第十五章

作业

程序 relocation.py 让你看到，在带有基址和边界寄存器的系统中，如何执行地址转换。详情请参阅 README 文件。

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。

种子一：

```
guoruiling@guoruiling-virtual-machine:~/os/hw2$ python3 ./relocation.py -s 1

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

Base-and-Bounds register information:

  Base   : 0x0000363c (decimal 13884)
  Limit  : 290

Virtual Address Trace
  VA  0: 0x0000030e (decimal:  782) --> PA or segmentation violation?
  VA  1: 0x00000105 (decimal:  261) --> PA or segmentation violation?
  VA  2: 0x000001fb (decimal:  507) --> PA or segmentation violation?
  VA  3: 0x000001cc (decimal:  460) --> PA or segmentation violation?
  VA  4: 0x0000029b (decimal:  667) --> PA or segmentation violation?

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

界限为 290。

VA 0: 0x0000030e (decimal: 782) --> 界限外。

VA 1: 0x00000105 (decimal: 261) --> 界限内。0x00003741(decimal:14145)。

VA 2: 0x000001fb (decimal: 507) --> 界限外。

VA 3: 0x000001cc (decimal: 460) --> 界限外。

VA 4: 0x0000029b (decimal: 667) --> 界限外。

检查答案：

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

种子二：

```
guoruiling@guoruiling-virtual-machine:~/os/hw2$ python3 ./relocation.py -s 2

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

Base-and-Bounds register information:

  Base   : 0x00003ca9 (decimal 15529)
  Limit  : 500

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

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

VA 0: 0x00000039 (decimal: 57) --> 界限内。0x00003ce2(decimal:15586)
VA 1: 0x00000056 (decimal: 86) --> 界限内。0x00003cff(decimal:15615)
VA 2: 0x00000357 (decimal: 855) --> 界限外。
VA 3: 0x000002f1 (decimal: 753) --> 界限外。
VA 4: 0x000002ad (decimal: 685) --> 界限外。

检查答案：

```
Virtual Address Trace
  VA  0: 0x00000039 (decimal:   57) --> VALID: 0x00003ce2 (decimal: 15586)
  VA  1: 0x00000056 (decimal:   86) --> VALID: 0x00003cff (decimal: 15615)
  VA  2: 0x00000357 (decimal:  855) --> SEGMENTATION VIOLATION
  VA  3: 0x000002f1 (decimal:  753) --> SEGMENTATION VIOLATION
  VA  4: 0x000002ad (decimal:  685) --> SEGMENTATION VIOLATION
```

种子三：

```
guoruiling@guoruiling-virtual-machine:~/os/hw2$ python3 ./relocation.py -s 3

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

Base-and-Bounds register information:

  Base   : 0x000022d4 (decimal 8916)
  Limit  : 316

Virtual Address Trace
  VA  0: 0x0000017a (decimal:  378) --> PA or segmentation violation?
  VA  1: 0x0000026a (decimal:  618) --> PA or segmentation violation?
  VA  2: 0x00000280 (decimal:  640) --> PA or segmentation violation?
  VA  3: 0x00000043 (decimal:   67) --> PA or segmentation violation?
  VA  4: 0x0000000d (decimal:   13) --> PA or segmentation violation?

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

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

检查答案：

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

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

```
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, 32m)
  -p PSIZE, --physmem=PSIZE physical memory size (e.g., 16, 64k)
  -n NUM, --addresses=NUM # of virtual addresses to generate
  -b BASE, --b=BASE     value of base register
  -l LIMIT, --l=LIMIT   value of limit register
  -c, --compute         compute answers for me
```

-s 1，使用种子 1。-n 10，地址样例个数为 10。-l 100，界限 Limit 为 100。

```
guorulling@guorulling-virtual-machine:~/os/hw2$ python3 ./relocation.py -s 1 -n 10 -l 100

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

Base-and-Bounds register information:

  Base   : 0x00000899 (decimal 2201)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x00000363 (decimal:  867) --> PA or segmentation violation?
  VA  1: 0x0000030e (decimal:  782) --> PA or segmentation violation?
  VA  2: 0x00000105 (decimal:  261) --> PA or segmentation violation?
  VA  3: 0x000001fb (decimal:  507) --> PA or segmentation violation?
  VA  4: 0x000001cc (decimal:  460) --> PA or segmentation violation?
  VA  5: 0x0000029b (decimal:  667) --> PA or segmentation violation?
  VA  6: 0x00000327 (decimal:  807) --> PA or segmentation violation?
  VA  7: 0x00000060 (decimal:   96) --> PA or segmentation violation?
  VA  8: 0x0000001d (decimal:   29) --> PA or segmentation violation?
  VA  9: 0x00000357 (decimal:  855) --> PA or segmentation violation?

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

无需考虑这里给的样例值，界限为 100，最大地址空间为 16k，最大基址为 16k-100=16284.

第十六章

```
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, 32m, 1g)
  -p PSIZE, --physmem=PSIZE
                        physical memory size (e.g., 16, 64k, 32m, 1g)
  -n NUM, --addresses=NUM
                        number of virtual addresses to generate
  -b BASE, --b=BASE     value of base register
  -l LIMIT, --l=LIMIT   value of limit register
  -c, --compute         compute answers for me
```

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
```

地址一：

```
guorulling@guorulling-virtual-machine:~/os/hw2$ python3 ./segmentation.py -a 128 -p 512 -b
0 -l 20 -B 512 -L 20 -s 0
ARG seed 0
ARG address space size 128
ARG phys mem size 512

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 20

  Segment 1 base  (grows negative) : 0x00000200 (decimal 512)
  Segment 1 limit                  : 20

Virtual Address Trace
  VA  0: 0x0000006c (decimal:  108) --> PA or segmentation violation?
  VA  1: 0x00000061 (decimal:   97) --> PA or segmentation violation?
  VA  2: 0x00000035 (decimal:   53) --> PA or segmentation violation?
  VA  3: 0x00000021 (decimal:   33) --> PA or segmentation violation?
  VA  4: 0x00000041 (decimal:   65) --> PA or segmentation violation?

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.
```

地址空间大小 128，物理空间大小 512。seg0 基地址 0，Limit20，seg1 基地址 512，Limit20。随机种子 0。

例如：0x6c 的二进制为 110 1100，首位的 1 指定段为 1，后面的 10 1100 指定段内偏移，由于是反向增长，101100 即 44 减去最大的**段地址 64（按标识位后面的位数最大计算）**，再加物理首地址 0x00000200，即可得到答案。

VA 0: 0x0000006c (decimal: 108) --> seg1:0x000001ec (decimal:492)
VA 1: 0x00000061 (decimal: 97) --> seg1
VA 2: 0x00000035 (decimal: 53) --> seg0
VA 3: 0x00000021 (decimal: 33) --> seg0
VA 4: 0x00000041 (decimal: 65) --> seg1

检查答案：

```
Virtual Address Trace
  VA  0: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492)
  VA  1: 0x00000061 (decimal:   97) --> SEGMENTATION VIOLATION (SEG1)
  VA  2: 0x00000035 (decimal:   53) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x00000021 (decimal:   33) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x00000041 (decimal:   65) --> SEGMENTATION VIOLATION (SEG1)
```

地址二：

```
guoruiling@guoruiling-virtual-machine:~/os/hw2$ 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.
```

地址空间大小 128，物理空间大小 512。seg0 基地址 0，Limit20，seg1 基地址 512，Limit20。随机种子 1。

VA 0: 0x00000011 (decimal: 17) --> seg0:0x00000011 (decimal:17)
VA 1: 0x0000006c (decimal: 108) --> seg1:0x000001ec (decimal:492)
VA 2: 0x00000061 (decimal: 97) --> seg1
VA 3: 0x00000020 (decimal: 32) --> seg0
VA 4: 0x0000003f (decimal: 63) --> seg0

检查答案：

```
Virtual Address Trace
  VA  0: 0x00000011 (decimal:   17) --> VALID in SEG0: 0x00000011 (decimal:   17)
  VA  1: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492)
  VA  2: 0x00000061 (decimal:   97) --> SEGMENTATION VIOLATION (SEG1)
  VA  3: 0x00000020 (decimal:   32) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x0000003f (decimal:   63) --> SEGMENTATION VIOLATION (SEG0)
```

地址三：

```
guorulling@guorulling-virtual-machine:~/os/hw2$ 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.
```

地址空间大小 128，物理空间大小 512。seg0 基地址 0，Limit20，seg1 基地址 512，Limit20。随机种子 2。

VA 0: 0x0000007a (decimal: 122) --> seg1:0x000001fa(decimal:506)
VA 1: 0x00000079 (decimal: 121) --> seg1:0x000001f9(decimal:505)
VA 2: 0x00000007 (decimal: 7) --> seg0:0x00000007 (decimal:7)
VA 3: 0x0000000a (decimal: 10) --> seg0:0x0000000a (decimal:10)
VA 4: 0x0000006a (decimal: 106) --> seg1

检查答案：

```
Virtual Address Trace
  VA  0: 0x0000007a (decimal:  122) --> VALID in SEG1: 0x000001fa (decimal:  506)
  VA  1: 0x00000079 (decimal:  121) --> VALID in SEG1: 0x000001f9 (decimal:  505)
  VA  2: 0x00000007 (decimal:    7) --> VALID in SEG0: 0x00000007 (decimal:    7)
  VA  3: 0x0000000a (decimal:   10) --> VALID in SEG0: 0x0000000a (decimal:   10)
  VA  4: 0x0000006a (decimal:  106) --> SEGMENTATION VIOLATION (SEG1)
```

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？

段 0：0x001 0011。19。段 1：0x110 1100.108。
最低非法地址：0x001 0100.20。最高非法地址：0x110 1011.107。
注意：地址 128 实际是不被用的，栈向上增长，仅做指示，所以 seg0 为 0-19，而 seg1 为 108-127。

```
Virtual Address Trace
  VA  0: 0x00000013 (decimal:   19) --> VALID in SEG0: 0x00000013 (decimal:   19)
  VA  1: 0x00000014 (decimal:   20) --> SEGMENTATION VIOLATION (SEG0)
  VA  2: 0x0000006b (decimal:  107) --> SEGMENTATION VIOLATION (SEG1)
  VA  3: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492)
```

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，违反，有效，有效？假设用以下参数：

```
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 ?
```

只有首尾各两个是有效的，因此界限各为 2，设置为--b0 0 --l0 2 --b1 16 --l1 2 即可。

```
guoruiling@guoruiling-virtual-machine:~/os/hw2$ 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)
```

第十七章

作业

程序 malloc.py 让你探索本章中描述的简单空闲空间分配程序的行为。有关其基本操作的详细信息，请参见 README 文件。

```
-h, --help            show this help message and exit
-s SEED, --seed=SEED  the random seed
-S HEAPSIZE, --size=HEAPSIZE
                      size of the heap
-b BASEADDR, --baseAddr=BASEADDR
                      base address of heap
-H HEADERSIZE, --headerSize=HEADERSIZE
                      size of the header
-a ALIGNMENT, --alignment=ALIGNMENT
                      align allocated units to size; -1->no align
-p POLICY, --policy=POLICY
                      list search (BEST, WORST, FIRST)
-l ORDER, --listOrder=ORDER
                      list order (ADDRSORT, SIZESORT+, SIZESORT-, INSERT-FRONT, INSERT-BACK)
-C, --coalesce        coalesce the free list?
-n OPSNUM, --numOps=OPSNUM
                      number of random ops to generate
-r OPSRANGE, --range=OPSRANGE
                      max alloc size
-P OPSPALLOC, --percentAlloc=OPSPALLOC
                      percent of ops that are allocs
-A OPSLIST, --allocList=OPSLIST
                      instead of random, list of ops (+10,-0,etc)
-c, --compute         compute answers for me
```

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？

生成随机操作数目 10，头的大小为 0，使用 best 匹配，随机种子-s 0. 基址为 1000.

```
guoruiling@guoruiling-virtual-machine:~/os/hw2$ python3 malloc.py -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?
```

开辟一个空间为 3 的块，返回指向的指针为 1000，空闲列表：指针 1003，大小 97。释放该块。

开辟一个空间为 5 的块，返回指向的指针为 1003，空闲列表：指针 1003，大小 3，指针 1008，大小 92。释放该块。

开辟一个空间为 8 的块，返回指向的指针为 1008，空闲列表：指针 1003，大小 3，指针 1003，大小 5，指针 1016，大小 84。释放该块。

开辟一个空间为 2 的块，返回指向的指针为 1000，空闲列表：指针 1002，大小 1，指针 1003，大小 5，指针 1008，大小 8，指针 1016，大小 84。不释放该块。

开辟一个空间为 7 的块，返回指向的指针为 1008，空闲列表：指针 1002，大小 1，指针 1003，大小 5，指针 10015，大小 1，指针 1016，大小 84。不释放该块。

6 次分配空间查找空闲块的次数分别为：1,2,3,4,4,4。

由于我们没有合并，空闲列表变得越来越破碎，每一个块都很小。

```
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 ]
```

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？

```
guoruiling@guoruiling-virtual-machine:~/os/hw2$ python3 malloc.py -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?
```

空间分配即空闲列表与最佳匹配时相同，但是分配空间查找空闲块的次数分别为：1,2,3,3,1,3。

搜索空闲列表变快了。

```
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 ]
```

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。

最优匹配和最差匹配都需要遍历整个列表，不同的排序策略没有影响，只需要查看对于首次匹配的影响。

开辟：3,5,8,8,2,7。

-l ADDRSORT：

根据地址排序。搜索次数 1,2,3,3,1,3.

```
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 ]
```

-l SIZESORT +：

根据大小升序排序。搜索次数：1,2,3,3,1,3

```
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 ]
```

-l SIZESORT -：
根据大小降序排序。搜索次数：1,1,1,1,1,1
搜索次数显著减少，但是空间变得更加零碎，后续如果使用大块时可能会无法分配，可维护性降低。

```
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 ]
```

第十八章

作业

在这个作业中，你将使用一个简单的程序（名为 paging-linear-translate.py），来看看你是否理解了简单的虚拟—物理地址转换如何与线性页表一起工作。详情请参阅 README 文件。

```
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
```

1. 在做地址转换之前，让我们用模拟器来研究线性页表在给定不同参数的情况下如何改变大小。在不同参数变化时，计算线性页表的大小。一些建议输入如下，通过使用-v 标志，你可以看到填充了多少个页表项。

首先，要理解线性页表大小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
```

在运行这些命令之前，请试着想想预期的趋势。页表大小如何随地址空间的增长而改变？随着页大小的增长呢？为什么一般来说，我们不应该使用很大的页呢？

运行结果过长，此处不再截图展示。

页大小为 1kb，地址空间大小分别为 1mb，2mb，4mb，页表项应分别有 1mb/1kb=1024，2mb/1kb=2048，4mb/1kb=4096。随-a，即虚拟地址空间增大，页表的大小也增大，呈线性关系。

页大小分别为 1kb，2kb，4kb，地址空间大小为 1mb，页表项分别有 1mb/1kb=1024，1mb/2kb=512，1mb/4kb=256。随-P，即页的大小增大，页表的大小减小，呈线性关系。

上述结果显而易见。物理空间为 512k，且物理空间大小不影响页表大小。

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？

页大小为 1k，虚拟空间大小为 16K，物理空间大小为 32K。

被使用百分比为 0：

```
guoruiling@guoruiling-virtual-machine:~/os/hw2$ python3 paging-linear-translate.py -P 1k -a 16k
-p 32k -v -u 0
ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 1k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]  0x00000000
  [       1]  0x00000000
  [       2]  0x00000000
  [       3]  0x00000000
  [       4]  0x00000000
  [       5]  0x00000000
  [       6]  0x00000000
  [       7]  0x00000000
  [       8]  0x00000000
  [       9]  0x00000000
  [      10]  0x00000000
  [      11]  0x00000000
  [      12]  0x00000000
  [      13]  0x00000000
  [      14]  0x00000000
  [      15]  0x00000000

Virtual Address Trace
  VA 0x00003a39 (decimal:    14905) --> PA or invalid address?
  VA 0x00003ee5 (decimal:    16101) --> PA or invalid address?
  VA 0x000033da (decimal:    13274) --> PA or invalid address?
  VA 0x000039bd (decimal:    14781) --> PA or invalid address?
  VA 0x000013d9 (decimal:     5081) --> 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).
```

```
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)
```

被使用百分比为 25：

```
guoruiling@guoruiling-virtual-machine:~/os/hw2$ 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).
```

```
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)
```

第二个地址，在虚拟页 10，偏移量为 966。映射到物理地址为 0x8000 0013，即物理页 19,19*1024+966=20422。（因为有 0，所以计算时第 19 页乘 19 而不是 18）

被使用百分比为 50：

```
guorulling@guorulling-virtual-machine:~/os/hw2$ 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).
```

```
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]
```

被使用百分比为 75：

```
guorulling@guorulling-virtual-machine:~/os/hw2$ 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).
```

```
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]
```

被使用百分比为 100：

```
guoruiling@guoruiling-virtual-machine:~/os/hw2$ 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).
```

```
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]
```

增加每个地址空间中的页的百分比，有效地址将更多。

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？

第一组：

页大小为 8，虚拟地址空间为 32，页数为 4（页表项为 4），物理地址空间为 1024。

虚拟地址一共 5 位，前两位用于表示页号，后三位偏移量。

组合不合理，空间太小，存不下东西。

```
guoruiling@guoruiling-virtual-machine:~/os/hw2$ python3 paging-linear-translate.py -P 8 -a 32 -p
 1024 -v -s 1
ARG seed 1
ARG address space size 32
ARG phys mem size 1024
ARG page size 8
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]  0x00000000
  [       1]  0x80000061
  [       2]  0x00000000
  [       3]  0x00000000

Virtual Address Trace
  VA 0x0000000e (decimal:       14) --> PA or invalid address?
  VA 0x00000014 (decimal:       20) --> PA or invalid address?
  VA 0x00000019 (decimal:       25) --> PA or invalid address?
  VA 0x00000003 (decimal:        3) --> PA or invalid address?
  VA 0x00000000 (decimal:        0) --> PA or invalid address?

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).
```

```
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)
```

第二组：

页大小为 8k，虚拟地址空间为 32k，页数为 4（页表项为 4），物理地址空间为 1m。

虚拟地址一共 15 位，前 2 位用于表示页号，后 13 位偏移量。

组合不合理，页空间太大而分页太少，会导致大量空间浪费。

```
guoruiling@guoruiling-virtual-machine:~/os/hw2$ python3 paging-linear-translate.py -P 8k -a 32k
-p 1m -v -s 2
ARG seed 2
ARG address space size 32k
ARG phys mem size 1m
ARG page size 8k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]  0x80000079
  [       1]  0x00000000
  [       2]  0x00000000
  [       3]  0x8000005e

Virtual Address Trace
  VA 0x000055b9 (decimal:    21945) --> PA or invalid address?
  VA 0x00002771 (decimal:    10097) --> PA or invalid address?
  VA 0x00004d8f (decimal:    19855) --> PA or invalid address?
  VA 0x00004dab (decimal:    19883) --> PA or invalid address?
  VA 0x00004a64 (decimal:    19044) --> 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).
```

```
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)
```

第三组：
页大小为 1m，虚拟地址空间为 256m，页数为 256（页表项为 256），物理地址空间为 512m。
虚拟地址一共 28 位，前 8 位用于表示页号，后 20 位偏移量。
组合相对较合理。

```
  [     222]  0x80000183
  [     223]  0x80000108
  [     224]  0x00000000
  [     225]  0x00000000
  [     226]  0x00000000
  [     227]  0x800001d5
  [     228]  0x800000e2
  [     229]  0x8000005f
  [     230]  0x00000000
  [     231]  0x00000000
  [     232]  0x00000000
  [     233]  0x800001e8
  [     234]  0x00000000
  [     235]  0x800000d3
  [     236]  0x00000000
  [     237]  0x00000000
  [     238]  0x00000000
  [     239]  0x00000000
  [     240]  0x00000000
  [     241]  0x00000000
  [     242]  0x00000000
  [     243]  0x00000000
  [     244]  0x80000049
  [     245]  0x800000f5
  [     246]  0x800000ef
  [     247]  0x800001a4
  [     248]  0x800000f6
  [     249]  0x00000000
  [     250]  0x800001eb
  [     251]  0x00000000
  [     252]  0x00000000
  [     253]  0x00000000
  [     254]  0x80000159
  [     255]  0x00000000

Virtual Address Trace
  VA 0x0308b24d (decimal: 50901581) --> PA or invalid address?
  VA 0x042351e6 (decimal: 69423590) --> PA or invalid address?
  VA 0x02feb67b (decimal: 50247291) --> PA or invalid address?
  VA 0x0b46977d (decimal: 189175677) --> PA or invalid address?
  VA 0x0dbcceb4 (decimal: 230477492) --> PA or invalid address?

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

Help

```
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]
```