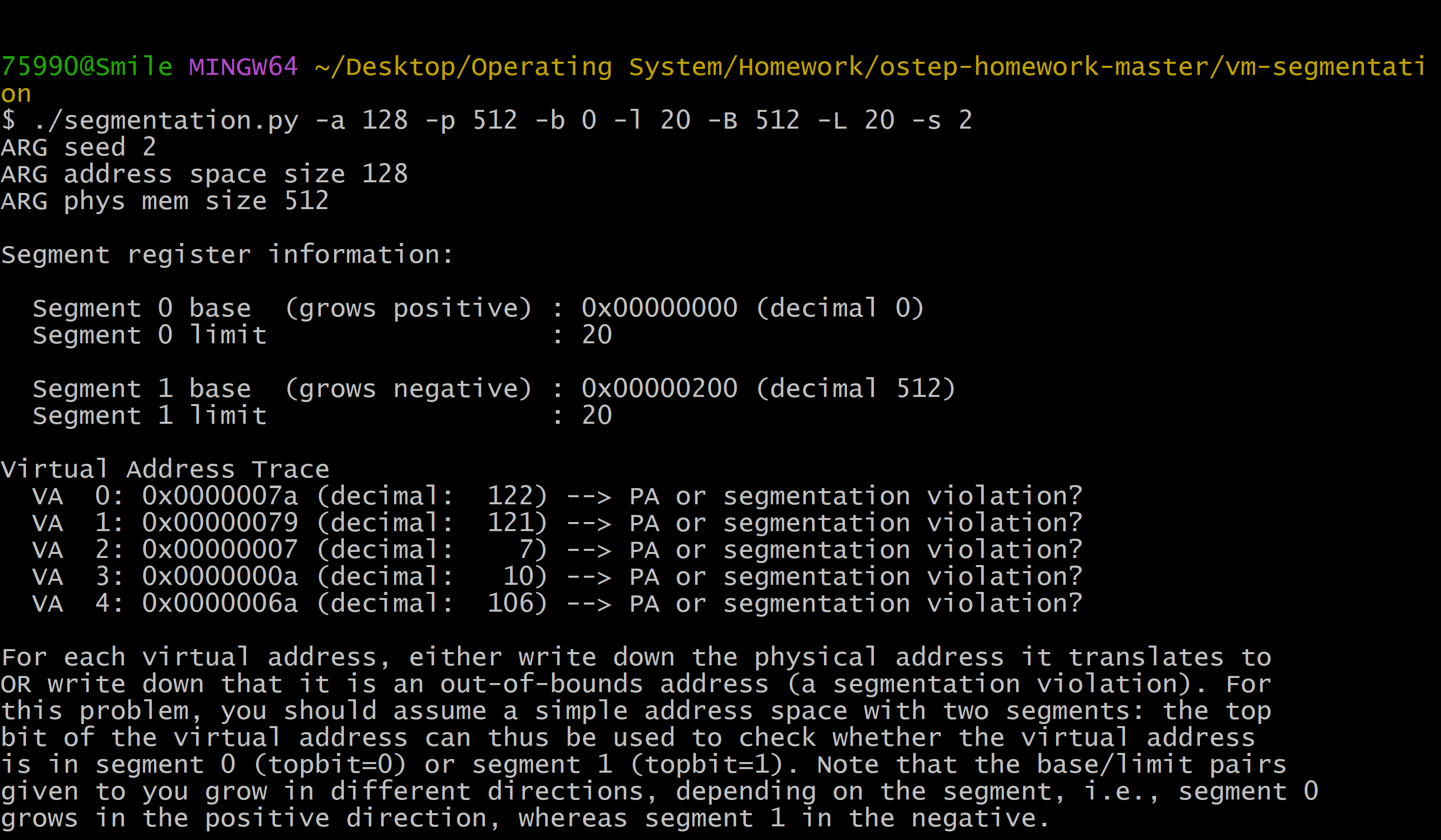
PA:17

PA:492

violation

violation

violation



PA:506

PA:505

PA:7

PA:10

Violation

## 2.

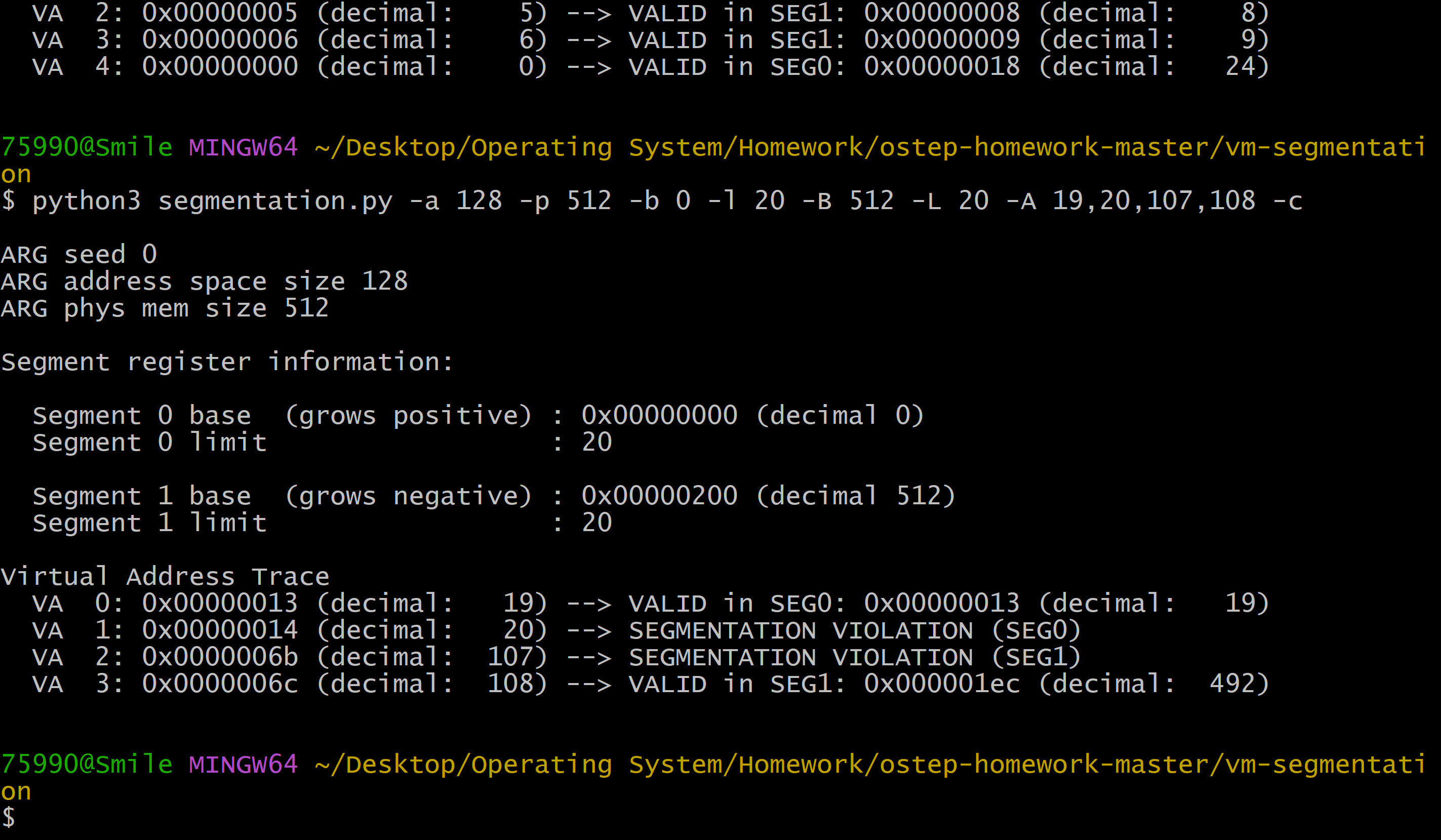
段0的最高合法虚拟地址为19

段1中最低的合法虚拟地址为108

整个地址空间中最低的非法地址为20

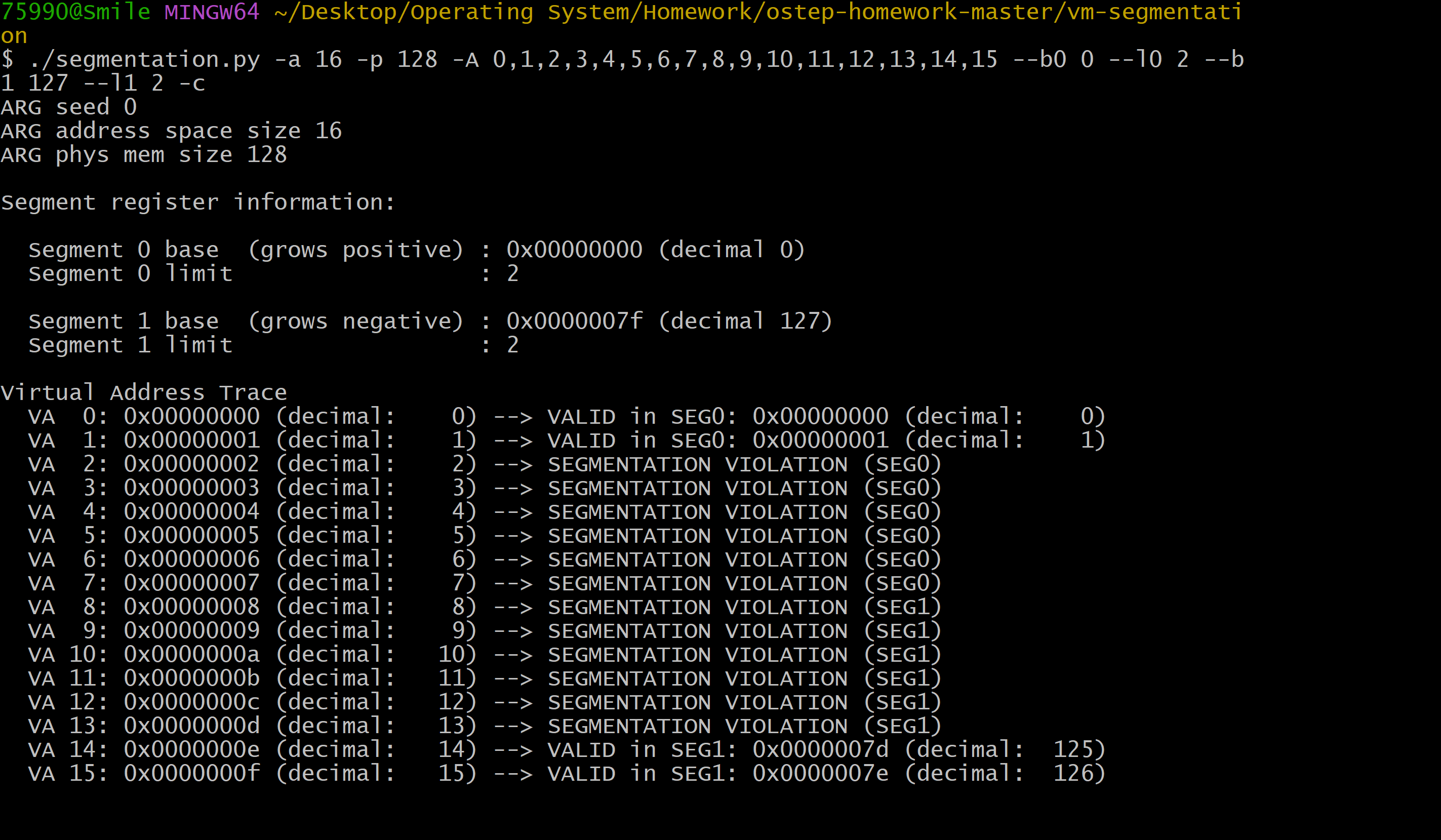
整个地址空间中最高非法地址为107

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



## 3.

使用的参数：--b0 0 --l0 2 --b1 127 --l1 2



# Chp 17

## 1.

|  |
| --- |
| ptr[0] = Alloc(3) returned 1000 (searched 1 elements)  Free List [ Size 1 ]: [ addr:1003 sz:97 ] |
| Free(ptr[0]) returned 0  Free List [ Size 2 ]: [ addr:1000 sz:3 ] [ addr:1003 sz:97 ] |
| ptr[1] = Alloc(5) returned 1003 (searched 2 elements)  Free List [ Size 2 ]: [ addr:1000 sz:3 ] [ addr:1008 sz:92 ] |
| Free(ptr[1]) returned 0  Free List [ Size 3 ]: [ addr:1000 sz:3 ] [ addr:1003 sz:5 ] [ addr:1008 sz:92 ] |
| ptr[2] = Alloc(8) returned 1008 (searched 3 elements)  Free List [ Size 3 ]: [ addr:1000 sz:3 ] [ addr:1003 sz:5 ] [ addr:1016 sz:84 ] |
| Free(ptr[2]) returned 0  Free List [ Size 4 ]: [ addr:1000 sz:3 ] [ addr:1003 sz:5 ] [ addr:1008 sz:8 ] [ addr:1016 sz:84 ] |
| ptr[3] = Alloc(8) returned 1008 (searched 4 elements)  Free List [ Size 3 ]: [ addr:1000 sz:3 ] [ addr:1003 sz:5 ] [ addr:1016 sz:84 ] |
| Free(ptr[3]) returned 0  Free List [ Size 4 ]: [ addr:1000 sz:3 ] [ addr:1003 sz:5 ] [ addr:1008 sz:8 ] [ addr:1016 sz:84 ] |
| ptr[4] = Alloc(2) returned 1000 (searched 4 elements)  Free List [ Size 4 ]: [ addr:1002 sz:1 ] [ addr:1003 sz:5 ] [ addr:1008 sz:8 ] [ addr:1016 sz:84 ] |
| ptr[5] = Alloc(7) returned 1008 (searched 4 elements)  Free List [ Size 4 ]: [ addr:1002 sz:1 ] [ addr:1003 sz:5 ] [ addr:1015 sz: 1] [ addr:1016 sz:84 ] |

发现：6次分配空间查找空闲块的次数分别为：1,2,3,4,4,4。

空闲列表里面的内存块越来越多了，且分配搜索空闲块的次数也随之增加。

## 3.

|  |  |
| --- | --- |
|  | ptr[0] = Alloc(3) returned 1000 (searched 1 elements)  Free List [ Size 1 ]: [ addr:1003 sz:97 ] |
|  | Free(ptr[0])  returned 0  Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1003 sz:97 ] |
|  | ptr[1] = Alloc(5) returned 1003 (searched 2 elements)  Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1008 sz:92 ] |
|  | Free(ptr[1])  returned 0  Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:92 ] |
|  | ptr[2] = Alloc(8) returned 1008 (searched 3 elements)  Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ] |
|  | Free(ptr[2]) returned 0  Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ] |
|  | ptr[3] = Alloc(8) returned 1008 (searched 3 elements)  Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ] |
|  | Free(ptr[3]) returned 0  Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ] |
|  | ptr[4] = Alloc(2) returned 1000 (searched 1 elements)  Free List [ Size 4 ]: [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ] |
|  | ptr[5] = Alloc(7) returned 1008 (searched 3 elements)  Free List [ Size 4 ]: [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1015 sz:1 ][ addr:1016 sz:84 ] |

6次分配空间所需要的搜索次数分别为：1,2,3,3,1,3，搜索空闲列表变快了。

## 4.

由于最优匹配和最差匹配都需要遍历整个空闲列表，不同的空闲列表排序对找到空闲位置的时间没有影响。

故只需要查看排序方法对首次匹配的时间产生的影响，比较使用不同排序需要的搜索次数。

运行程序后得到结果：

| **排序算法** | **各次搜索次数** |
| --- | --- |
| ADDRSORT | 1，2，3，3，1，3 |
| SIZESORT + | 1，2，3，3，1，3 |
| SIZESORT - | 1，1，1，1，1，1 |

SIZESORT-搜索次数显著减少。

因为排序将最大的空闲块放在列表第一位，首次匹配只要一次就可以完成搜索分配。

# Chp 18

## 1.

线性页表大小如何随着地址空间的增长而变化:

1024->2048->4096

在页大小相同的情况下，地址空间增长，页表项随之增长，页表增大。

线性页面大小如何随页大小的增长而变化：

1024->512->256

在地址空间大小相同的情况下，页的大小增大，页表项减少，页表减小。

使用很大的页会导致页内部的空间可能存在大量浪费和内部碎片，因此不应该使用很大的页。

## 2.

1. -u 0

所有的虚拟地址都是无效的

1. -u 25

Virtual Address Trace

VA 0x00003986 (decimal: 14726) --> Invalid (VPN 14 not valid)

VA 0x00002bc6 (decimal: 11206) --> 00004fc6 (decimal 20422) [VPN 10]

VA 0x00001e37 (decimal: 7735) --> Invalid (VPN 7 not valid)

VA 0x00000671 (decimal: 1649) --> Invalid (VPN 1 not valid)

VA 0x00001bc9 (decimal: 7113) --> Invalid (VPN 6 not valid)

1. -u 50

Virtual Address Trace

VA 0x00003385 (decimal: 13189) --> PA 16261

VA 0x0000231d (decimal: 8989) --> Invalid address

VA 0x000000e6 (decimal: 230) --> PA 24806

VA 0x00002e0f (decimal: 11791) --> Invalid address

VA 0x00001986 (decimal: 6534) --> PA 30086

1. -u 75

Virtual Address Trace

VA 0x00002e0f (decimal: 11791) --> 00004e0f (decimal 19983) [VPN 11]

VA 0x00001986 (decimal: 6534) --> 00007d86 (decimal 32134) [VPN 6]

VA 0x000034ca (decimal: 13514) --> 00006cca (decimal 27850) [VPN 13]

VA 0x00002ac3 (decimal: 10947) --> 00000ec3 (decimal 3779) [VPN 10]

VA 0x00000012 (decimal: 18) --> 00006012 (decimal 24594) [VPN 0]

1. -u 100

Virtual Address Trace

VA 0x00002e0f (decimal: 11791) --> 00004e0f (decimal 19983) [VPN 11]

VA 0x00001986 (decimal: 6534) --> 00007d86 (decimal 32134) [VPN 6]

VA 0x000034ca (decimal: 13514) --> 00006cca (decimal 27850) [VPN 13]

VA 0x00002ac3 (decimal: 10947) --> 00000ec3 (decimal 3779) [VPN 10]

VA 0x00000012 (decimal: 18) --> 00006012 (decimal 24594) [VPN 0]

## 3.

1. 现代计算机的页表大小至少是几 KB 甚至 MB 级别，而这里只有 8B，可能连存储一个页表项都不够。

地址空间大小 32B 也过小，几乎无法运行任何实际程序。地址空间包含的页面数量太少了

1. 相对现实，32KB 的地址空间虽然较小，但仍然可以用于嵌入式系统或早期计算机。
2. 不现实：1 MB 的页大小远远大于实际系统中常用的 4KB 或 8KB。

这样做会导致：

页表项数量极少，不利于内存的细粒度管理

内存利用率低，因为每次换入换出都是 1 MB 的数据，浪费大量内存资源

# Chp 20

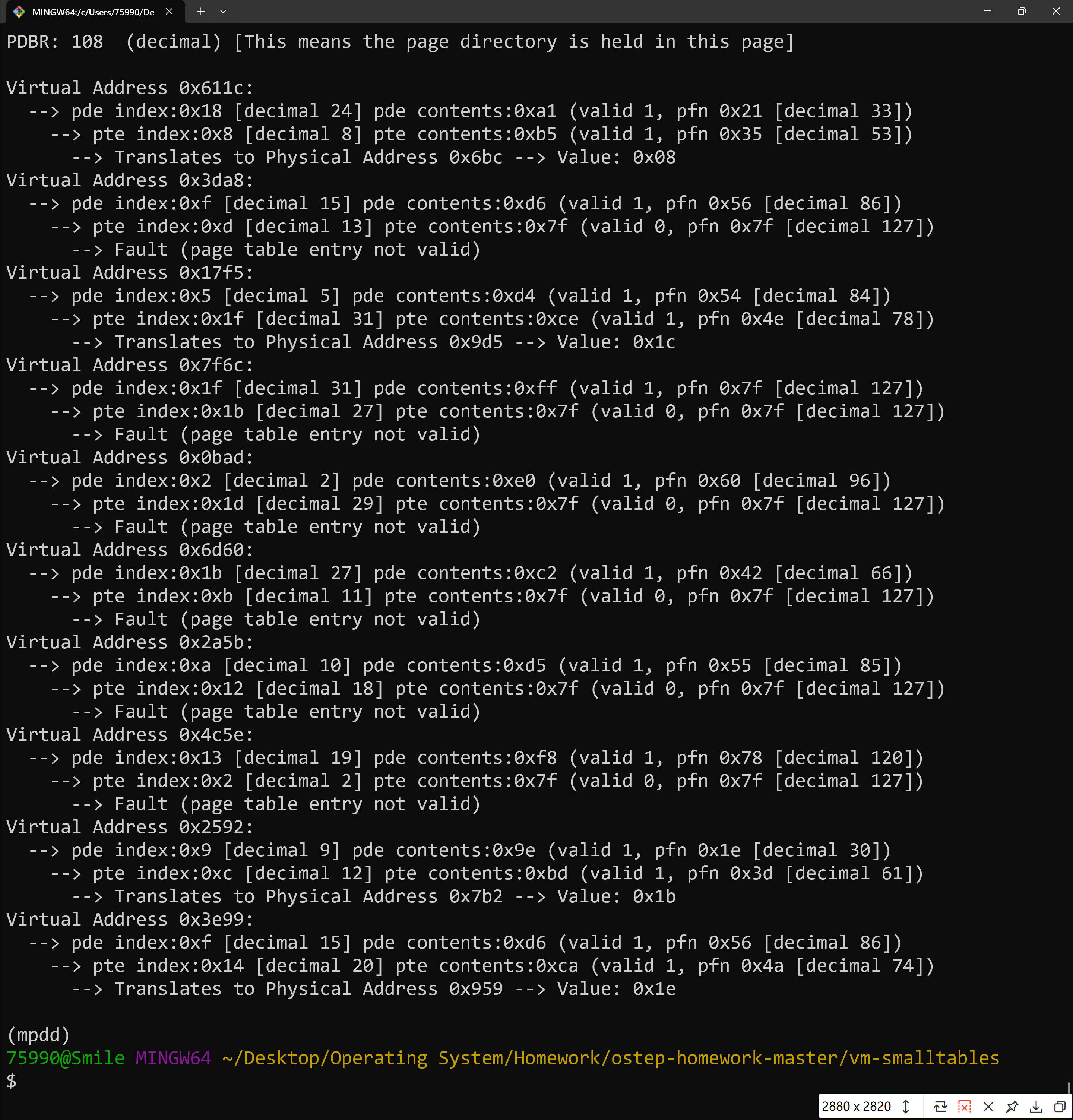
## 1.

对于二级页表，只需要一个寄存器找到页目录的位置，从页目录中找到存放页表的位置，从页表中就可以找到页表项。

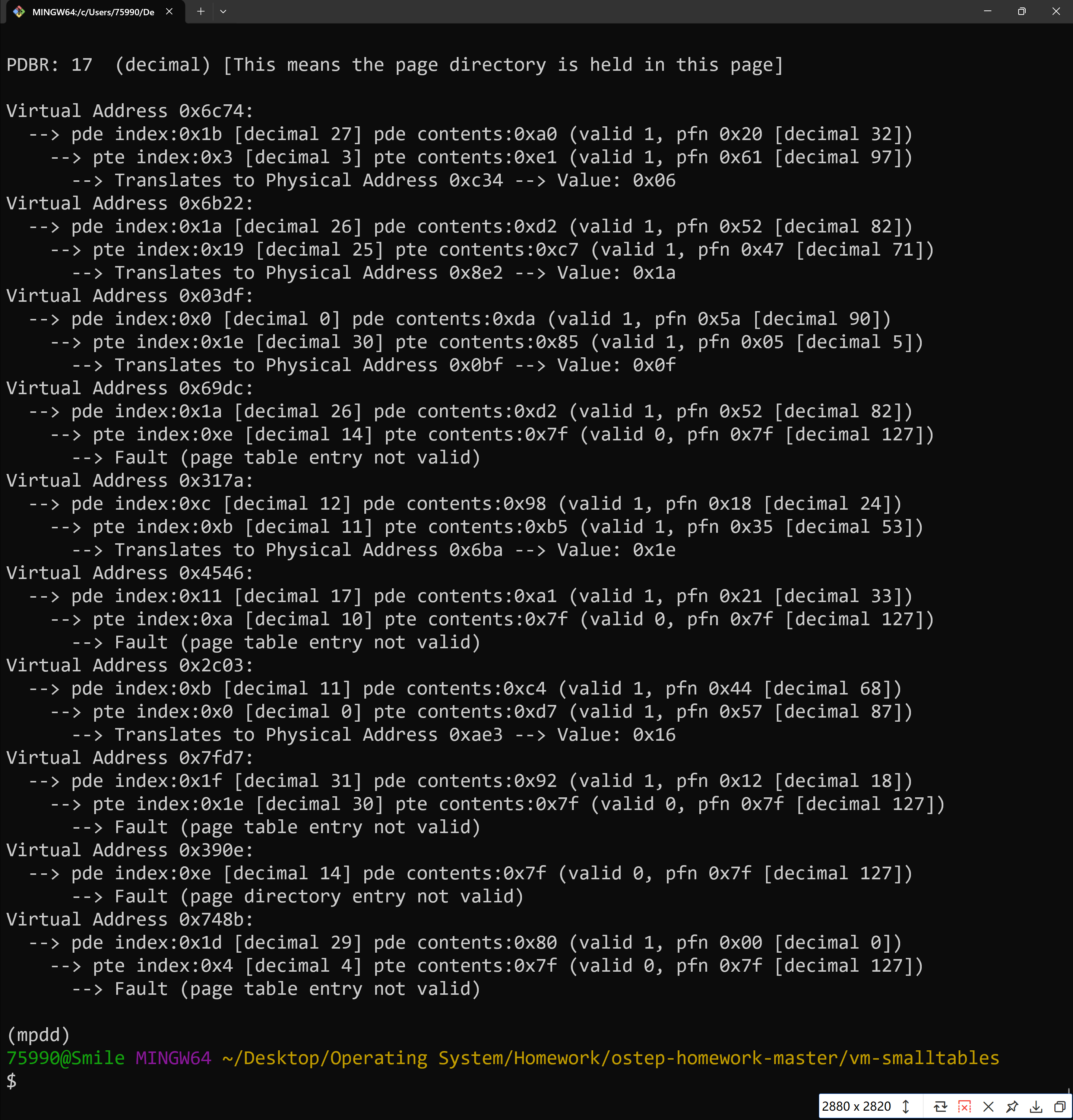
三级页表同样只需要一个寄存器找到页目录的位置，接下来就可以通过一级页目录找到二级页目录的位置，通过二级页目录找到页表的位置，最后找到页表项。

## 2.

-s 0



-s 1



-s 2



需要3次内存引用来执行每次查找

## 3.

初次访问内存中的某个位置时，会产生不命中，这个不命中是必然发生的。

系统将访问页表，找到虚拟页所对应的物理页，并将这个映射保存到缓存中。根据时间局部性与空间局部性，接下来的访问很有可能导致大量的缓存命中，从而导致快速访问。

在一些特定的情况下，程序短时间访问的页数大于缓存中的页数，也可能会产生大量的缓存不命中。

# Chp 22

## T1

先介绍一下可以使用的策略：  
**​​1. ​​FIFO (First-In, First-Out)​​**

* **​​默认策略​​，按照页面进入缓存的顺序替换。**
* **​​规则​​：替换最早进入缓存的页面（队首元素）。**
* **​​缺点​​：可能淘汰频繁访问的页面（即使最近仍在使用）。**

**2. ​​LRU (Least Recently Used)​​**

* **​​最近最少使用​​，基于时间局部性原理。**
* **​​规则​​：替换最长时间未被访问的页面。**
* **​​实现​​：维护访问时间戳或链表，每次访问时更新顺序。**
* **​​优点​​：对多数工作负载表现良好。**

**3. ​​MRU (Most Recently Used)​​**

* **​​最近最多使用​​，与LRU相反。**
* **​​规则​​：替换最近被访问过的页面。**
* **​​适用场景​​：某些特定访问模式（如循环访问序列）。**

**4. ​​OPT (Optimal Replacement)​​**

* **​​理想最优策略​​（理论上的上限）。**
* **​​规则​​：替换未来最长时间不会被访问的页面（需预知未来访问序列）。**
* **​​用途​​：作为性能比较基准，实际无法实现。**

**5. ​​UNOPT (Pessimal Replacement)​​**

* **​​最差策略​​（与OPT相反）。**
* **​​规则​​：故意替换未来最频繁访问的页面（反向OPT）。**
* **​​用途​​：测试或演示最坏情况下的性能。**

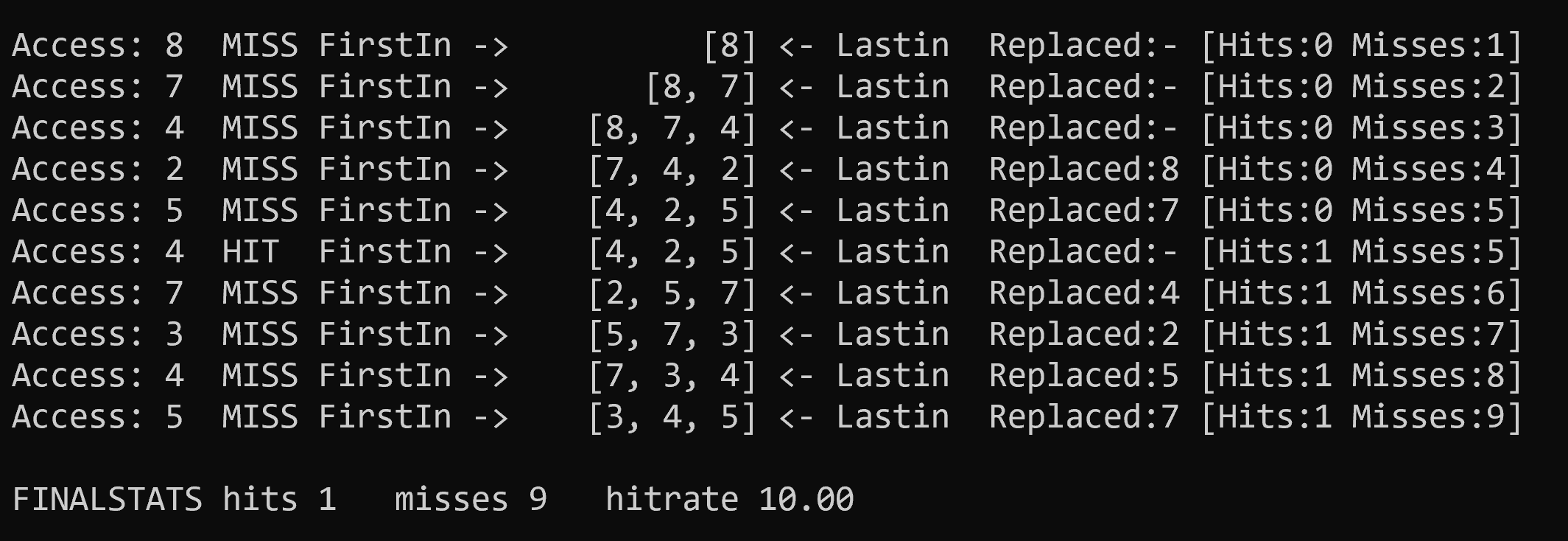
**6. ​​RAND (Random Replacement)​​**

* **​​随机替换​​。**
* **​​规则​​：随机选择一个页面替换。**
* **​​优点​​：实现简单，无额外开销；​​缺点​​：性能不稳定。**

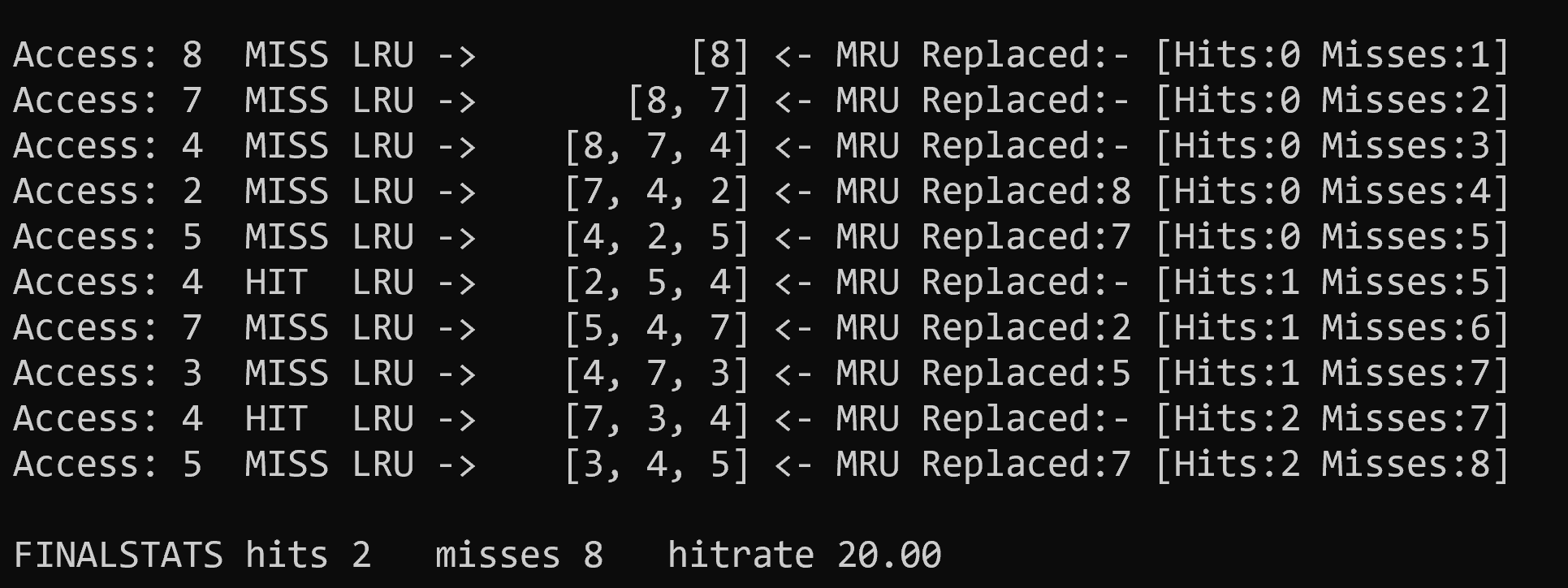
**7. ​​CLOCK (Clock Algorithm)​​**

* **​​近似LRU的改进版​​，使用环形链表（类似时钟指针）。**
* **​​规则​​：**
  1. **每个页面有一个引用位（clock bit），访问时置1。**
  2. **指针循环扫描页面，遇到引用位1则置0，遇到0则替换。**
* **​​参数 -b​​：指定每个页面的引用位数（如多级CLOCK），更多位数可提高精度。**
* **​​优点​​：平衡开销与效果，广泛用于实际系统。**

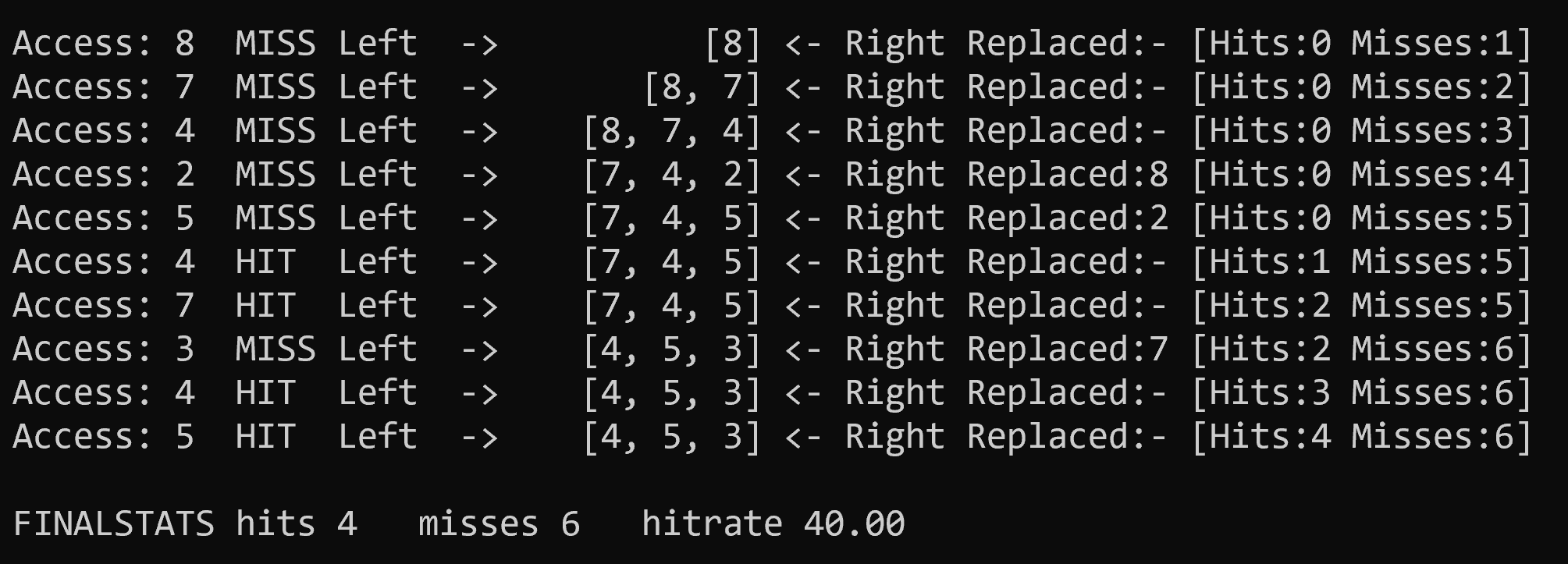
./paging-policy.py -s 0 -n 10



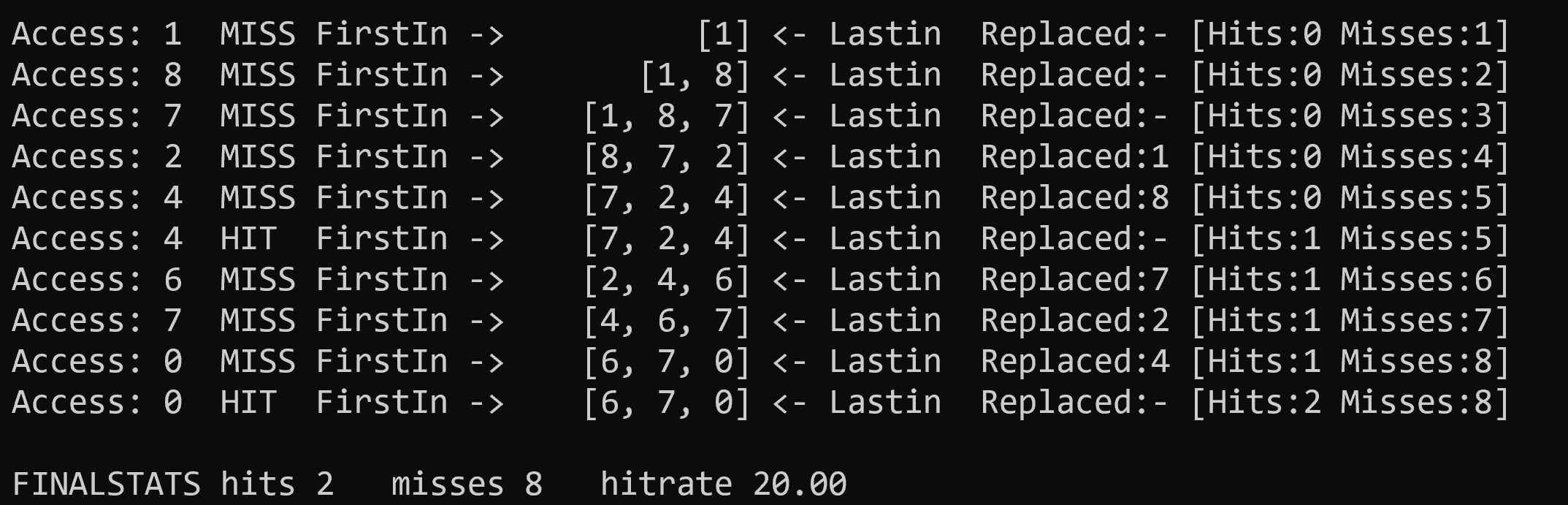
./paging-policy.py -s 0 -n 10 -p LRU



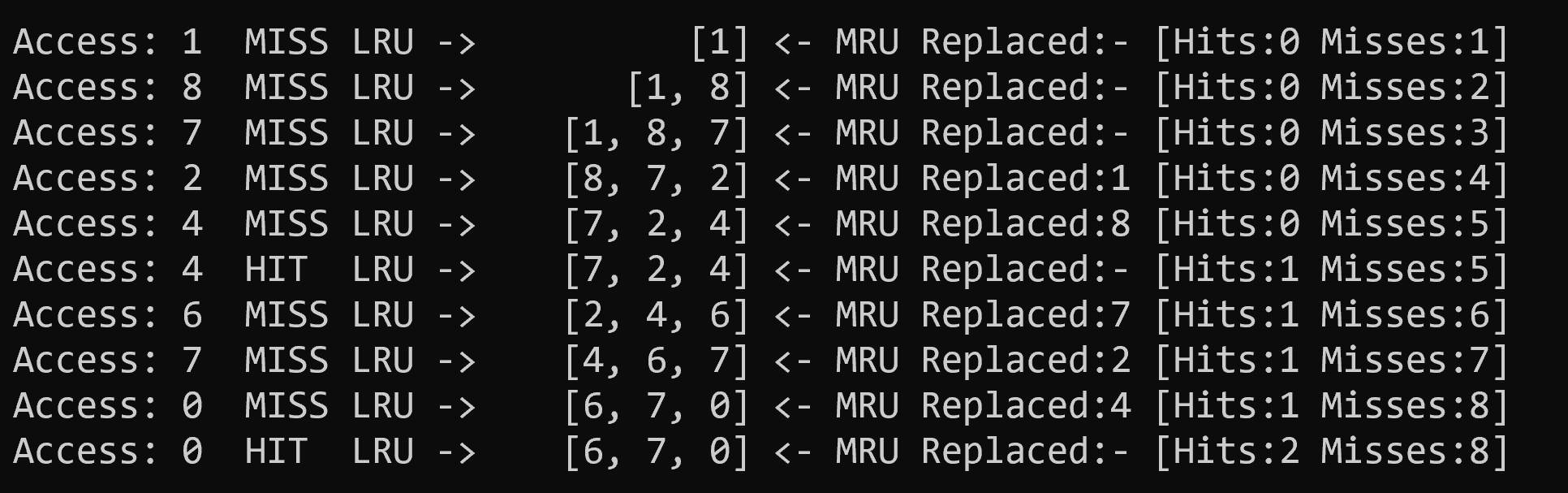
./paging-policy.py -s 0 -n 10 -p OPT



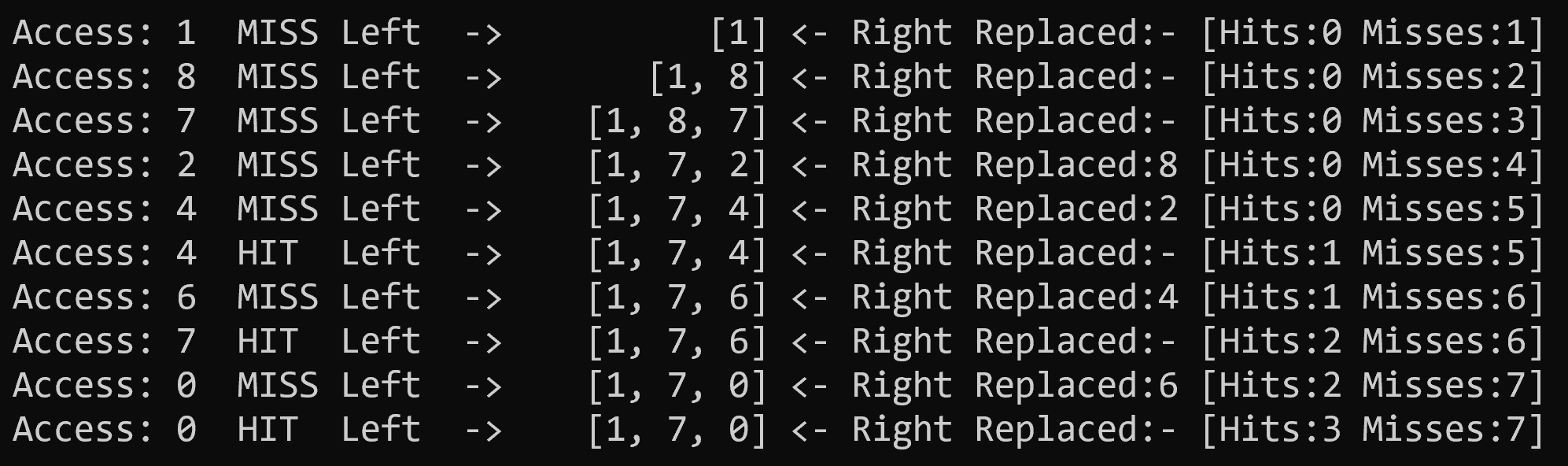
./paging-policy.py -s 1 -n 10



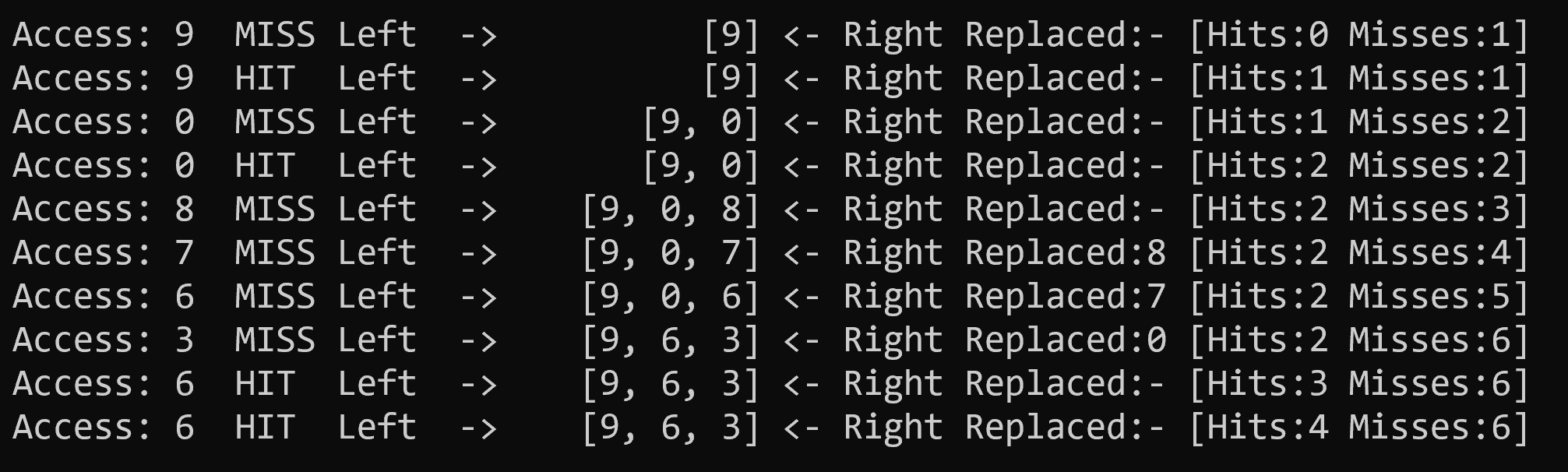
./paging-policy.py -s 1 -n 10 -p LRU



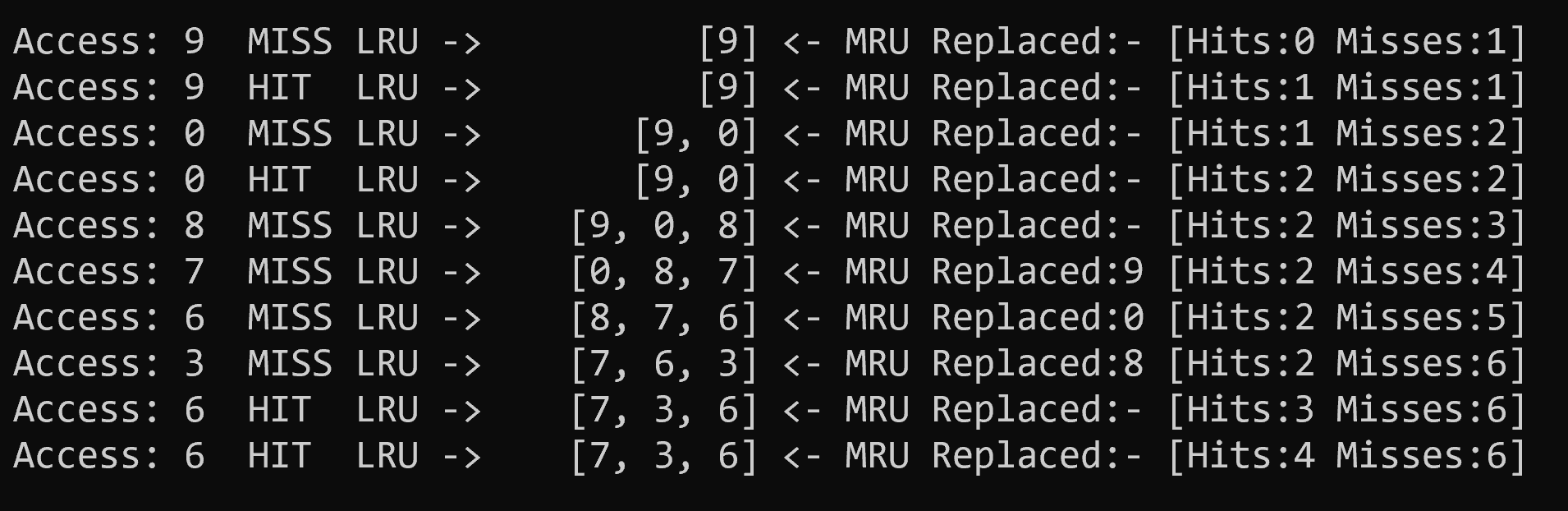
./paging-policy.py -s 1 -n 10 -p OPT



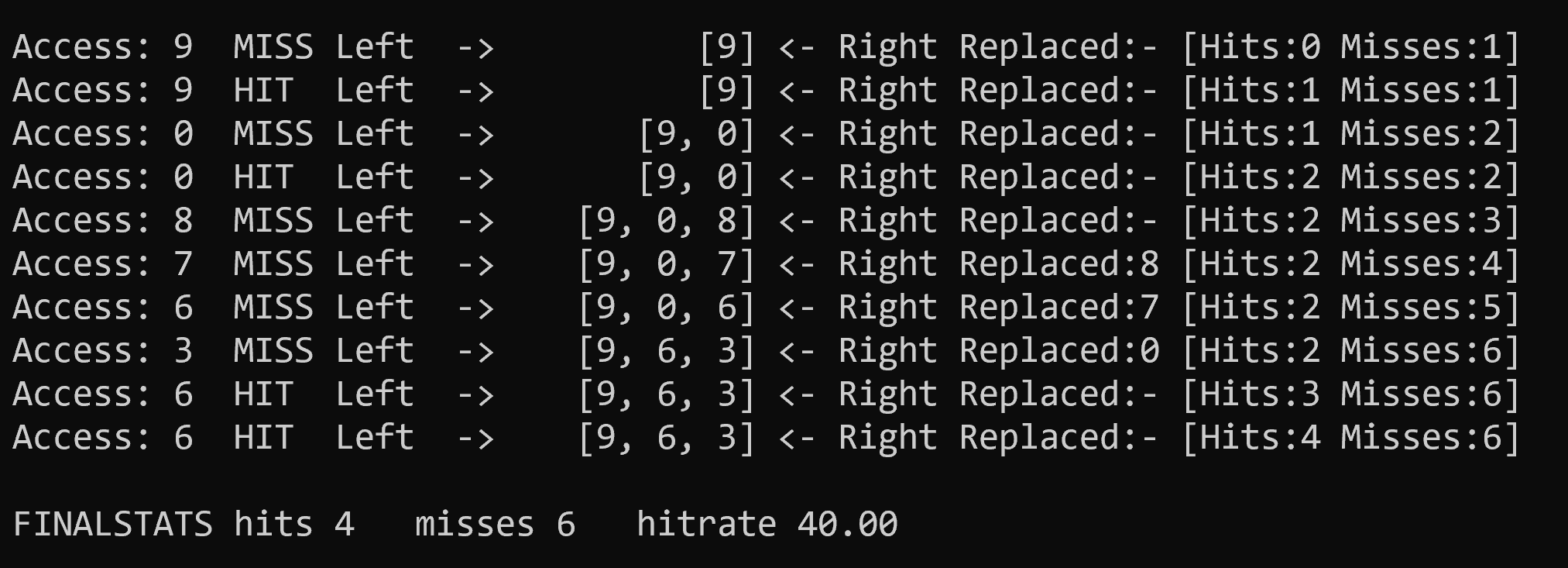
./paging-policy.py -s 2 -n 10



./paging-policy.py -s 2 -n 10 -p LRU



./paging-policy.py -s 2 -n 10 -p OPT



## T2

**FIFO**

最差情况的序列： 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, ...

改进⽅式：将缓存⼤⼩增加到6

**LRU**

最差情况的序列： 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, ...

改进⽅式：将缓存⼤⼩增加到6

**MRU**

最差情况的序列： 1, 2, 3, 4, 5, 6, 5, 6, 5, 6, ...

改进⽅式：将缓存⼤⼩增加到6

## T3

生成的随机序列：4 3 8 2 5 5 7 4 2 6

设缓存大小为5

**FIFO**策略：缓存命中了2次

**LRU**策略：缓存命中了2次

**MRU**策略：缓存命中了3次

由于这个序列的特殊性：最开始的地址访问被重复了一次，因此MRU策略在这个地址序列的表现优于LRU策略

## T4

使用如下python代码生成这样的追踪序列（1000次访问）

#tool.py

import random

import sys

numAddr = 10

# 空间局部性

def generate\_spatial\_locality\_trace():

    trace = [random.randint(0, numAddr)]

    for i in range(1000):

        l = trace[-1]

        rand = [l, (l + 1) % numAddr, (l - 1) % numAddr, random.randint(0, numAddr)]

        trace.append(random.choice(rand))

    # 问题给的paging-policy.py -a参数里, 逗号后不能空格,因此拼接再打印

    print(','.join([str(i) for i in trace]))

# 时间局部性

def generate\_temporal\_locality\_trace():

    trace = [random.randint(0, numAddr)]

    for i in range(1000):

        if len(trace) > 10 and random.random() < 0.7:

            # 70%概率从最近10个地址中选

            trace.append(random.choice(trace[-10:]))

        else:

            trace.append(random.randint(0, numAddr))

    print(','.join([str(i) for i in trace]))

if len(sys.argv) != 1:

    if sys.argv[1] == '-t':

        generate\_temporal\_locality\_trace()

    elif sys.argv[1] == '-s':

        generate\_spatial\_locality\_trace()

#用法如下

#python3 tool.py -s #产生具有空间局部性序列

#python3 tool.py -t #产生具有时间局部性序列

使用如下指令测试：  
./paging-policy.py -c -a $(python 22.4.py -t) -p LRU |grep "hitrate"

./paging-policy.py -c -a $(python 22.4.py -t) -p RAND |grep "hitrate"

./paging-policy.py -c -a $(python 22.4.py -t) -p CLOCK |grep "hitrate"

./paging-policy.py -c -a $(python 22.4.py -t) -p OPT |grep "hitrate"

./paging-policy.py -c -a $(python 22.4.py -s) -p LRU |grep "hitrate"

./paging-policy.py -c -a $(python 22.4.py -s) -p RAND |grep "hitrate"

./paging-policy.py -c -a $(python 22.4.py -s) -p CLOCK |grep "hitrate"

./paging-policy.py -c -a $(python 22.4.py -s) -p OPT |grep "hitrate"

测试结果如下：

| **策略** | **时间局部性-命中率** | **空间局部性-命中率** |
| --- | --- | --- |
| LRU | 64.24% | 55.84% |
| RAND | 59.34% | 52.55% |
| CLOCK | 60.64% | 55.14% |
| OPT | 75.82% | 66.33% |

可以发现OPT策略的效果最好

其次是LRU策略和CLOCK策略

表现最差的是RAND策略

CLOCK 使用不同数量的时钟位，结果如下：

$ ./paging-policy.py -c -a $(python 22.4.py -t) -p CLOCK -b 1 |grep "hitrate"

./paging-policy.py -c -a $(python 22.4.py -t) -p CLOCK -b 2 |grep "hitrate"

./paging-policy.py -c -a $(python 22.4.py -t) -p CLOCK -b 3 |grep "hitrate"

./paging-policy.py -c -a $(python 22.4.py -t) -p CLOCK -b 4 |grep "hitrate"

./paging-policy.py -c -a $(python 22.4.py -t) -p CLOCK -b 5 |grep "hitrate"

./paging-policy.py -c -a $(python 22.4.py -t) -p CLOCK -b 6 |grep "hitrate"

./paging-policy.py -c -a $(python 22.4.py -s) -p CLOCK -b 1 |grep "hitrate"

./paging-policy.py -c -a $(python 22.4.py -s) -p CLOCK -b 2 |grep "hitrate"

./paging-policy.py -c -a $(python 22.4.py -s) -p CLOCK -b 3 |grep "hitrate"

./paging-policy.py -c -a $(python 22.4.py -s) -p CLOCK -b 4 |grep "hitrate"

./paging-policy.py -c -a $(python 22.4.py -s) -p CLOCK -b 5 |grep "hitrate"

./paging-policy.py -c -a $(python 22.4.py -s) -p CLOCK -b 6 |grep "hitrate"

FINALSTATS hits 635 misses 366 hitrate 63.44

FINALSTATS hits 632 misses 369 hitrate 63.14

FINALSTATS hits 638 misses 363 hitrate 63.74

FINALSTATS hits 655 misses 346 hitrate 65.43

FINALSTATS hits 643 misses 358 hitrate 64.24

FINALSTATS hits 645 misses 356 hitrate 64.44

FINALSTATS hits 518 misses 483 hitrate 51.75

FINALSTATS hits 525 misses 476 hitrate 52.45

FINALSTATS hits 522 misses 479 hitrate 52.15

FINALSTATS hits 516 misses 485 hitrate 51.55

FINALSTATS hits 533 misses 468 hitrate 53.25

FINALSTATS hits 529 misses 472 hitrate 52.85

可以观察到，不论是对于时间局部性还是空间局部性，在一定的范围内，随着时钟位的增加，CLOCK策略的效果逐步提升。但是超过一定范围效果会减弱直至保持不变，甚至会回落。