OS_homework_2

这份文件是OS_homework_2 by计科210X 甘晴void 202108010XXX

文档设置了目录，可以通过目录快速跳转至答案部分。

第**15**章

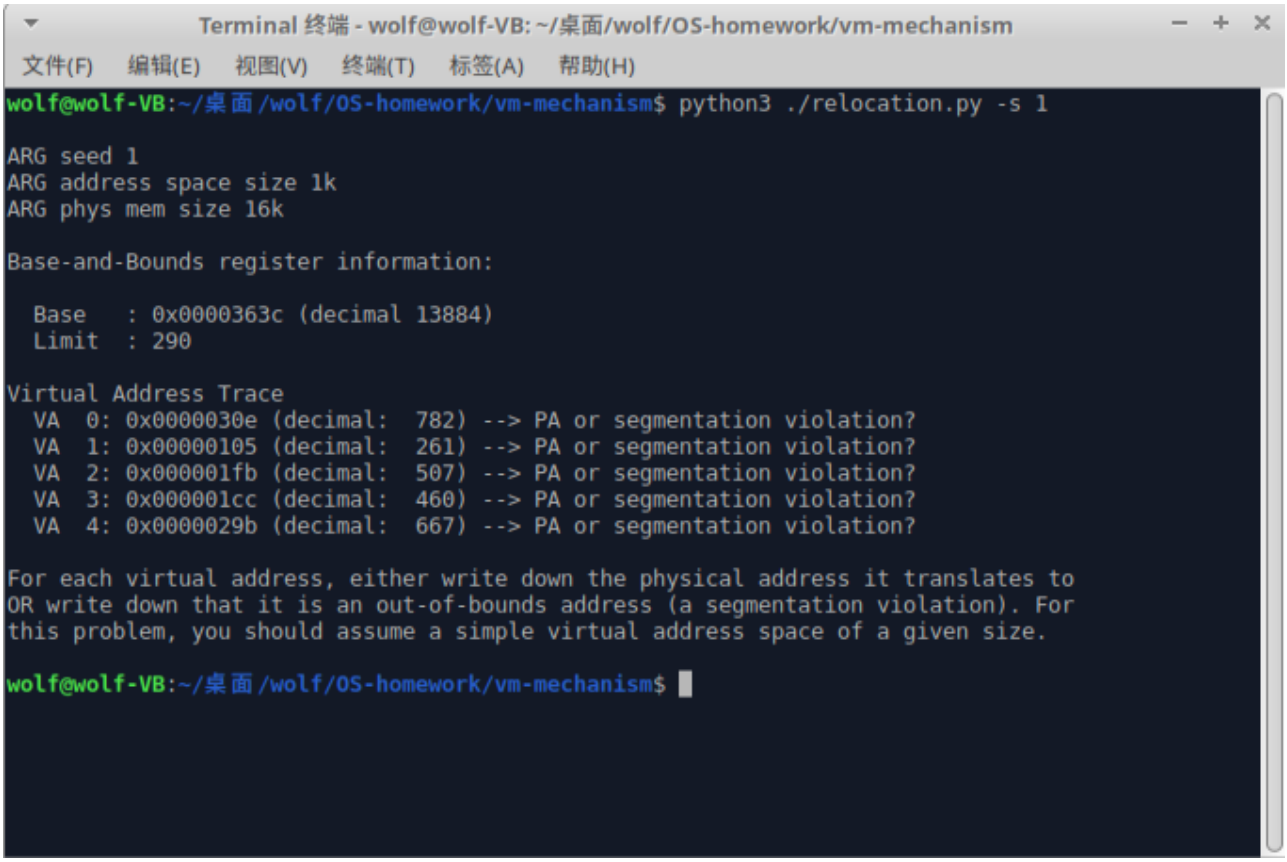
运行程序OS-homework/vm-mechanism/relocation.py

15.1

> 用种子 *1*、*2* 和 *3* 运行，并计算进程生成的每个虚拟地址是处于界限内还是界限外*?* 如果在界限内，请计算地址转换。

①种子**1**

运行截图如下：

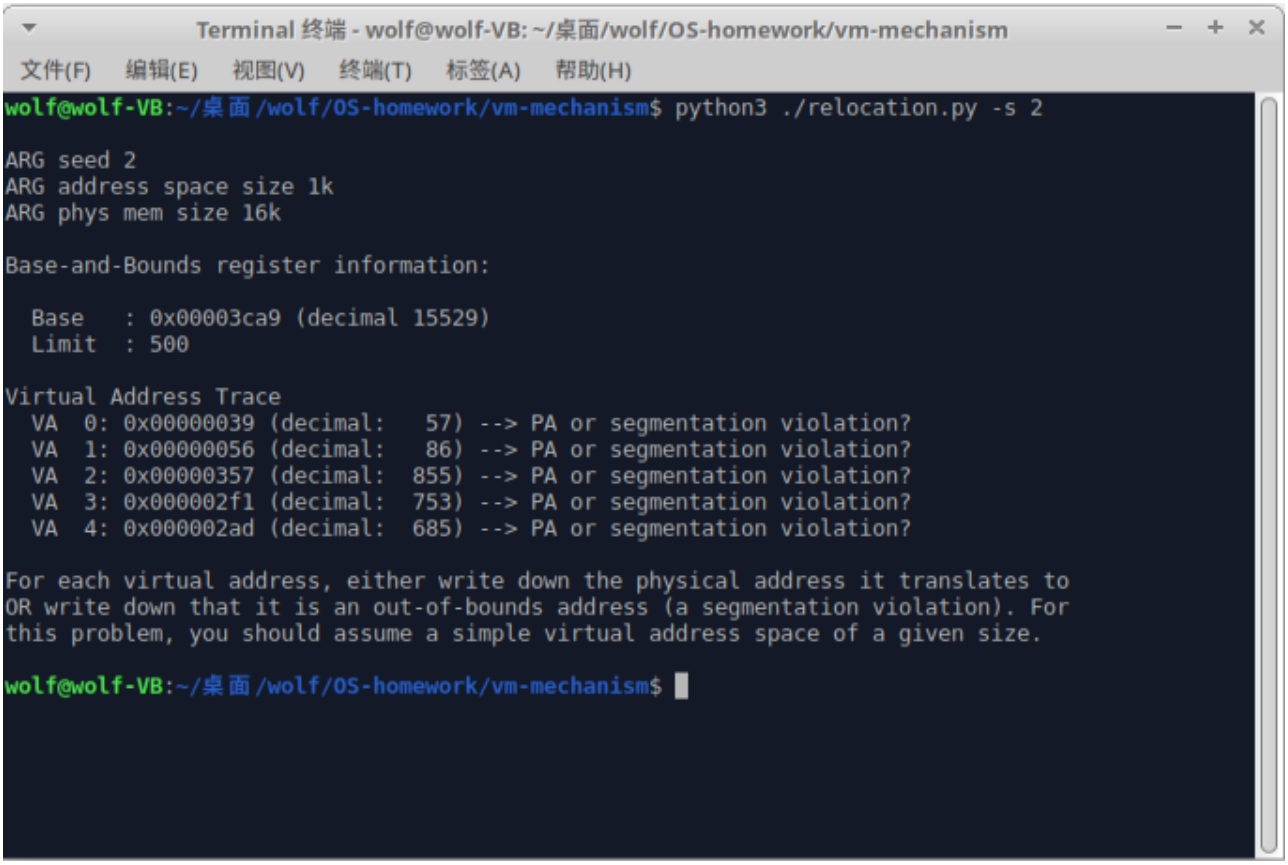


答案如下：

```
  VA  0: 0x0000030e (decimal:  782) --> SEGMENTATION VIOLATION（界限外）
  VA  1: 0x00000105 (decimal:  261) --> VALID: 0x00003741 (decimal: 14145)
  VA  2: 0x000001fb (decimal:  507) --> SEGMENTATION VIOLATION（界限外）
  VA  3: 0x000001cc (decimal:  460) --> SEGMENTATION VIOLATION（界限外）
  VA  4: 0x0000029b (decimal:  667) --> SEGMENTATION VIOLATION（界限外）
```

②种子**2**

运行截图如下：

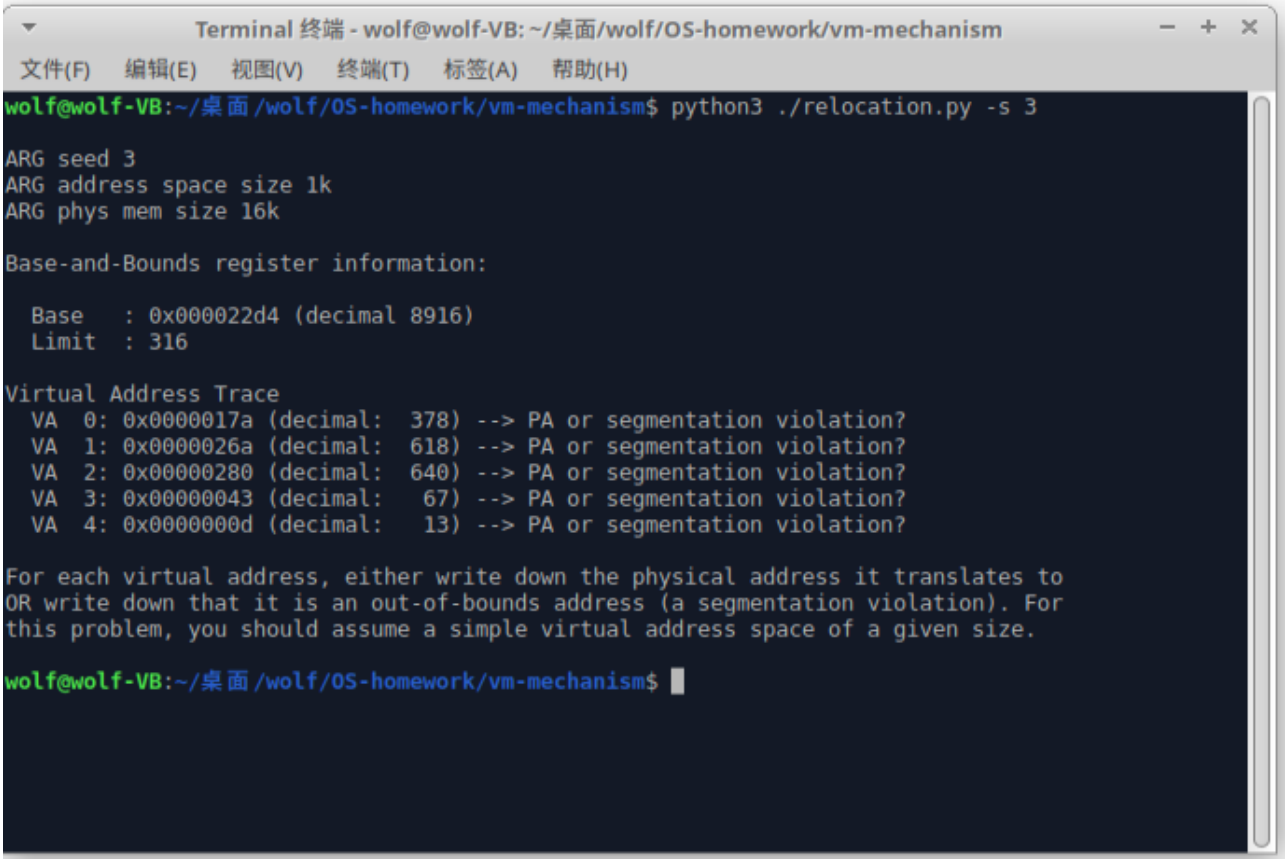


答案如下：

```
  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（界限
外）
  VA  3: 0x000002f1 (decimal:  753) --> SEGMENTATION VIOLATION（界限
外）
  VA  4: 0x000002ad (decimal:  685) --> SEGMENTATION VIOLATION（界限
外）
```

③种子**3**

运行截图如下：



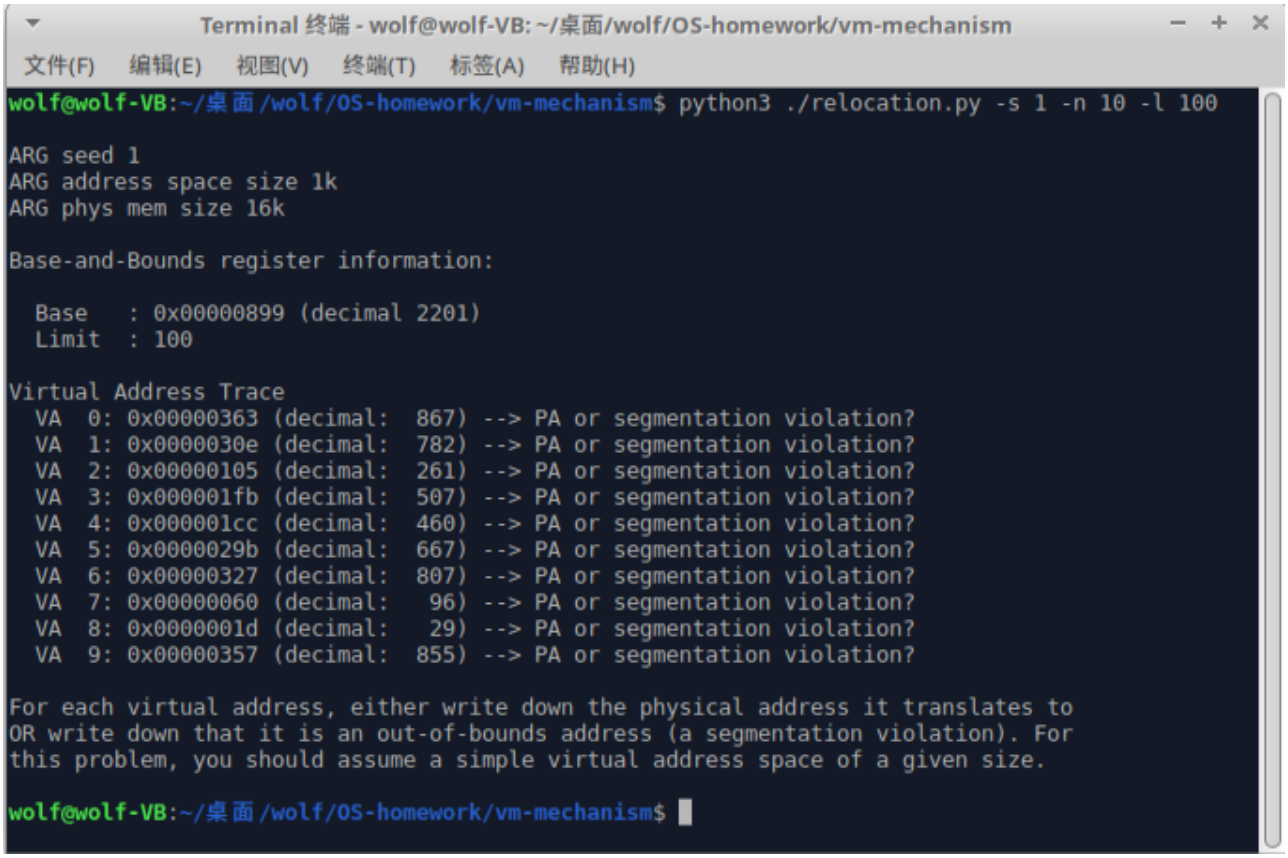
答案如下：

```
  VA  0: 0x0000017a (decimal:  378) --> SEGMENTATION VIOLATION（界限外）
  VA  1: 0x0000026a (decimal:  618) --> SEGMENTATION VIOLATION（界限外）
  VA  2: 0x00000280 (decimal:  640) --> SEGMENTATION VIOLATION（界限外）
  VA  3: 0x00000043 (decimal:   67) --> VALID: 0x00002317 (decimal: 8983)
  VA  4: 0x0000000d (decimal:   13) --> VALID: 0x000022e1 (decimal: 8929)
```

15.3

> 使用以下标志运行：*-s 1 -n 10 -l 100*。可以设置基址的最大值是多少，以便地址空间仍然完全放在物理内存中？

运行截图如下：



解答如下：

（这里无需考虑给出的这些VA，只需关注界限值和最大内存空间即可）

界限值为100，最大内存空间为16k，最大的基址值为16k-100=16284

第16章

运行程序OS-homework/vm-segmentation/segmentation.py

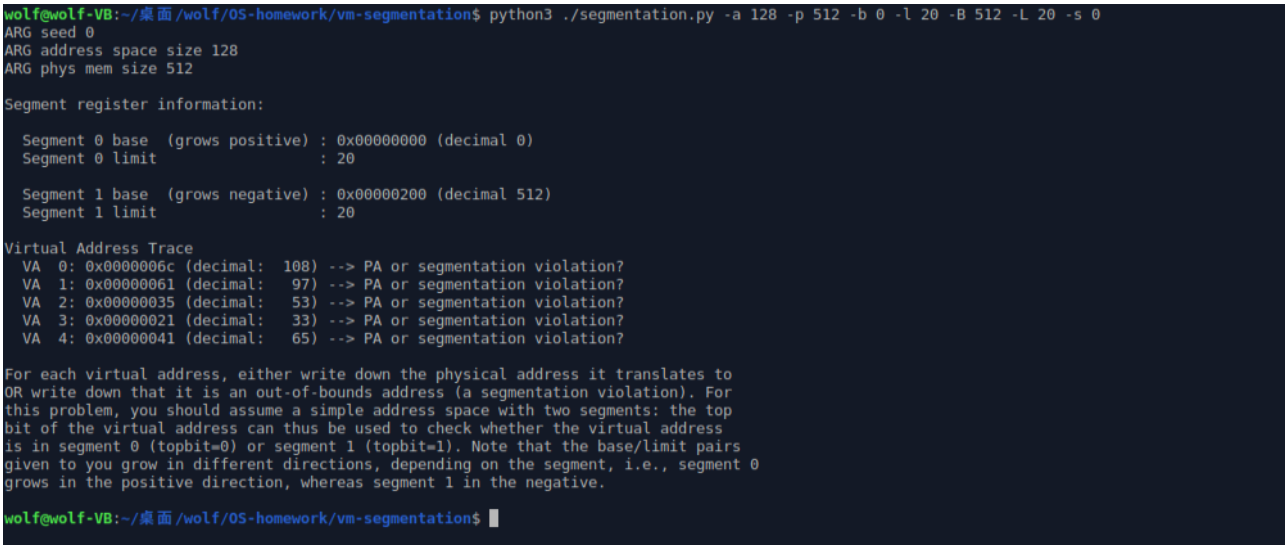
16.1

> 先让我们用一个小地址空间来转换一些地址。这里有一组简单的参数和几个不同的随机种子。你可以转换这些地址吗。

```
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2
```

①种子**0**

运行截图如下：



针对这个我可以多做一些说明：

以第一个为例。0x0000006c写成二进制为110 1100，其中第一个1指定段，后面的10 1100为段内偏移，但是因为是负向增长，所以要减去最大的段地址为（二进制下的）100 0000即64，所以44-64=-20

最终答案如下：

（先根据最高位（第7位）判断段，再计算偏移量OFFSET，若OFFSET非法则结束，最后根据基址加界限计算有效地址）

```
  VA  0: 1101100 -> OFFSET=-20 --> VALID in SEG1: 0x000001ec
(decimal:  492)
  VA  1: 1100001 -> OFFSET=-31 --> SEGMENTATION VIOLATION (SEG1)
  VA  2: 0110101 -> OFFSET=53  --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0100001 -> OFFSET=33  --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 1000001 -> OFFSET=-63 --> SEGMENTATION VIOLATION (SEG1)
```

②种子**1**

运行截图如下：

```
wolf@wolf-VB:~/桌面/wolf/OS-homework/vm-segmentation$ python3 ./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?

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 address space with two segments: the top
bit of the virtual address can thus be used to check whether the virtual address
is in segment 0 (topbit=0) or segment 1 (topbit=1). Note that the base/limit pairs
given to you grow in different directions, depending on the segment, i.e., segment 0
grows in the positive direction, whereas segment 1 in the negative.

wolf@wolf-VB:~/桌面/wolf/OS-homework/vm-segmentation$
```

最终答案如下：

```
  VA  0: 0010001 -> OFFSET=17  --> VALID in SEG0: 0x00000011
(decimal:  17)
  VA  1: 1101100 -> OFFSET=-20 --> VALID in SEG1: 0x000001ec
(decimal:  492)
  VA  2: 1100001 -> OFFSET=-31 --> SEGMENTATION VIOLATION (SEG1)
  VA  3: 0100000 -> OFFSET=32  --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0111111 -> OFFSET=63  --> SEGMENTATION VIOLATION (SEG0)
```

③种子**2**

运行截图如下：

```
wolf@wolf-VB:~/桌面/wolf/OS-homework/vm-segmentation$ python3 ./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?

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 address space with two segments: the top
bit of the virtual address can thus be used to check whether the virtual address
is in segment 0 (topbit=0) or segment 1 (topbit=1). Note that the base/limit pairs
given to you grow in different directions, depending on the segment, i.e., segment 0
grows in the positive direction, whereas segment 1 in the negative.

wolf@wolf-VB:~/桌面/wolf/OS-homework/vm-segmentation$
```

最终答案如下：

```
  VA  0: 1111010 -> OFFSET=-6  --> VALID in SEG1: 0x000001fa
(decimal:  506)
  VA  1: 1111001 -> OFFSET=-7  --> VALID in SEG1: 0x000001f9
(decimal:  505)
  VA  2: 0000111 -> OFFSET=7   --> VALID in SEG0: 0x00000007
(decimal:  7)
  VA  3: 0001010 -> OFFSET=10  --> VALID in SEG0: 0x0000000a
(decimal:  10)
  VA  4: 1101010 -> OFFSET=-22 --> SEGMENTATION VIOLATION (SEG1)
```

16.2

> 现在，让我们看看是否理解了这个构建的小地址空间（使用上面问题的参数）。段 *0* 中最高的合法虚拟地址是什么？段 *1* 中最低的合法虚拟地址是什么？在整个地址空间中， 最低和最高的非法地址是什么？最后，如何运行带有*-A* 标志的 *segmentation.py* 来测试你是 否正确？

答案如下：

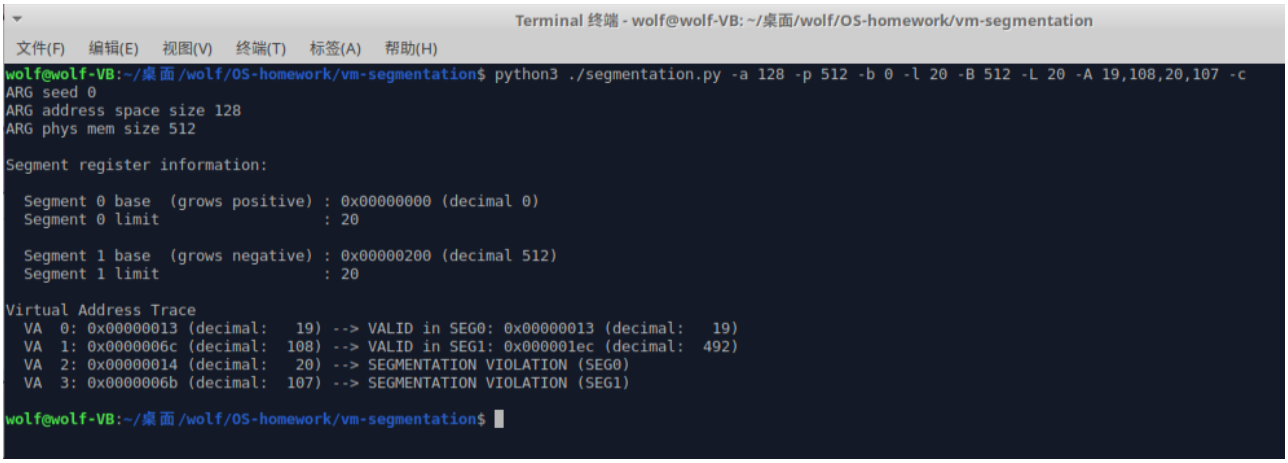
- 段0的最高合法虚拟地址为19
- 段1中最低的合法虚拟地址为108
- 整个地址空间中最低的非法地址为20（seg0）
- 整个地址空间中最高非法地址为107（seg1）

验证如下：

指令

```
python3 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -A
19,20,107,108 -c
```

运行截图如下：



16.3

> 假设我们在一个*128*字节的物理内存中有一个很小的*16*字节地址空间。你会设置什么样的基址和界限，以便让模拟器为指定的地址流生成以下转换结果：有效，有效，……，违规，违规，有效，有效？
>
> 指令为
>
> ```
> segmentation.py -a 16 -p 128 -A
> 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 --b0 ? --l0 ? --b1 ? --
> l1 ?
> ```

由题意，只要前两个和末尾两个是有效的，很容易想到两个界限各取2，由于本题的模拟器只提供两个段，一个从前往后，另一个从后往前，故界限都取2就好。

基址其实是无所谓的，如果我们取了0和16，就能保证虚拟地址等于物理地址，但不这样取也没关系。

指令：

```
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 16 --l1 2
```

验证如下：

```
wolf@wolf-VB:~/桌面/wolf/OS-homework/vm-segmentation$ python3 ./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 16 --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) : 0x00000010 (decimal 16)
  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: 0x0000000e (decimal:   14)
  VA 15: 0x0000000f (decimal:   15) --> VALID in SEG1: 0x0000000f (decimal:   15)

wolf@wolf-VB:~/桌面/wolf/OS-homework/vm-segmentation$
```

第17章

运行程序OS-homework/vm-freespace/malloc.py

17.1

> 首先运行 *flag -n 10 -H 0 -p BEST -s 0* 来产生一些随机分配和释放。你能预测 *malloc()/free()*会返回什么吗？你可以在每次请求后猜测空闲列表的状态吗？随着时间的推移，你对空闲列表有什么发现？

运行程序截图如下：

```
wolf@wolf-VB:~/桌面/wolf/OS-homework/vm-freespace$ python3 ./malloc.py flag -n 10 -H 0 -p BEST -s 0
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute False

ptr[0] = Alloc(3) returned ?
List?

Free(ptr[0])
returned ?
List?

ptr[1] = Alloc(5) returned ?
List?

Free(ptr[1])
returned ?
List?

ptr[2] = Alloc(8) returned ?
List?

Free(ptr[2])
returned ?
List?

ptr[3] = Alloc(8) returned ?
List?

Free(ptr[3])
returned ?
List?

ptr[4] = Alloc(2) returned ?
List?

ptr[5] = Alloc(7) returned ?
List?

wolf@wolf-VB:~/桌面/wolf/OS-homework/vm-freespace$
```

这是使用的BEST Fit的算法进行分配，接下来对其进行模拟。

```
//格式：指令，返回值，搜索次数
//空闲链表格式：Free List [ 大小 ]: [ 地址，大小 ]
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 ]
```

6次分配空间查找空闲块的次数分别为：1,2,3,4,4,4。

随着时间推移，我的发现：

随着不断分配和释放空间，空闲列表中的空闲块逐渐增加，且都为分隔后的较小的块，且分配搜索空闲块的次数也随之增加。

17.3

> 如果使用首次匹配（*-p FIRST*）会如何？使用首次匹配时，什么变快了？

运行截图如下：

```
wolf@wolf-VB:~/桌面/wolf/OS-homework/vm-freespace$ python3 ./malloc.py flag -n 10 -H 0 -p FIRST -s 0
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy FIRST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute False

ptr[0] = Alloc(3) returned ?
List?

Free(ptr[0])
returned ?
List?

ptr[1] = Alloc(5) returned ?
List?

Free(ptr[1])
returned ?
List?

ptr[2] = Alloc(8) returned ?
List?

Free(ptr[2])
returned ?
List?

ptr[3] = Alloc(8) returned ?
List?

Free(ptr[3])
returned ?
List?

ptr[4] = Alloc(2) returned ?
List?

ptr[5] = Alloc(7) returned ?
List?

wolf@wolf-VB:~/桌面/wolf/OS-homework/vm-freespace$
```

模拟结果如下：

```
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 ]
```

解答：

使用首次匹配，空间分配情况和空闲列表的情况与1中相同，但在搜索空闲空间时不需要全部遍历完，因此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。

17.4

> 对于上述问题，列表在保持有序时，可能会影响某些策略找到空闲位置所需的时间。使用不同的空闲列表排序（*-l ADDRSORT*，*-l SIZESORT+*，*-l SIZESORT-*）查看策略和列表排序如何相互影响。

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。

下面将分别展示三种算法的搜索次数并总结

这是ADDRSORT排序算法的结果。

```
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 ]
```

这是SIZESORT +排序算法的结果。

```
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 ]
```

这是SIZESORT -排序算法的结果。

```
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:1003 sz:97 ][ addr:1000 sz:3 ]

ptr[1] = Alloc(5) returned 1003 (searched 1 elements)
Free List [ Size 2 ]: [ addr:1008 sz:92 ][ addr:1000 sz:3 ]

Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1008 sz:92 ][ addr:1003 sz:5 ][
addr:1000 sz:3 ]

ptr[2] = Alloc(8) returned 1008 (searched 1 elements)
Free List [ Size 3 ]: [ addr:1016 sz:84 ][ addr:1003 sz:5 ][
addr:1000 sz:3 ]

Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1016 sz:84 ][ addr:1008 sz:8 ][
addr:1003 sz:5 ][ addr:1000 sz:3 ]

ptr[3] = Alloc(8) returned 1016 (searched 1 elements)
Free List [ Size 4 ]: [ addr:1024 sz:76 ][ addr:1008 sz:8 ][
addr:1003 sz:5 ][ addr:1000 sz:3 ]

Free(ptr[3])
returned 0
Free List [ Size 5 ]: [ addr:1024 sz:76 ][ addr:1008 sz:8 ][
addr:1016 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]

ptr[4] = Alloc(2) returned 1024 (searched 1 elements)
Free List [ Size 5 ]: [ addr:1026 sz:74 ][ addr:1008 sz:8 ][
addr:1016 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]

ptr[5] = Alloc(7) returned 1026 (searched 1 elements)
Free List [ Size 5 ]: [ addr:1033 sz:67 ][ addr:1008 sz:8 ][
addr:1016 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]
```

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

可见SIZESORT-搜索次数显著减少。

理由其实显而易见，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。

第18章

运行程序OS-homework/vm-paging/paging-linear-translate.py

18.1

根据题中所给参数计算线性页表大小和不同情况下的变化

首先，要理解线性页表如何随着地址空间的增长而变化：

```
paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
paging-linear-translate.py -P 1k -a 2m -p 512m -v -n 0
paging-linear-translate.py -P 1k -a 4m -p 512m -v -n 0
```

然后，理解线性页面大小如何随页大小的增长而变化：

```
paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
paging-linear-translate.py -P 2k -a 1m -p 512m -v -n 0
paging-linear-translate.py -P 4k -a 1m -p 512m -v -n 0
```

运行结果过长，故略去不放。

查阅README文件可以看到这些options的含义如下

```
Options:
-h, --help              show this help message and exit
-s SEED, --seed=SEED    the random seed
-a ASIZE, --asize=ASIZE
                        address space size (e.g., 16, 64k, ...)
-p PSIZE, --physmem=PSIZE
                        physical memory size (e.g., 16, 64k, ...)
-P PAGESIZE, --pagesize=PAGESIZE
                        page size (e.g., 4k, 8k, ...)
-n NUM, --addresses=NUM number of virtual addresses to generate
-u USED, --used=USED    percent of address space that is used
-v                      verbose mode
-c                      compute answers for me
```

根据options分别获取页大小，地址空间大小与物理地址空间大小，并分别计算页表项。

故答案如下：

对于第一组数据：

页大小为1kb，地址空间大小分别为1mb，2mb，4mb，物理地址空间大小为512mb，页表项应分别有1mb/1kb=1024，2mb/1kb=2048，4mb/1kb=4096。假设每个页表项需要4字节的空间，页表的大小分别为4kb，8kb，16kb。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。

对于第二组数据：

页大小分别为1kb，2kb，4kb，地址空间大小为1mb，物理地址空间大小为512mb，页表项分别有1mb/1kb=1024，1mb/2kb=512，1mb/4kb=256。假设每个页表项需要4字节的空间，页表的大小分别为4kb，2kb，1kb。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，因此不使用很大的页。

18.2

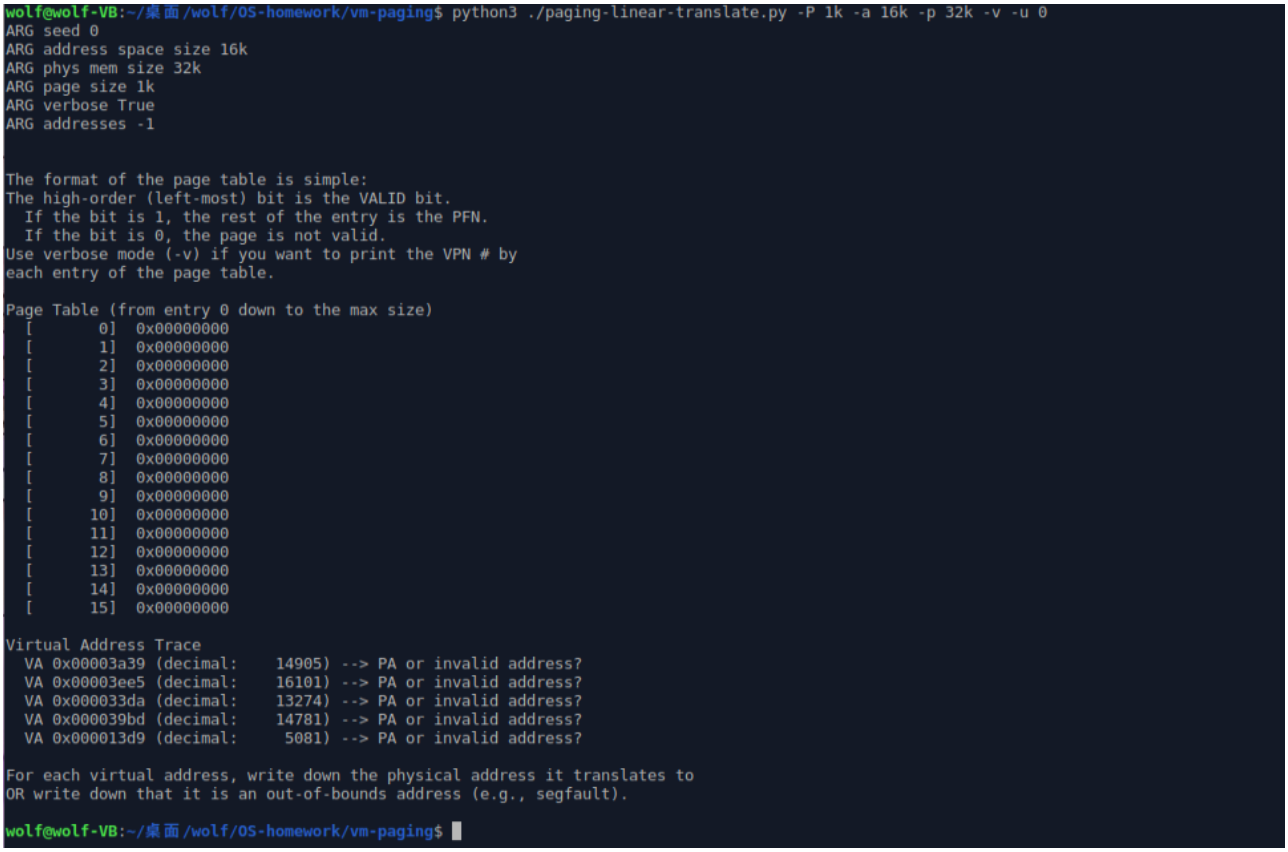
现在让我们做一些地址转换。从一些小例子开始，使用-u标志更改分配给地址空间的页数。例如：

```
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 0
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 25
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 50
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 75
paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 100
```

如果增加每个地址空间中的页的百分比，会发生什么？

①分配**0**

运行截图如下：



地址转换：

```
Virtual Address Trace
  VA 0x00003a39 (decimal:    14905) -->  Invalid (VPN 14 not valid)
  VA 0x00003ee5 (decimal:    16101) -->  Invalid (VPN 15 not valid)
  VA 0x000033da (decimal:    13274) -->  Invalid (VPN 12 not valid)
  VA 0x000039bd (decimal:    14781) -->  Invalid (VPN 14 not valid)
  VA 0x000013d9 (decimal:     5081) -->  Invalid (VPN 4 not valid)
```

②分配**25**

运行截图如下：

```
wolf@wolf-VB:~/桌面/wolf/OS-homework/vm-paging$ python3 ./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]  0x80000018
  [       1]  0x00000000
  [       2]  0x00000000
  [       3]  0x00000000
  [       4]  0x00000000
  [       5]  0x80000009
  [       6]  0x00000000
  [       7]  0x00000000
  [       8]  0x80000010
  [       9]  0x00000000
  [      10]  0x80000013
  [      11]  0x00000000
  [      12]  0x8000001f
  [      13]  0x8000001c
  [      14]  0x00000000
  [      15]  0x00000000

Virtual Address Trace
  VA 0x00003986 (decimal:    14726) --> PA or invalid address?
  VA 0x00002bc6 (decimal:    11206) --> PA or invalid address?
  VA 0x00001e37 (decimal:     7735) --> PA or invalid address?
  VA 0x00000671 (decimal:     1649) --> PA or invalid address?
  VA 0x00001bc9 (decimal:     7113) --> PA or invalid address?

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).

wolf@wolf-VB:~/桌面/wolf/OS-homework/vm-paging$
```

地址转换：

```
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)
```

③分配**50**

运行截图如下：

```
wolf@wolf-VB:~/桌面/wolf/OS-homework/vm-paging$ python3 ./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]  0x00000000
  [       2]  0x00000000
  [       3]  0x8000000c
  [       4]  0x80000009
  [       5]  0x00000000
  [       6]  0x8000001d
  [       7]  0x80000013
  [       8]  0x00000000
  [       9]  0x8000001f
  [      10]  0x8000001c
  [      11]  0x00000000
  [      12]  0x8000000f
  [      13]  0x00000000
  [      14]  0x00000000
  [      15]  0x80000008

Virtual Address Trace
  VA 0x00003385 (decimal:    13189) --> PA or invalid address?
  VA 0x0000231d (decimal:     8989) --> PA or invalid address?
  VA 0x000000e6 (decimal:      230) --> PA or invalid address?
  VA 0x00002e0f (decimal:    11791) --> PA or invalid address?
  VA 0x00001986 (decimal:     6534) --> PA or invalid address?

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).

wolf@wolf-VB:~/桌面/wolf/OS-homework/vm-paging$
```

地址转换：

```
Virtual Address Trace
  VA 0x00003385 (decimal:    13189) --> 00003f85 (decimal    16261)
[VPN 12]
  VA 0x0000231d (decimal:     8989) -->  Invalid (VPN 8 not valid)
  VA 0x000000e6 (decimal:      230) --> 000060e6 (decimal    24806)
[VPN 0]
  VA 0x00002e0f (decimal:    11791) -->  Invalid (VPN 11 not valid)
  VA 0x00001986 (decimal:     6534) --> 00007586 (decimal    30086)
[VPN 6]
```

④分配**75**

运行截图如下：

```
wolf@wolf-VB:~/桌面/wolf/OS-homework/vm-paging$ python3 ./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

Virtual Address Trace
  VA 0x00002e0f (decimal:    11791) --> PA or invalid address?
  VA 0x00001986 (decimal:     6534) --> PA or invalid address?
  VA 0x000034ca (decimal:    13514) --> PA or invalid address?
  VA 0x00002ac3 (decimal:    10947) --> PA or invalid address?
  VA 0x00000012 (decimal:       18) --> PA or invalid address?

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).

wolf@wolf-VB:~/桌面/wolf/OS-homework/vm-paging$
```

地址转换：

```
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]
```

⑤分配**100**

运行截图如下：

```
wolf@wolf-VB:~/桌面/wolf/OS-homework/vm-paging$ python3 ./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]  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

Virtual Address Trace
  VA 0x00002e0f (decimal:    11791) --> PA or invalid address?
  VA 0x00001986 (decimal:     6534) --> PA or invalid address?
  VA 0x000034ca (decimal:    13514) --> PA or invalid address?
  VA 0x00002ac3 (decimal:    10947) --> PA or invalid address?
  VA 0x00000012 (decimal:       18) --> PA or invalid address?

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).

wolf@wolf-VB:~/桌面/wolf/OS-homework/vm-paging$
```

地址转换：

```
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]
```

总结：

增加每个地址空间中的页的百分比，有效地址将更多。

18.3

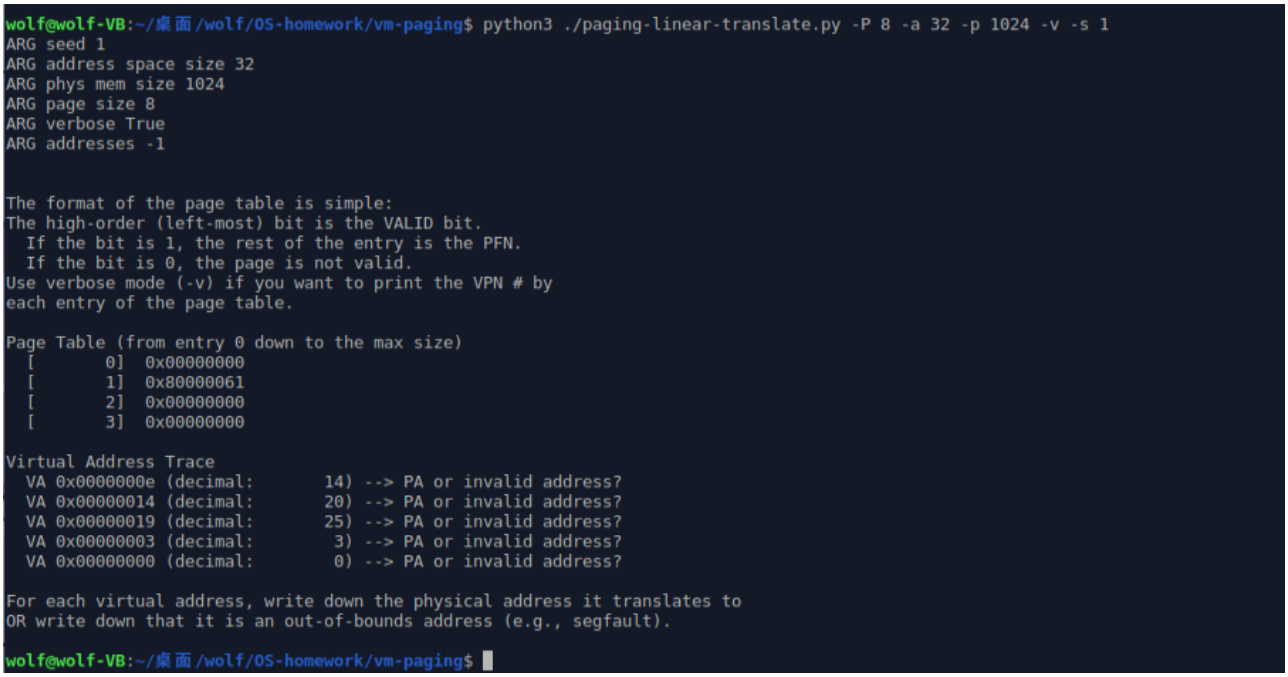
现在让我们尝试一些不同的随机种子，以及一些不同的（有时相当疯狂的）地址空间参数：

```
paging-linear-translate.py -P 8 -a 32 -p 1024 -v -s 1
paging-linear-translate.py -P 8k -a 32k -p 1m -v -s 2
paging-linear-translate.py -P 1m -a 256m -p 512m -v -s 3
```

哪些参数组合是不现实的，为什么？

①第一组

运行截图如下：



页的大小为8，地址空间32，物理地址空间大小为1024，页数为4。

虚拟地址共5位，前2位用来表示页号，后3位用来表示偏移量。

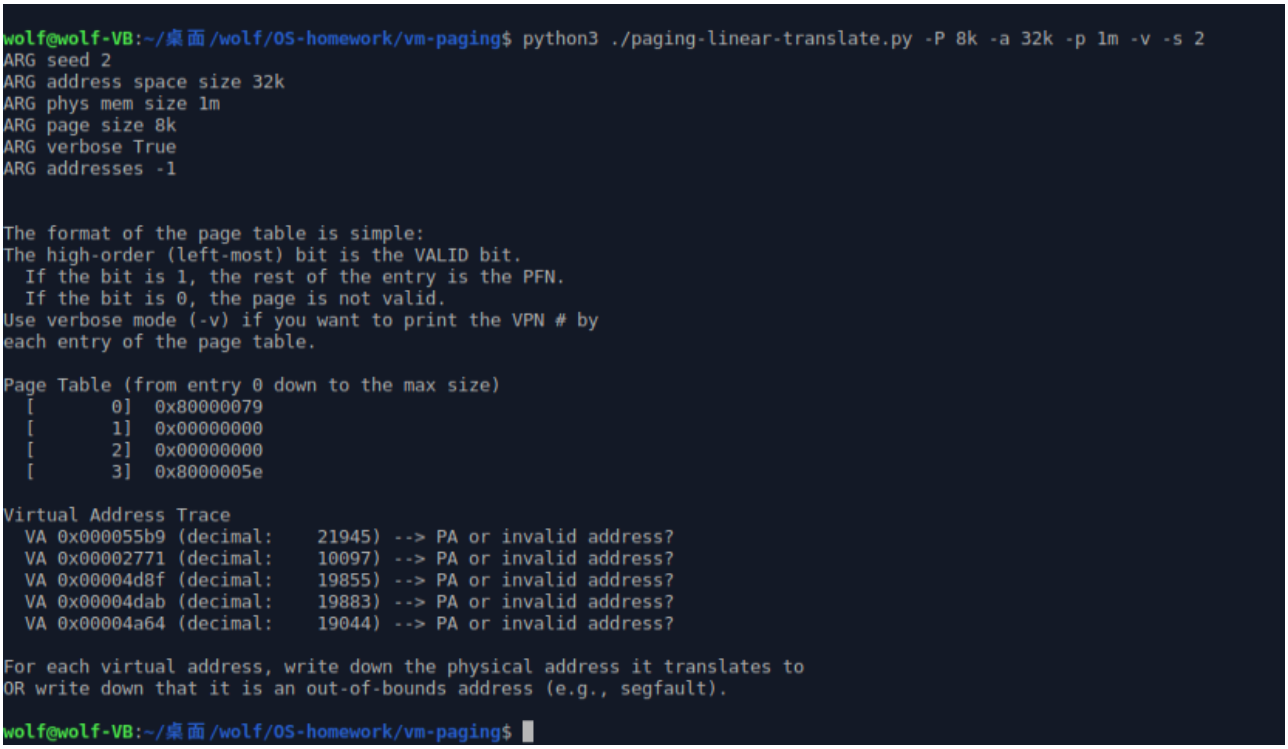
地址转换：

```
Virtual Address Trace
  VA 0x0000000e (decimal:       14) --> 0000030e (decimal      782)
[VPN 1]
  VA 0x00000014 (decimal:       20) -->  Invalid (VPN 2 not valid)
  VA 0x00000019 (decimal:       25) -->  Invalid (VPN 3 not valid)
  VA 0x00000003 (decimal:        3) -->  Invalid (VPN 0 not valid)
  VA 0x00000000 (decimal:        0) -->  Invalid (VPN 0 not valid)
```

★这样的参数组合不合理，地址空间太小了。不仅地址空间太小，一个页内的大小也太小，存不下东西。

②第二组

运行截图如下：



页的大小为8k，地址空间32k，物理地址空间大小为1m，页数为4。

虚拟地址共15位，前2位用来表示页号，后13位用来表示偏移量。

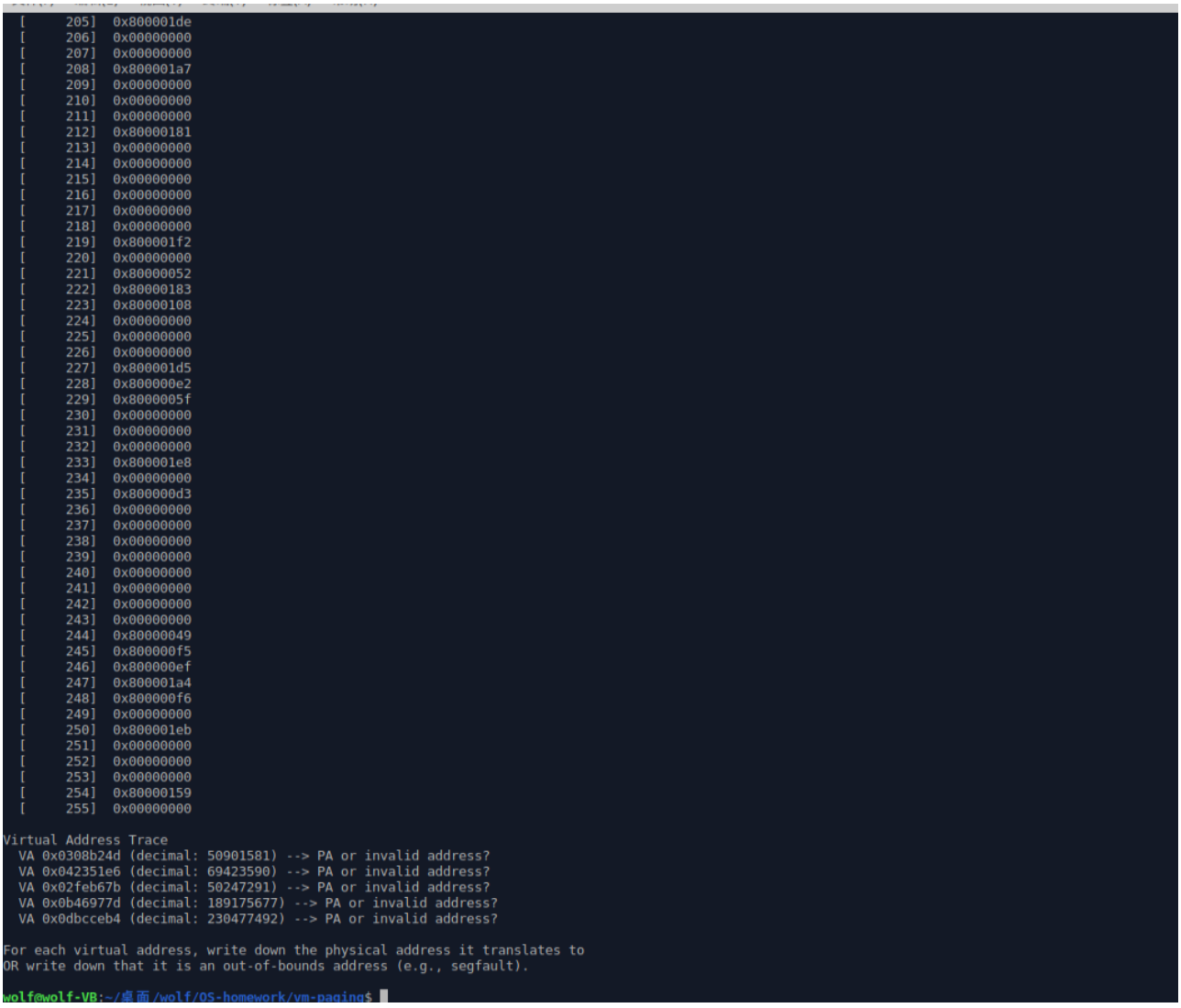
地址转换：

```
Virtual Address Trace
  VA 0x000055b9 (decimal:    21945) -->  Invalid (VPN 2 not valid)
  VA 0x00002771 (decimal:    10097) -->  Invalid (VPN 1 not valid)
  VA 0x00004d8f (decimal:    19855) -->  Invalid (VPN 2 not valid)
  VA 0x00004dab (decimal:    19883) -->  Invalid (VPN 2 not valid)
  VA 0x00004a64 (decimal:    19044) -->  Invalid (VPN 2 not valid)
```

★这样的组合不合理，地址空间太小了。在此之外，相对来说页数也太少了，这样分配会导致页的浪费。

③第三组

运行截图如下：



页的大小为1m，地址空间256m，物理地址空间大小为512m，页数为256。

虚拟地址共28位，前8位用来表示页号，后20位用来表示偏移量。

地址转换：

```
Virtual Address Trace
  VA 0x0308b24d (decimal: 50901581) --> 1f68b24d (decimal
526955085) [VPN 48]
  VA 0x042351e6 (decimal: 69423590) -->  Invalid (VPN 66 not valid)
  VA 0x02feb67b (decimal: 50247291) --> 0a9eb67b (decimal
178173563) [VPN 47]
  VA 0x0b46977d (decimal: 189175677) -->  Invalid (VPN 180 not
valid)
  VA 0x0dbcceb4 (decimal: 230477492) --> 1f2cceb4 (decimal
523030196) [VPN 219]
```

★肉眼可见有很多页。如果考虑物理地址空间，这个组合是不够合理的。因为地址空间太大了，物理地址空间只够共给两个地址空间。但是如果不考虑物理地址空间，这个组合还是有可取之处，因为与之前的两道题目相比，页的大小与页数的比例相对较合适了一些。

第20章

运行程序OS-homework/vm-smalltables/paging-multilevel-translate.py

20.1

> 对于线性页表，你需要一个寄存器来定位页表，假设硬件在*TLB* 未命中时进行查找。你需要多少个寄存器才能找到两级页表？三级页表呢？

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。

20.2

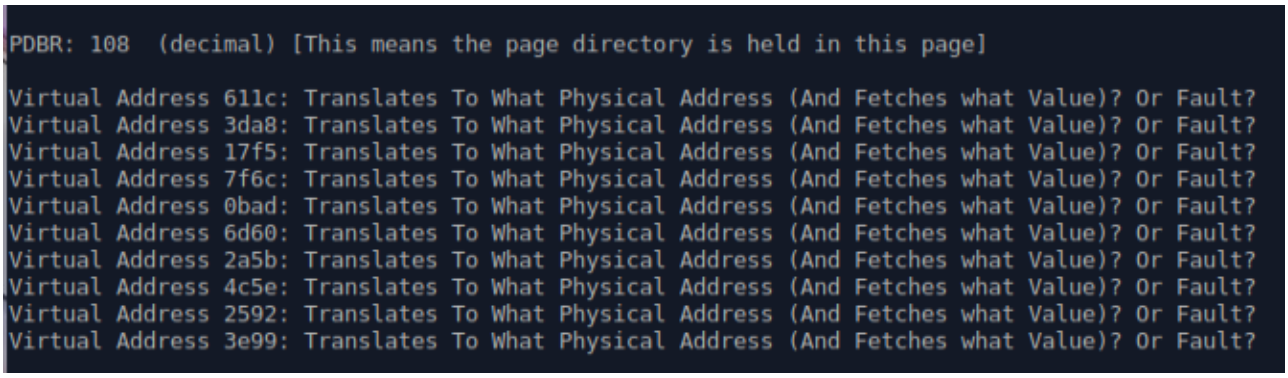
> 使用模拟器对随机种子*0*、*1* 和*2* 执行翻译，并使用*-c* 标志检查你的答案。需要多少内存引用来执行每次查找？

运行如下指令：

```
python3 paging-multilevel-translate.py -s 0
python3 paging-multilevel-translate.py -s 1
python3 paging-multilevel-translate.py -s 2
```

①种子**0**

运行截图如下：



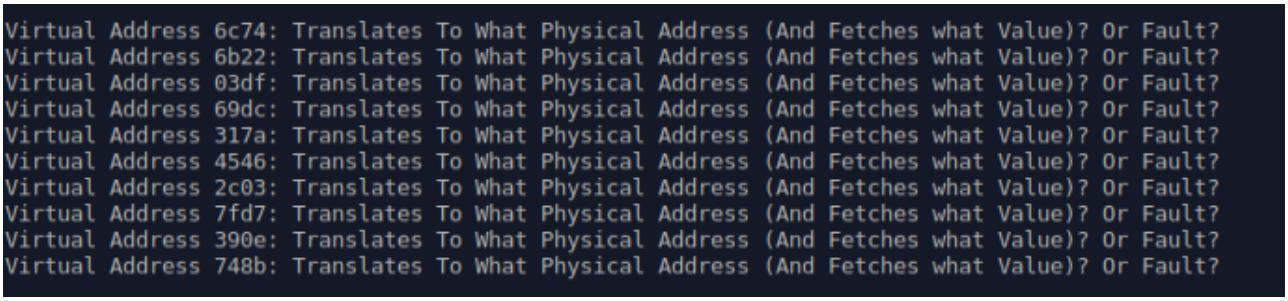
具体访存：

```
Virtual Address 0x611c:
  --> pde index:0x18 [decimal 24] pde contents:0xa1 (valid 1, pfn
0x21 [decimal 33])
    --> pte index:0x8 [decimal 8] pte contents:0xb5 (valid 1, pfn
0x35 [decimal 53])
      --> Translates to Physical Address 0x6bc --> Value: 0x08
Virtual Address 0x3da8:
  --> pde index:0xf [decimal 15] pde contents:0xd6 (valid 1, pfn
0x56 [decimal 86])
    --> pte index:0xd [decimal 13] pte contents:0x7f (valid 0, pfn
0x7f [decimal 127])
      --> Fault (page table entry not valid)
Virtual Address 0x17f5:
  --> pde index:0x5 [decimal 5] pde contents:0xd4 (valid 1, pfn
0x54 [decimal 84])
    --> pte index:0x1f [decimal 31] pte contents:0xce (valid 1, pfn
0x4e [decimal 78])
      --> Translates to Physical Address 0x9d5 --> Value: 0x1c
Virtual Address 0x7f6c:
  --> pde index:0x1f [decimal 31] pde contents:0xff (valid 1, pfn
0x7f [decimal 127])
    --> pte index:0x1b [decimal 27] pte contents:0x7f (valid 0, pfn
0x7f [decimal 127])
```

```
      --> Fault (page table entry not valid)
Virtual Address 0x0bad:
  --> pde index:0x2 [decimal 2] pde contents:0xe0 (valid 1, pfn
0x60 [decimal 96])
    --> pte index:0x1d [decimal 29] pte contents:0x7f (valid 0, pfn
0x7f [decimal 127])
      --> Fault (page table entry not valid)
Virtual Address 0x6d60:
  --> pde index:0x1b [decimal 27] pde contents:0xc2 (valid 1, pfn
0x42 [decimal 66])
    --> pte index:0xb [decimal 11] pte contents:0x7f (valid 0, pfn
0x7f [decimal 127])
      --> Fault (page table entry not valid)
Virtual Address 0x2a5b:
  --> pde index:0xa [decimal 10] pde contents:0xd5 (valid 1, pfn
0x55 [decimal 85])
    --> pte index:0x12 [decimal 18] pte contents:0x7f (valid 0, pfn
0x7f [decimal 127])
      --> Fault (page table entry not valid)
Virtual Address 0x4c5e:
  --> pde index:0x13 [decimal 19] pde contents:0xf8 (valid 1, pfn
0x78 [decimal 120])
    --> pte index:0x2 [decimal 2] pte contents:0x7f (valid 0, pfn
0x7f [decimal 127])
      --> Fault (page table entry not valid)
Virtual Address 0x2592:
  --> pde index:0x9 [decimal 9] pde contents:0x9e (valid 1, pfn
0x1e [decimal 30])
    --> pte index:0xc [decimal 12] pte contents:0xbd (valid 1, pfn
0x3d [decimal 61])
      --> Translates to Physical Address 0x7b2 --> Value: 0x1b
Virtual Address 0x3e99:
  --> pde index:0xf [decimal 15] pde contents:0xd6 (valid 1, pfn
0x56 [decimal 86])
    --> pte index:0x14 [decimal 20] pte contents:0xca (valid 1, pfn
0x4a [decimal 74])
      --> Translates to Physical Address 0x959 --> Value: 0x1e
```

②种子**1**

运行截图如下：



具体访存：

```
Virtual Address 0x6c74:
  --> pde index:0x1b [decimal 27] pde contents:0xa0 (valid 1, pfn
0x20 [decimal 32])
    --> pte index:0x3 [decimal 3] pte contents:0xe1 (valid 1, pfn
0x61 [decimal 97])
      --> Translates to Physical Address 0xc34 --> Value: 0x06
Virtual Address 0x6b22:
  --> pde index:0x1a [decimal 26] pde contents:0xd2 (valid 1, pfn
0x52 [decimal 82])
    --> pte index:0x19 [decimal 25] pte contents:0xc7 (valid 1, pfn
0x47 [decimal 71])
      --> Translates to Physical Address 0x8e2 --> Value: 0x1a
Virtual Address 0x03df:
  --> pde index:0x0 [decimal 0] pde contents:0xda (valid 1, pfn
0x5a [decimal 90])
    --> pte index:0x1e [decimal 30] pte contents:0x85 (valid 1, pfn
0x05 [decimal 5])
      --> Translates to Physical Address 0x0bf --> Value: 0x0f
Virtual Address 0x69dc:
  --> pde index:0x1a [decimal 26] pde contents:0xd2 (valid 1, pfn
0x52 [decimal 82])
    --> pte index:0xe [decimal 14] pte contents:0x7f (valid 0, pfn
0x7f [decimal 127])
      --> Fault (page table entry not valid)
Virtual Address 0x317a:
  --> pde index:0xc [decimal 12] pde contents:0x98 (valid 1, pfn
0x18 [decimal 24])
    --> pte index:0xb [decimal 11] pte contents:0xb5 (valid 1, pfn
0x35 [decimal 53])
```

```
      --> Translates to Physical Address 0x6ba --> Value: 0x1e
Virtual Address 0x4546:
  --> pde index:0x11 [decimal 17] pde contents:0xa1 (valid 1, pfn
0x21 [decimal 33])
    --> pte index:0xa [decimal 10] pte contents:0x7f (valid 0, pfn
0x7f [decimal 127])
      --> Fault (page table entry not valid)
Virtual Address 0x2c03:
  --> pde index:0xb [decimal 11] pde contents:0xc4 (valid 1, pfn
0x44 [decimal 68])
    --> pte index:0x0 [decimal 0] pte contents:0xd7 (valid 1, pfn
0x57 [decimal 87])
      --> Translates to Physical Address 0xae3 --> Value: 0x16
Virtual Address 0x7fd7:
  --> pde index:0x1f [decimal 31] pde contents:0x92 (valid 1, pfn
0x12 [decimal 18])
    --> pte index:0x1e [decimal 30] pte contents:0x7f (valid 0, pfn
0x7f [decimal 127])
      --> Fault (page table entry not valid)
Virtual Address 0x390e:
  --> pde index:0xe [decimal 14] pde contents:0x7f (valid 0, pfn
0x7f [decimal 127])
      --> Fault (page directory entry not valid)
Virtual Address 0x748b:
  --> pde index:0x1d [decimal 29] pde contents:0x80 (valid 1, pfn
0x00 [decimal 0])
    --> pte index:0x4 [decimal 4] pte contents:0x7f (valid 0, pfn
0x7f [decimal 127])
      --> Fault (page table entry not valid)
```

③种子**2**

运行截图如下：

```
PDBR: 122  (decimal) [This means the page directory is held in this page]

Virtual Address 7570: Translates To What Physical Address (And Fetches what Value)? Or Fault?
Virtual Address 7268: Translates To What Physical Address (And Fetches what Value)? Or Fault?
Virtual Address 1f9f: Translates To What Physical Address (And Fetches what Value)? Or Fault?
Virtual Address 0325: Translates To What Physical Address (And Fetches what Value)? Or Fault?
Virtual Address 64c4: Translates To What Physical Address (And Fetches what Value)? Or Fault?
Virtual Address 0cdf: Translates To What Physical Address (And Fetches what Value)? Or Fault?
Virtual Address 2906: Translates To What Physical Address (And Fetches what Value)? Or Fault?
Virtual Address 7a36: Translates To What Physical Address (And Fetches what Value)? Or Fault?
Virtual Address 21e1: Translates To What Physical Address (And Fetches what Value)? Or Fault?
Virtual Address 5149: Translates To What Physical Address (And Fetches what Value)? Or Fault?
```

具体访存：

```
Virtual Address 0x7570:
  --> pde index:0x1d [decimal 29] pde contents:0xb3 (valid 1, pfn
0x33 [decimal 51])
    --> pte index:0xb [decimal 11] pte contents:0x7f (valid 0, pfn
0x7f [decimal 127])
      --> Fault (page table entry not valid)
Virtual Address 0x7268:
  --> pde index:0x1c [decimal 28] pde contents:0xde (valid 1, pfn
0x5e [decimal 94])
    --> pte index:0x13 [decimal 19] pte contents:0xe5 (valid 1, pfn
0x65 [decimal 101])
      --> Translates to Physical Address 0xca8 --> Value: 0x16
Virtual Address 0x1f9f:
  --> pde index:0x7 [decimal 7] pde contents:0xaf (valid 1, pfn
0x2f [decimal 47])
    --> pte index:0x1c [decimal 28] pte contents:0x7f (valid 0, pfn
0x7f [decimal 127])
      --> Fault (page table entry not valid)
Virtual Address 0x0325:
  --> pde index:0x0 [decimal 0] pde contents:0x82 (valid 1, pfn
0x02 [decimal 2])
    --> pte index:0x19 [decimal 25] pte contents:0xdd (valid 1, pfn
0x5d [decimal 93])
      --> Translates to Physical Address 0xba5 --> Value: 0x0b
Virtual Address 0x64c4:
  --> pde index:0x19 [decimal 25] pde contents:0xb8 (valid 1, pfn
0x38 [decimal 56])
    --> pte index:0x6 [decimal 6] pte contents:0x7f (valid 0, pfn
0x7f [decimal 127])
      --> Fault (page table entry not valid)
Virtual Address 0x0cdf:
  --> pde index:0x3 [decimal 3] pde contents:0x9d (valid 1, pfn
0x1d [decimal 29])
    --> pte index:0x6 [decimal 6] pte contents:0x97 (valid 1, pfn
0x17 [decimal 23])
      --> Translates to Physical Address 0x2ff --> Value: 0x00
Virtual Address 0x2906:
  --> pde index:0xa [decimal 10] pde contents:0x7f (valid 0, pfn
0x7f [decimal 127])
      --> Fault (page directory entry not valid)
```

```
Virtual Address 0x7a36:
  --> pde index:0x1e [decimal 30] pde contents:0x8a (valid 1, pfn
0x0a [decimal 10])
    --> pte index:0x11 [decimal 17] pte contents:0xe6 (valid 1, pfn
0x66 [decimal 102])
      --> Translates to Physical Address 0xcd6 --> Value: 0x09
Virtual Address 0x21e1:
  --> pde index:0x8 [decimal 8] pde contents:0x7f (valid 0, pfn
0x7f [decimal 127])
      --> Fault (page directory entry not valid)
Virtual Address 0x5149:
  --> pde index:0x14 [decimal 20] pde contents:0xbb (valid 1, pfn
0x3b [decimal 59])
    --> pte index:0xa [decimal 10] pte contents:0x81 (valid 1, pfn
0x01 [decimal 1])
      --> Translates to Physical Address 0x029 --> Value: 0x1b
```

20.3

> 根据你对缓存内存的工作原理的理解，你认为对页表的内存引用如何在缓存中工作？它们是否会导致大量的缓存命中（并导致快速访问）或者很多未命中（并导致访问缓慢）？

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。利用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。

除此之外，缓存的替换算法和访问的行为模式也会造成一定的影响。

第22章

运行程序OS-homework/vm-beyondphys-policy/paging-policy.py

22.1

> 使用以下参数生成随机地址：*-s 0 -n 10*，*-s 1 -n 10* 和*-s 2 -n 10*。将策略从*FIFO* 更改为*LRU*，并将其更改为*OPT*。计算所述地址追踪中的每个访问是否命中或未命中。

①种子**0**

(1)FIFO

使用指令

```
python3 paging-policy.py -s 0 -n 10 -p FIFO
```

得到结果

```
Access: 8  MISS FirstIn ->          [8] <- Lastin  Replaced:-
[Hits:0 Misses:1]
Access: 7  MISS FirstIn ->       [8, 7] <- Lastin  Replaced:-
[Hits:0 Misses:2]
Access: 4  MISS FirstIn ->    [8, 7, 4] <- Lastin  Replaced:-
[Hits:0 Misses:3]
Access: 2  MISS FirstIn ->    [7, 4, 2] <- Lastin  Replaced:8
[Hits:0 Misses:4]
Access: 5  MISS FirstIn ->    [4, 2, 5] <- Lastin  Replaced:7
[Hits:0 Misses:5]
Access: 4  HIT  FirstIn ->    [4, 2, 5] <- Lastin  Replaced:-
[Hits:1 Misses:5]
Access: 7  MISS FirstIn ->    [2, 5, 7] <- Lastin  Replaced:4
[Hits:1 Misses:6]
Access: 3  MISS FirstIn ->    [5, 7, 3] <- Lastin  Replaced:2
[Hits:1 Misses:7]
Access: 4  MISS FirstIn ->    [7, 3, 4] <- Lastin  Replaced:5
[Hits:1 Misses:8]
Access: 5  MISS FirstIn ->    [3, 4, 5] <- Lastin  Replaced:7
[Hits:1 Misses:9]

FINALSTATS hits 1   misses 9   hitrate 10.00
```

(2)LRU

使用指令

```
python3 paging-policy.py -s 0 -n 10 -p LRU
```

得到结果

```
Access: 8  MISS LRU ->          [8] <- MRU Replaced:- [Hits:0
Misses:1]
Access: 7  MISS LRU ->       [8, 7] <- MRU Replaced:- [Hits:0
Misses:2]
Access: 4  MISS LRU ->    [8, 7, 4] <- MRU Replaced:- [Hits:0
Misses:3]
Access: 2  MISS LRU ->    [7, 4, 2] <- MRU Replaced:8 [Hits:0
Misses:4]
Access: 5  MISS LRU ->    [4, 2, 5] <- MRU Replaced:7 [Hits:0
Misses:5]
Access: 4  HIT  LRU ->    [2, 5, 4] <- MRU Replaced:- [Hits:1
Misses:5]
Access: 7  MISS LRU ->    [5, 4, 7] <- MRU Replaced:2 [Hits:1
Misses:6]
Access: 3  MISS LRU ->    [4, 7, 3] <- MRU Replaced:5 [Hits:1
Misses:7]
Access: 4  HIT  LRU ->    [7, 3, 4] <- MRU Replaced:- [Hits:2
Misses:7]
Access: 5  MISS LRU ->    [3, 4, 5] <- MRU Replaced:7 [Hits:2
Misses:8]

FINALSTATS hits 2   misses 8   hitrate 20.00
```

(3)OPT

使用指令

```
python3 paging-policy.py -s 0 -n 10 -p OPT
```

得到结果

```
Access: 8  MISS Left  ->          [8] <- Right Replaced:- [Hits:0
Misses:1]
Access: 7  MISS Left  ->       [8, 7] <- Right Replaced:- [Hits:0
Misses:2]
Access: 4  MISS Left  ->    [8, 7, 4] <- Right Replaced:- [Hits:0
Misses:3]
Access: 2  MISS Left  ->    [7, 4, 2] <- Right Replaced:8 [Hits:0
Misses:4]
Access: 5  MISS Left  ->    [7, 4, 5] <- Right Replaced:2 [Hits:0
Misses:5]
Access: 4  HIT  Left  ->    [7, 4, 5] <- Right Replaced:- [Hits:1
Misses:5]
Access: 7  HIT  Left  ->    [7, 4, 5] <- Right Replaced:- [Hits:2
Misses:5]
Access: 3  MISS Left  ->    [4, 5, 3] <- Right Replaced:7 [Hits:2
Misses:6]
Access: 4  HIT  Left  ->    [4, 5, 3] <- Right Replaced:- [Hits:3
Misses:6]
Access: 5  HIT  Left  ->    [4, 5, 3] <- Right Replaced:- [Hits:4
Misses:6]

FINALSTATS hits 4   misses 6   hitrate 40.00
```

②种子**1**

(1)FIFO

使用指令

```
python3 paging-policy.py -s 1 -n 10 -p FIFO
```

得到结果

```
Access: 1  MISS FirstIn ->          [1] <- Lastin  Replaced:-
[Hits:0 Misses:1]
Access: 8  MISS FirstIn ->       [1, 8] <- Lastin  Replaced:-
[Hits:0 Misses:2]
Access: 7  MISS FirstIn ->    [1, 8, 7] <- Lastin  Replaced:-
[Hits:0 Misses:3]
Access: 2  MISS FirstIn ->    [8, 7, 2] <- Lastin  Replaced:1
[Hits:0 Misses:4]
Access: 4  MISS FirstIn ->    [7, 2, 4] <- Lastin  Replaced:8
[Hits:0 Misses:5]
Access: 4  HIT  FirstIn ->    [7, 2, 4] <- Lastin  Replaced:-
[Hits:1 Misses:5]
Access: 6  MISS FirstIn ->    [2, 4, 6] <- Lastin  Replaced:7
[Hits:1 Misses:6]
Access: 7  MISS FirstIn ->    [4, 6, 7] <- Lastin  Replaced:2
[Hits:1 Misses:7]
Access: 0  MISS FirstIn ->    [6, 7, 0] <- Lastin  Replaced:4
[Hits:1 Misses:8]
Access: 0  HIT  FirstIn ->    [6, 7, 0] <- Lastin  Replaced:-
[Hits:2 Misses:8]

FINALSTATS hits 2   misses 8   hitrate 20.00
```

(2)LRU

使用指令

```
python3 paging-policy.py -s 1 -n 10 -p LRU
```

得到结果

```
Access: 1  MISS LRU ->          [1] <- MRU Replaced:- [Hits:0
Misses:1]
Access: 8  MISS LRU ->       [1, 8] <- MRU Replaced:- [Hits:0
Misses:2]
Access: 7  MISS LRU ->    [1, 8, 7] <- MRU Replaced:- [Hits:0
Misses:3]
Access: 2  MISS LRU ->    [8, 7, 2] <- MRU Replaced:1 [Hits:0
Misses:4]
Access: 4  MISS LRU ->    [7, 2, 4] <- MRU Replaced:8 [Hits:0
Misses:5]
Access: 4  HIT  LRU ->    [7, 2, 4] <- MRU Replaced:- [Hits:1
Misses:5]
Access: 6  MISS LRU ->    [2, 4, 6] <- MRU Replaced:7 [Hits:1
Misses:6]
Access: 7  MISS LRU ->    [4, 6, 7] <- MRU Replaced:2 [Hits:1
Misses:7]
Access: 0  MISS LRU ->    [6, 7, 0] <- MRU Replaced:4 [Hits:1
Misses:8]
Access: 0  HIT  LRU ->    [6, 7, 0] <- MRU Replaced:- [Hits:2
Misses:8]

FINALSTATS hits 2   misses 8   hitrate 20.00
```

(3)OPT

使用指令

```
python3 paging-policy.py -s 1 -n 10 -p OPT
```

得到结果

```
Access: 1  MISS Left  ->          [1] <- Right Replaced:- [Hits:0
Misses:1]
Access: 8  MISS Left  ->       [1, 8] <- Right Replaced:- [Hits:0
Misses:2]
Access: 7  MISS Left  ->    [1, 8, 7] <- Right Replaced:- [Hits:0
Misses:3]
Access: 2  MISS Left  ->    [1, 7, 2] <- Right Replaced:8 [Hits:0
Misses:4]
Access: 4  MISS Left  ->    [1, 7, 4] <- Right Replaced:2 [Hits:0
Misses:5]
Access: 4  HIT  Left  ->    [1, 7, 4] <- Right Replaced:- [Hits:1
Misses:5]
Access: 6  MISS Left  ->    [1, 7, 6] <- Right Replaced:4 [Hits:1
Misses:6]
Access: 7  HIT  Left  ->    [1, 7, 6] <- Right Replaced:- [Hits:2
Misses:6]
Access: 0  MISS Left  ->    [1, 7, 0] <- Right Replaced:6 [Hits:2
Misses:7]
Access: 0  HIT  Left  ->    [1, 7, 0] <- Right Replaced:- [Hits:3
Misses:7]

FINALSTATS hits 3   misses 7   hitrate 30.00
```

③种子**2**

(1)FIFO

使用指令

```
python3 paging-policy.py -s 2 -n 10 -p FIFO
```

得到结果

```
Access: 9  MISS FirstIn ->          [9] <- Lastin  Replaced:-
[Hits:0 Misses:1]
Access: 9  HIT  FirstIn ->          [9] <- Lastin  Replaced:-
[Hits:1 Misses:1]
Access: 0  MISS FirstIn ->       [9, 0] <- Lastin  Replaced:-
[Hits:1 Misses:2]
Access: 0  HIT  FirstIn ->       [9, 0] <- Lastin  Replaced:-
[Hits:2 Misses:2]
Access: 8  MISS FirstIn ->    [9, 0, 8] <- Lastin  Replaced:-
[Hits:2 Misses:3]
Access: 7  MISS FirstIn ->    [0, 8, 7] <- Lastin  Replaced:9
[Hits:2 Misses:4]
Access: 6  MISS FirstIn ->    [8, 7, 6] <- Lastin  Replaced:0
[Hits:2 Misses:5]
Access: 3  MISS FirstIn ->    [7, 6, 3] <- Lastin  Replaced:8
[Hits:2 Misses:6]
Access: 6  HIT  FirstIn ->    [7, 6, 3] <- Lastin  Replaced:-
[Hits:3 Misses:6]
Access: 6  HIT  FirstIn ->    [7, 6, 3] <- Lastin  Replaced:-
[Hits:4 Misses:6]

FINALSTATS hits 4   misses 6   hitrate 40.00
```

(2)LRU

使用指令

```
python3 paging-policy.py -s 2 -n 10 -p LRU
```

得到结果

```
Access: 9  MISS LRU ->          [9] <- MRU Replaced:- [Hits:0
Misses:1]
Access: 9  HIT  LRU ->          [9] <- MRU Replaced:- [Hits:1
Misses:1]
Access: 0  MISS LRU ->       [9, 0] <- MRU Replaced:- [Hits:1
Misses:2]
Access: 0  HIT  LRU ->       [9, 0] <- MRU Replaced:- [Hits:2
Misses:2]
Access: 8  MISS LRU ->    [9, 0, 8] <- MRU Replaced:- [Hits:2
Misses:3]
Access: 7  MISS LRU ->    [0, 8, 7] <- MRU Replaced:9 [Hits:2
Misses:4]
Access: 6  MISS LRU ->    [8, 7, 6] <- MRU Replaced:0 [Hits:2
Misses:5]
Access: 3  MISS LRU ->    [7, 6, 3] <- MRU Replaced:8 [Hits:2
Misses:6]
Access: 6  HIT  LRU ->    [7, 3, 6] <- MRU Replaced:- [Hits:3
Misses:6]
Access: 6  HIT  LRU ->    [7, 3, 6] <- MRU Replaced:- [Hits:4
Misses:6]

FINALSTATS hits 4   misses 6   hitrate 40.00
```

(3)OPT

使用指令

```
python3 paging-policy.py -s 2 -n 10 -p OPT
```

得到结果

```
Access: 9  MISS Left  ->          [9] <- Right Replaced:- [Hits:0
Misses:1]
Access: 9  HIT  Left  ->          [9] <- Right Replaced:- [Hits:1
Misses:1]
Access: 0  MISS Left  ->       [9, 0] <- Right Replaced:- [Hits:1
Misses:2]
Access: 0  HIT  Left  ->       [9, 0] <- Right Replaced:- [Hits:2
Misses:2]
Access: 8  MISS Left  ->    [9, 0, 8] <- Right Replaced:- [Hits:2
Misses:3]
Access: 7  MISS Left  ->    [9, 0, 7] <- Right Replaced:8 [Hits:2
Misses:4]
Access: 6  MISS Left  ->    [9, 0, 6] <- Right Replaced:7 [Hits:2
Misses:5]
Access: 3  MISS Left  ->    [9, 6, 3] <- Right Replaced:0 [Hits:2
Misses:6]
Access: 6  HIT  Left  ->    [9, 6, 3] <- Right Replaced:- [Hits:3
Misses:6]
Access: 6  HIT  Left  ->    [9, 6, 3] <- Right Replaced:- [Hits:4
Misses:6]

FINALSTATS hits 4   misses 6   hitrate 40.00
```

22.2

> 对于大小为*5* 的高速缓存，为以下每个策略生成最差情况的地址引用序列：*FIFO*、*LRU* 和*MRU*（最差情况下的引用序列导致尽可能多的未命中）。对于最差情况下的引用序列，需要的缓存增大多少，才能大幅提高性能，并接近*OPT*？

不同策略下的最差情况的地址引用序列

- FIFO：1,2,3,4,5,6
- LRU：1,2,3,4,5,6
- MRU：1,2,3,4,5,6,5,6,5,6...

对于LRU和FIFO，只要缓存大小与序列大小相同就可以提高命中率，且除了冷启动不命中外均命中。

对于MRU，如果缓存容量刚好已满，而有两页交替进行访问，只要缓存增大1就可以提高命中率，大幅提高性能。但若仍然考虑具有其他页参与的情况，则需要缓存大小与序列大小相同。

22.3

> 生成一个随机追踪序列（使用*Python* 或*Perl*）。你预计不同的策略在这样的追踪序列上的表现如何？

该序列在OPT策略（作为衡量其他算法的一个标准）中表现一般，在其他策略的表现都相当糟糕。

验证如下：

使用python编写随机生成序列的程序

```
import random
num=200
sequence=[]
for i in range(0,num):
    address=random.randint(0,9)
    sequence.append(address)
file=open("./sequence.txt","w")
for i in sequence:
    file.write(str(i)+"\n")
file.close()
```

生成序列如下：

```
2,3,9,0,2,2,4,5,9,4,2,5,6,0,8,9,8,0,0,7,9,4,4,9,7,8,5,8,0,4,5,1,3,3
,0,9,3,9,8,6,3,5,5,1,1,8,3,2,2,1,8,6,5,3,1,9,5,8,4,1,2,3,3,7,8,5,5,
5,8,7,5,5,8,4,6,9,3,8,7,5,2,9,5,9,4,6,9,3,0,5,2,2,6,3,8,6,3,4,8,2,9
,7,9,0,6,8,7,7,6,9,1,3,8,5,8,5,1,6,6,9,2,8,9,3,1,4,0,1,9,2,9,3,2,7,
5,2,7,5,5,1,7,0,2,1,3,9,4,0,3,3,4,3,0,7,1,0,8,5,7,2,4,0,4,7,7,8,4,2
,9,2,4,3,6,8,0,5,1,1,5,8,0,0,8,3,5,3,0,8,5,6,9,6,5,5,5,4,2,8,1,7
```

分别使用

```
python3 paging-policy.py -s 2 -n 10 -p OPT -c -a （序列）
python3 paging-policy.py -s 2 -n 10 -p FIFO -c -a （序列）
python3 paging-policy.py -s 2 -n 10 -p LRU -c -a （序列）
python3 paging-policy.py -s 2 -n 10 -p CLOCK -c -a （序列）
python3 paging-policy.py -s 2 -n 10 -p RAND -c -a （序列）
python3 paging-policy.py -s 2 -n 10 -p MRU -c -a （序列）
python3 paging-policy.py -s 2 -n 10 -p UNOPT -c -a （序列）
```

进行测试，结果如下：

策略	命中率
OPT	51.50%
FIFO	32.00%
LRU	31.50%
CLOCK	28.00%
RAND	32.50%
MRU	26.50%
UNOPT	10.50%

符合预期。

22.4

> 现在生成一些局部性追踪序列。如何能够产生这样的追踪序列？*LRU* 表现如何？*RAND*比*LRU* 好多少？*CLOCK* 表现如何？*CLOCK* 使用不同数量的时钟位，表现如何？

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):
        rand = [random.randint(0, numAddr), random.choice(trace)]
        trace.append(random.choice(rand))
    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
```

使用以下指令测试

```
#测试空间局部性
python3 paging-policy.py -c -N -a  $(python3 22.4.py -s) -p LRU
python3 paging-policy.py -c -N -a  $(python3 22.4.py -s) -p RAND
python3 paging-policy.py -c -N -a  $(python3 22.4.py -s) -p CLOCK
python3 paging-policy.py -c -N -a  $(python3 22.4.py -s) -p OPT
#测试时间局部性
python3 paging-policy.py -c -N -a  $(python3 22.4.py -s) -p LRU
python3 paging-policy.py -c -N -a  $(python3 22.4.py -s) -p RAND
python3 paging-policy.py -c -N -a  $(python3 22.4.py -s) -p CLOCK
python3 paging-policy.py -c -N -a  $(python3 22.4.py -s) -p OPT
```

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	29.87%	55.44%
RAND	30.57%	53.35%
CLOCK	30.47%	52.95%
OPT	51.75%	66.33%

可见，OPT效果最好。LRU在空间局部性好的序列上效果较好。RAND比LRU好不了多少。CLOCK表现差不多，但比OPT差距较大。

对于CLOCK策略，使用如下指令改变时钟位

```
python3 paging-policy.py -c -N -a $(python3 22.4.py -s) -p CLOCK -b
1
python3 paging-policy.py -c -N -a $(python3 22.4.py -s) -p CLOCK -b
2
python3 paging-policy.py -c -N -a $(python3 22.4.py -s) -p CLOCK -b
3
python3 paging-policy.py -c -N -a $(python3 22.4.py -s) -p CLOCK -b
4
python3 paging-policy.py -c -N -a $(python3 22.4.py -s) -p CLOCK -b
5
python3 paging-policy.py -c -N -a $(python3 22.4.py -s) -p CLOCK -b
6
python3 paging-policy.py -c -N -a $(python3 22.4.py -s) -p CLOCK -b
7
python3 paging-policy.py -c -N -a $(python3 22.4.py -s) -p CLOCK -b
8
python3 paging-policy.py -c -N -a $(python3 22.4.py -s) -p CLOCK -b
9
python3 paging-policy.py -c -N -a $(python3 22.4.py -s) -p CLOCK -b
10
```

为公平起见，使用同一段序列替换上述-a标志位后的序列

#空间局部性采取测试数据

8,0,1,0,8,7,7,8,8,7,7,8,7,7,6,5,4,5,4,5,5,4,4,3,3,3,4,5,4,9,9,9,0,1
0,9,5,4,4,4,5,5,6,6,4,3,8,10,3,8,3,4,4,3,3,2,0,2,2,7,7,8,7,8,7,7,8,
1,2,3,4,5,6,5,6,6,6,3,4,5,1,0,9,4,8,5,5,4,3,6,7,6,7,8,9,0,0,1,1,2,3
,4,4,4,5,6,5,4,3,10,9,0,1,2,9,0,0,0,4,4,4,3,4,10,1,2,2,5,6,9,8,4,5,
6,7,8,7,0,9,0,9,0,8,8,5,5,4,5,4,4,4,3,2,1,2,1,1,1,2,1,0,5,6,5,6,5,7
,6,9,5,6,1,1,2,3,3,8,8,8,2,1,0,9,8,8,8,8,9,8,8,7,10,1,2,1,1,2,6,6,6
,7,8,9,0,1,2,2,2,5,5,5,5,6,7,8,8,8,7,8,7,7,8,7,7,6,2,3,3,2,2,2,2,3,
2,10,1,2,1,0,1,4,1,2,1,1,0,0,1,2,2,3,3,3,3,1,0,0,9,9,6,6,7,8,9,8,7,
6,2,1,0,2,2,7,7,5,6,9,2,2,3,10,1,1,10,3,9,8,4,5,4,3,3,2,1,0,9,9,8,8
,9,8,8,8,8,8,9,8,9,0,9,0,1,0,9,9,0,0,0,9,8,9,8,7,7,7,6,6,6,7,8,5,5,
5,0,9,0,0,0,9,0,9,9,0,1,0,7,8,9,9,0,10,9,0,0,1,0,2,3,3,4,2,3,1,0,1,
1,2,3,2,3,2,3,2,1,1,9,9,8,7,8,5,6,6,3,3,2,3,7,8,7,7,3,2,1,7,2,1,2,1
,2,1,2,1,1,0,1,1,4,5,8,7,8,0,0,9,0,1,0,6,6,5,6,6,7,2,1,2,2,1,0,0,1,
1,0,0,9,0,0,9,0,0,1,0,9,0,1,0,0,1,7,6,7,7,6,5,6,6,6,6,5,5,5,6,0,1,0
,9,9,0,9,8,1,1,9,0,1,7,1,7,6,5,5,4,5,6,7,8,9,4,4,4,3,4,5,4,3,3,1,0,
1,0,1,2,3,2,2,1,0,4,5,6,3,4,9,8,9,0,0,1,1,0,0,9,8,8,5,1,2,9,9,1,1,2
,1,0,1,2,0,9,8,9,7,8,7,6,6,7,6,5,6,0,0,9,10,2,2,2,2,1,0,0,1,8,5,6,7
,7,4,5,6,7,7,7,8,8,8,9,0,8,8,3,3,3,4,7,6,7,8,9,1,2,3,3,3,4,5,2,3,4,
4,4,4,3,2,3,2,3,3,4,5,5,6,6,5,2,5,8,0,1,7,7,8,4,4,8,8,1,2,1,2,8,7,7
,10,9,8,8,8,7,8,9,1,0,9,9,0,1,2,2,2,2,2,2,7,6,5,4,0,9,9,0,8,9,0,1,8
,7,8,7,8,8,9,9,0,0,9,9,5,5,6,7,6,7,0,9,9,8,7,7,6,6,7,6,7,6,6,7,8,7,
10,1,7,8,9,8,0,9,9,8,9,0,9,0,3,4,4,3,4,3,3,2,1,0,1,0,0,7,8,8,9,0,9,
2,1,1,2,1,0,9,9,2,10,9,9,0,0,1,2,3,6,7,6,6,1,0,6,7,6,7,8,9,9,9,0,0,
9,8,7,7,7,3,4,3,6,5,10,10,9,0,1,0,2,2,3,4,5,6,6,6,6,10,10,10,10,1,0
,9,3,4,5,6,6,7,7,7,6,7,7,8,7,7,3,2,2,8,8,3,2,10,9,1,2,3,2,3,3,2,1,1
,0,0,1,0,0,9,9,8,7,7,6,7,8,9,4,4,3,2,2,3,3,8,8,9,0,1,2,4,5,6,5,0,1,
2,3,2,3,4,5,4,3,4,5,4,1,1,2,2,6,7,7,8,9,9,9,5,5,4,3,4,4,0,1,1,5,6,5
,6,7,9,9,5,10,1,2,2,3,4,4,4,8,8,9,9,8,10,10,1,1,7,6,5,5,5,5,6,6,6,6
,6,6,5,6,7,8,2,1,0,4,3,8,9,0,9,0,9,8,9,0,1,1,1,0,9,0,9,9,9,8,7,7,6,
6,0,9,0,9,8,7,6,5,4,3,3,4,4,5,6,2,2,1,2,3,3,3,7,8,9,9,8,9,8,8,6,5,4
,3,2,2,2,7,6,7

#时间局部性采取测试数据

```
5,5,5,5,5,3,1,8,5,5,0,8,4,6,5,5,3,6,5,3,5,6,6,10,8,6,3,5,6,5,9,6,6,
9,2,2,6,5,8,3,6,1,1,3,8,9,9,1,3,3,6,8,5,6,2,6,8,3,8,5,5,6,9,1,6,7,2
,3,5,8,5,10,8,7,10,4,1,2,10,5,5,6,3,6,3,1,6,6,5,5,2,0,0,5,7,9,4,8,5
,1,7,9,5,2,8,5,1,3,2,6,1,5,6,2,0,6,2,3,5,8,5,4,2,6,3,4,3,3,10,2,1,1
,5,0,9,8,8,4,9,5,8,9,6,2,5,1,8,6,3,7,1,5,10,2,9,5,1,5,7,0,5,2,9,0,1
0,3,0,0,5,6,4,3,8,8,7,6,0,8,6,2,1,2,6,1,4,0,10,6,5,5,9,8,5,3,10,5,1
,1,9,8,7,6,2,1,5,6,3,5,4,2,6,10,6,5,7,6,7,8,1,0,6,3,8,1,10,7,10,6,9
,3,5,5,6,6,9,6,9,2,5,7,5,5,6,1,3,3,5,8,4,1,6,5,6,0,2,9,10,0,5,1,6,9
,5,1,5,4,1,1,5,2,6,3,0,1,3,6,9,1,5,7,4,5,5,4,7,6,1,6,5,6,1,6,5,5,8,
0,10,9,3,3,5,5,5,10,2,10,9,0,8,3,1,4,5,3,3,8,3,3,0,2,8,4,2,8,10,1,2
,7,1,5,4,9,8,4,7,5,2,0,3,10,10,7,7,6,8,10,7,5,4,10,7,7,0,5,0,1,9,9,
0,1,6,1,4,6,1,6,1,9,7,10,3,1,6,5,5,1,3,2,1,3,6,5,7,4,1,0,6,7,1,7,5,
6,4,4,1,3,10,0,6,1,8,1,9,3,5,10,9,7,2,2,7,2,2,2,10,8,5,6,1,6,1,2,9,
3,10,9,1,1,7,1,0,1,5,2,3,10,0,6,0,8,6,5,3,0,10,7,7,0,2,2,1,3,7,5,9,
10,5,10,3,8,7,7,5,9,3,9,10,2,2,0,10,5,5,7,10,7,6,5,1,6,4,5,8,2,8,5,
4,10,0,0,5,2,10,7,9,1,10,0,9,5,10,0,0,10,7,6,4,1,1,1,1,1,4,8,1,7,0,
9,10,8,2,3,1,1,9,7,6,7,6,1,7,7,0,1,8,0,1,0,4,8,8,7,2,9,5,5,8,1,6,7,
5,6,4,7,9,2,10,10,8,1,6,10,1,7,0,9,9,8,9,10,6,0,6,0,1,2,1,6,6,5,7,5
,3,0,3,8,10,7,0,2,0,5,5,0,5,9,2,2,9,9,10,6,9,6,9,5,3,2,3,5,7,9,5,6,
10,0,7,4,9,10,7,7,7,6,3,10,5,9,8,4,3,3,5,10,0,9,7,8,1,5,4,3,10,0,3,
6,6,3,0,2,9,6,6,9,2,1,1,5,0,1,2,7,6,0,5,3,7,6,0,2,0,8,2,7,3,5,2,6,6
,1,3,7,0,0,6,6,6,2,8,10,1,1,5,1,3,6,5,3,7,0,1,5,4,2,9,4,1,9,0,3,0,4
,8,5,8,6,7,1,2,9,10,1,5,7,5,5,4,2,2,7,7,1,6,7,10,0,8,7,5,0,7,2,7,10
,5,9,5,7,9,0,0,5,8,9,10,9,5,6,2,6,8,10,10,9,9,7,6,2,5,1,1,2,2,0,1,0
,8,0,2,9,3,3,8,9,6,6,8,0,1,3,10,6,5,8,5,1,6,10,3,2,3,6,6,7,6,4,5,0,
2,9,10,4,2,5,6,4,5,4,4,4,9,7,6,0,1,8,7,4,2,6,9,7,2,2,2,9,0,9,4,0,6,
0,0,5,10,9,7,2,2,9,8,2,6,5,6,8,8,3,6,3,1,7,6,3,8,9,10,0,8,7,10,4,8,
10,10,3,4,5,3,7,2,9,0,7,0,1,7,10,9,0,5,0,4,8,10,6,1,6,9,1,0,10,7,1,
6,10,6,10,10,10,1,6,7,5,6,5,4,6,10,10,8,7,6,2,9,8,9,7,3,6,1,4,9,4,3
,6,6,7,5,9,6,5,7,1,4,3,3,3,7,1,4,5,3,10,9,9,7,5,4,5,3,9,3,2,10,1,10
,6,10,5,2,2,7,5,6,10,7,6,6,5,6,7,2,9,10,9,2,1,8,7,1,9,5,10,2,4,0,3,
2,4,0
```

测试结果如下：

时钟位	时间局部性-命中率	空间局部性-命中率
1	30.27%	51.35%
2	31.67%	52.95%
3	30.37%	52.05%

时钟位	时间局部性-命中率	空间局部性-命中率
4	31.67%	53.25%
5	32.17%	51.05%
6	31.27%	50.75%
7	31.37%	51.15%
8	31.47%	51.45%
9	31.47%	50.55%
10	31.47%	50.55%

可以观察到，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。