Assignment 6

Operating Systems Lab

Mohammad Sameer | 190010024 | 190010024@iitdh.ac.in

All over the document 1 byte = 8 bits. $1KB = 1024$ bytes. $1MB = 1024 \times 1024$ bytes. $1GB = 1024 \times 1024 \times 1024$ bytes.

Part 1

The program `relocation.py` allows you to see how address translations are performed in a system with base and bounds registers.

1.1

Run with seeds 1, 2, and 3, and compute whether each virtual address generated by the process is in or out of bounds. If in bounds, compute the translation.

Answer:

1. Command used:

   ```
   python2 relocation.py -s 1
   ```

   Output and calculated translations:

   ```
   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) --> INVALID
     VA  1: 0x00000105 (decimal:  261) --> VALID: 0x00003741 (decimal: 14145)
     VA  2: 0x000001fb (decimal:  507) --> INVALID
     VA  3: 0x000001cc (decimal:  460) --> INVALID
     VA  4: 0x0000029b (decimal:  667) --> INVALID
   ```

2. Command used:

   ```
   python2 relocation.py -s 2
   ```

   Output and calculated translations:

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

   Base-and-Bounds register information:
   ```

```
  Base   : 0x00003ca9 (decimal 15529)
  Limit  : 500

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

3. Command used:

   ```
   python2 relocation.py -s 3
   ```

   Output and calculated translations:

   ```
   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) --> INVALID
     VA  1: 0x0000026a (decimal:  618) --> INVALID
     VA  2: 0x00000280 (decimal:  640) --> INVALID
     VA  3: 0x00000043 (decimal:   67) --> VALID: 0x00002317 (decimal: 8983)
     VA  4: 0x0000000d (decimal:   13) --> VALID: 0x000022e1 (decimal: 8929)
   ```

1.2

Run with these flags: -s 0 -n 10. What value do you have set -l (the bounds register) to in order to ensure that all the generated virtual addresses are within bounds?

Answer: The command used is:

```
python2 relocation.py -s 0 -n 10 -c
```

The output is:

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

Base-and-Bounds register information:

  Base   : 0x00003082 (decimal 12418)
  Limit  : 472

Virtual Address Trace
  VA  0: 0x000001ae (decimal:  430) --> VALID: 0x00003230 (decimal: 12848)
  VA  1: 0x00000109 (decimal:  265) --> VALID: 0x0000318b (decimal: 12683)
  VA  2: 0x0000020b (decimal:  523) --> SEGMENTATION VIOLATION
  VA  3: 0x0000019e (decimal:  414) --> VALID: 0x00003220 (decimal: 12832)
  VA  4: 0x00000322 (decimal:  802) --> SEGMENTATION VIOLATION
  VA  5: 0x00000136 (decimal:  310) --> VALID: 0x000031b8 (decimal: 12728)
  VA  6: 0x000001e8 (decimal:  488) --> SEGMENTATION VIOLATION
  VA  7: 0x00000255 (decimal:  597) --> SEGMENTATION VIOLATION
  VA  8: 0x000003a1 (decimal:  929) --> SEGMENTATION VIOLATION
  VA  9: 0x00000204 (decimal:  516) --> SEGMENTATION VIOLATION
```

By observation, the largest virtual address accessed is 929, so setting flag `-l 930` would ensure that all the generated virtual addresses are within bound.

The command after setting the needed flag:

```
python2 relocation.py -s 0 -n 10 -l 930 -c
```

The output is:

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

Base-and-Bounds register information:

  Base   : 0x0000360b (decimal 13835)
  Limit  : 930

Virtual Address Trace
  VA  0: 0x00000308 (decimal:  776) --> VALID: 0x00003913 (decimal: 14611)
  VA  1: 0x000001ae (decimal:  430) --> VALID: 0x000037b9 (decimal: 14265)
  VA  2: 0x00000109 (decimal:  265) --> VALID: 0x00003714 (decimal: 14100)
  VA  3: 0x0000020b (decimal:  523) --> VALID: 0x00003816 (decimal: 14358)
  VA  4: 0x0000019e (decimal:  414) --> VALID: 0x000037a9 (decimal: 14249)
  VA  5: 0x00000322 (decimal:  802) --> VALID: 0x0000392d (decimal: 14637)
  VA  6: 0x00000136 (decimal:  310) --> VALID: 0x00003741 (decimal: 14145)
  VA  7: 0x000001e8 (decimal:  488) --> VALID: 0x000037f3 (decimal: 14323)
  VA  8: 0x00000255 (decimal:  597) --> VALID: 0x00003860 (decimal: 14432)
  VA  9: 0x000003a1 (decimal:  929) --> VALID: 0x000039ac (decimal: 14764)
```

1.3

Run with these flags: -s 1 -n 10 -l 100. What is the maximum value that base can be set to, such that the address space still fits into physical memory in its entirety?

Answer: The command used is:

```
python2 relocation.py -s 1 -n 10 -l 100
```

The output is:

```
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.
```

From the above, we can see that the physical memory size is $16KB$ and limit / bound is 100. The maximum value that base can be set to is $(16 * 1024) - 100 = 16284$.

The command to set it is:

```
python2 relocation.py -s 1 -n 10 -l 100 -b 16284
```

1.4

Run some of the same problems above, but with larger address spaces (-a) and physical memories (-p).

Answer: Case 1: Size of address space (-a) is 4 KB, size of physical memory (-p) is 64 KB.

a. Command Used:

```
python2 relocation.py -s 2 -a 4k -p 64k
```

Output and calculated translations:

```
ARG seed 2
ARG address space size 4k
ARG phys mem size 64k

Base-and-Bounds register information:

  Base   : 0x0000f2a4 (decimal 62116)
  Limit  : 2002

Virtual Address Trace
  VA  0: 0x000000e7 (decimal:  231) --> VALID: 0x0000f38b (decimal: 62347)
  VA  1: 0x0000015b (decimal:  347) --> VALID: 0x0000f3ff (decimal: 62463)
  VA  2: 0x00000d5e (decimal: 3422) --> INVALID
  VA  3: 0x00000bc6 (decimal: 3014) --> INVALID
  VA  4: 0x00000ab7 (decimal: 2743) --> INVALID
```

b. Command Used:

```
python2 relocation.py -s 2 -n 10 -a 4k -p 64k
```

By observation, the largest virtual address accessed is 3422, so setting flag `-l 3423` would ensure that all the generated virtual addresses are within bound. The command after setting the needed flag:

```
python2 relocation.py -s 2 -n 10 -l 3423 -a 4k -p 64k -c
```

The output is:

```
ARG seed 2
ARG address space size 4k
ARG phys mem size 64k

Base-and-Bounds register information:

  Base   : 0x00000e7a (decimal 3706)
  Limit  : 3423

Virtual Address Trace
  VA  0: 0x0000015b (decimal:  347) --> VALID: 0x00000fd5 (decimal: 4053)
  VA  1: 0x00000d5e (decimal: 3422) --> VALID: 0x00001bd8 (decimal: 7128)
  VA  2: 0x00000bc6 (decimal: 3014) --> VALID: 0x00001a40 (decimal: 6720)
  VA  3: 0x00000ab7 (decimal: 2743) --> VALID: 0x00001931 (decimal: 6449)
  VA  4: 0x000004ee (decimal: 1262) --> VALID: 0x00001368 (decimal: 4968)
  VA  5: 0x000009b1 (decimal: 2481) --> VALID: 0x0000182b (decimal: 6187)
  VA  6: 0x000009b5 (decimal: 2485) --> VALID: 0x0000182f (decimal: 6191)
  VA  7: 0x0000094c (decimal: 2380) --> VALID: 0x000017c6 (decimal: 6086)
  VA  8: 0x00000288 (decimal:  648) --> VALID: 0x00001102 (decimal: 4354)
  VA  9: 0x000006e4 (decimal: 1764) --> VALID: 0x0000155e (decimal: 5470)
```

c. The command used is:

```
python2 relocation.py -s 2 -n 10 -l 100 -a 4k -p 64k
```

From the above, we can see that the physical memory size is $64KB$ and limit / bound is 100. The maximum value that base can be set to is $(64 * 1024) - 100 = 65436$.

The command to set it is:

```
python2 relocation.py -s 2 -n 10 -l 100 -b 65436 -a 4k -p 64k
```

Case 2: Size of address space (-a) is 64 MB, size of physical memory (-p) is 2 GB.

a. Command Used:

```
python2 relocation.py -s 1 -a 64m -p 2g
```

Output and calculated translations:

```
ARG seed 1
ARG address space size 64m
ARG phys mem size 2g

Base-and-Bounds register information:
  Base   : 0x6c78b56c (decimal 1819850092)
  Limit  : 19031473

Virtual Address Trace
  VA  0: 0x030e1aef (decimal: 51256047) --> INVALID
  VA  1: 0x010530d0 (decimal: 17117392) --> VALID: 0x6d7de63c (decimal: 1836967484)
  VA  2: 0x01fb5355 (decimal: 33248085) --> INVALID
  VA  3: 0x01cc4762 (decimal: 30164834) --> INVALID
  VA  4: 0x029b3b30 (decimal: 43727664) --> INVALID
```

b. Command Used:

```
python2 relocation.py -s 1 -n 10 -a 64m -p 2g
```

By observation, the largest virtual address accessed is 56870315, so setting flag `-l 56870316` would ensure that all the generated virtual addresses are within bound. The command after setting the needed flag:

```
python2 relocation.py -s 1 -n 10 -l 56870316 -a 64m -p 2g -c
```

The output is:

```
ARG seed 1
ARG address space size 64m
ARG phys mem size 2g

Base-and-Bounds register information:

  Base   : 0x1132d8f9 (decimal 288545017)
  Limit  : 56870316

Virtual Address Trace
  VA  0: 0x0363c5ab (decimal: 56870315) --> VALID: 0x14969ea4 (decimal: 345415332)
  VA  1: 0x030e1aef (decimal: 51256047) --> VALID: 0x1440f3e8 (decimal: 339801064)
  VA  2: 0x010530d0 (decimal: 17117392) --> VALID: 0x123809c9 (decimal: 305662409)
  VA  3: 0x01fb5355 (decimal: 33248085) --> VALID: 0x132e2c4e (decimal: 321793102)
  VA  4: 0x01cc4762 (decimal: 30164834) --> VALID: 0x12ff205b (decimal: 318709851)
  VA  5: 0x029b3b30 (decimal: 43727664) --> VALID: 0x13ce1429 (decimal: 332272681)
  VA  6: 0x0327a718 (decimal: 52930328) --> VALID: 0x145a8011 (decimal: 341475345)
  VA  7: 0x00601cba (decimal: 6298810) --> VALID: 0x1192f5b3 (decimal: 294843827)
  VA  8: 0x001d071e (decimal: 1902366) --> VALID: 0x114fe017 (decimal: 290447383)
  VA  9: 0x0357d2ce (decimal: 56087246) --> VALID: 0x148aabc7 (decimal: 344632263)
```

c. The command used is:

```
python2 relocation.py -s 1 -n 10 -l 100 -a 64m -p 2g
```

From the above, we can see that the physical memory size is $2GB$ and limit / bound is 100. The maximum value that base can be set to is $(2 * 1024 * 1024 * 1024) - 100 = 2147483548$.

The command to set it is:

```
python2 relocation.py -s 2 -n 10 -l 100 -b 2147483548 -a 64m -p 2g
```

1.5

What fraction of randomly-generated virtual addresses are valid, as a function of the value of the bounds register? Make a graph from running with different random seeds, with limit values ranging from 0 up to the maximum size of the address space.

Answer: The plot given below shows the fraction of randomly-generated virtual addresses are valid, as a function of the value of the bounds register where address space size is $1KB$ and physical memory size if $16KB$.

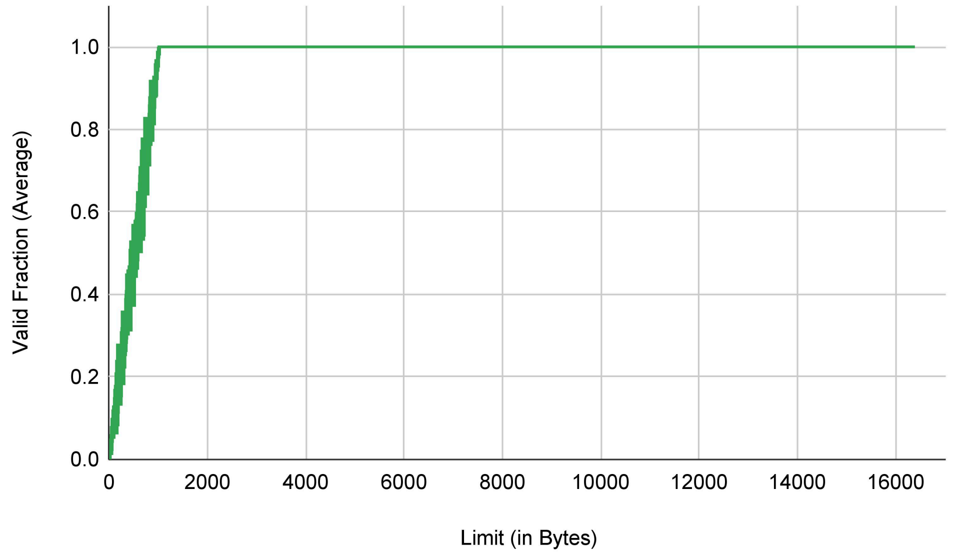

Figure 1: Plot of valid address as a function of limit

From this above graph we can see that as limit increases (until 1000 bytes), more and more virtual address are valid. But after the limit crosses the 1000 bytes value, which is equal to size of address space, all the virtual addresses are valid as expected.

Part 2

The program `segmentation.py` allows you to see how address translations are performed in a system with segmentation.

2.1

First let's use a tiny address space to translate some addresses. Here's a simple set of parameters with a few different random seeds; can you translate the addresses?

```
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
```

Answer: a. Command Used:

```
python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0
```

Output and calculated translations:

```
ARG seed 0
ARG address space size 128
ARG phys mem size 512

Segment register information:

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

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

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

b. Command Used:

```
python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1
```

Output and calculated translations:

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

c. Command Used:

```
python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2
```

Output and calculated translations:

```
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) --> 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) --> INVALID (SEG1)
```

2.2

Now, let's see if we understand this tiny address space we've constructed (using the parameters from the question above). What is the highest legal virtual address in segment 0? What about the lowest legal virtual address in segment 1? What are the lowest and highest illegal addresses in this entire address space? Finally, how would you run segmentation.py with the -A flag to test if you are right?

Answer:

- The highest legal virtual address in segment 0 is 19.
- The lowest legal virtual address in segment 1 is $128 - 20 = 108$.
- The lowest illegal addresses in this entire address space is 20 which is same as physical address 20.
- The highest illegal virtual addresses in this entire address space is 107 which is same as physical address $512 - 21 = 491$.
- To check the above the answer we use flag -A $19, 108, 20, 107$.

The command used is:

```
python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0 -A 19,108,20,107 -c
```

The output is:

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

2.3

Let's say we have a tiny 16-byte address space in a 128-byte physical memory. What base and bounds would you set up so as to get the simulator to generate the following translation results for the specified address stream: valid, valid, violation, ..., violation, valid, valid? Assume the following parameters:

```
    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 ?
```

Answer:

- For segment 0, base (b0) is 0 and bound (l0) is 2.
- For segment 1, base (b1) is 16 and bound (l1) is 2.

The command would become:

```
python2 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
```

The output is:

```
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)
```

2.4

Assume we want to generate a problem where roughly 90% of the randomly-generated virtual addresses are valid (not segmentation violations). How should you configure the simulator to do so? Which parameters are important to getting this outcome?

Answer: To achieve this, the valid address space should cover roughly 90% of the whole virtual address space size. Depending on the various sizes of virtual address space and physical memory size, we can modify bounds (l0 and l1) to achieve this. An example of which is given below:

```
python2 segmentation.py -a 16 -p 128 --b0 0 --l0 7 --b1 15 --l1 7 -c
```

Here, the virtual address space size is 16 bytes and physical memory size is 128 bytes. The bound are set in such a way that roughly 90% of the virtual address space size is covered.

2.5

Can you run the simulator such that no virtual addresses are valid? How?

Answer: Settings the both the segment 0 and segment 1 bounds to 0 would make no virtual addresses are valid. An example command would be:

```
python2 segmentation.py -a 16 -p 128 --b0 0 --l0 0 --b1 15 --l1 0 -c
```

Part 3

The program `paging-linear-size.py` lets you figure out the size of a linear page table given a variety of input parameters.

Compute how big a linear page table is with the characteristics such as different number of bits in the address space, different page size, different page table entry size. Explain your answers for various cases.

Answer: Case 1: Command Used:

```
python2 paging-linear-size.py
```

- This is the default case, here virtual address has 32 bits. Each page size is $4KB$ and page table entry (pte) size is 4 bytes.
- The virtual address has two parts, first V bits represent Virtual Page Number (VPN) and remaining O bits represent the offset.
- In this case, since page size is $4KB$ or 4096 bytes, we need $log_2(4096) = 12$ bits to represent the offset. So, $O = 12$.
- This leaves $32 - 12 = 20$ bits for VPN. So, $V = 20$.
- To get total size of linear page table we need to multiply size of each page table entry with total number of table entries.
  - The size of each page table entry is 4 bytes.
  - The total number of table entries is $2^V = 2^{20} = 1048576$.
- So, total size of linear page table is $4 \times 1048576 = 4194304$ bytes or $4096KB$ or $4MB$.

Case 2: Command Used:

```
python2 paging-linear-size.py -v 20 -e 2 -p 2k
```

- In this case the virtual address has 20 bits. Each page size is $2KB$ and page table entry (pte) size is 2 bytes.
- The virtual address has two parts, first V bits represent Virtual Page Number (VPN) and remaining O bits represent the offset.
- In this case, since page size is $2KB$ or 20486 bytes, we need $log_2(2046) = 11$ bits to represent the offset. So, $O = 11$.
- This leaves $20 - 11 = 9$ bits for VPN. So, $V = 9$.

- To get total size of linear page table we need to multiply size of each page table entry with total number of table entries.
  - The size of each page table entry is 2 bytes.
  - The total number of table entries is $2^V = 2^9 = 512$.
- So, total size of linear page table is $2 \times 512 = 1024$ bytes or $1KB$.

Case 3: Command Used:

```
python2 paging-linear-size.py -v 40 -e 16 -p 16k
```

- In this case the virtual address has 40 bits. Each page size is $16KB$ and page table entry (pte) size is 16 bytes.
- The virtual address has two parts, first V bits represent Virtual Page Number (VPN) and remaining O bits represent the offset.
- In this case, since page size is $16KB$ or 16384 bytes, we need $log_2(16384) = 14$ bits to represent the offset. So, $O = 14$.
- This leaves $40 - 14 = 26$ bits for VPN. So, $V = 26$.
- To get total size of linear page table we need to multiply size of each page table entry with total number of table entries.
  - The size of each page table entry is 16 bytes.
  - The total number of table entries is $2^V = 2^{26} = 67108864$.
- So, total size of linear page table is $16 \times 67108864 = 1073741824$ bytes or $1073741824KB$ or $1024MB$ or $1GB$.

Part 4

4.1

Before doing any translations, let's use the simulator to study how linear page tables change size given different parameters. Compute the size of linear page tables as different parameters change. Some suggested inputs are below; by using the -v flag,you can see how many page-table entries are filled. First, to understand how linear page table size changes as the address space grows:

```
-P 1k -a 1m -p 512m -v -n 0
-P 1k -a 2m -p 512m -v -n 0
-P 1k -a 4m -p 512m -v -n 0
```

Then, to understand how linear page table size changes as page size grows:

```
-P 1k -a 1m -p 512m -v -n 0
-P 2k -a 1m -p 512m -v -n 0
-P 4k -a 1m -p 512m -v -n 0
```

Before running any of these, try to think about the expected trends. How should page-table size change as the address space grows? As the page size grows? Why not use big pages in general?

Answer: In the first set of flags, we can see that (virtual) address space size increases while other parameters remain constant. This means more pages are needed to larger address space, therefore the number of page table entries increases, making page table size increases as well. For each of the flags below there linear page table size is as follows:

```
-P 1k -a 1m -p 512m -v -n 0  -->  1024 × (size of pte)
-P 1k -a 2m -p 512m -v -n 0  -->  2048 × (size of pte)
-P 1k -a 4m -p 512m -v -n 0  -->  4096 × (size of pte)
```

In the next set of flags, we can see that page size increases while other parameters remain constant. This means, for a constant size address space, we need less number of pages to represent it with a larger page size. Therefore, the number of page table entries decreases, making page table size decrease as well. For each of the flags below there linear page table size is as follows:

```
-P 1k -a 1m -p 512m -v -n 0  -->  1024 × (size of pte)
-P 2k -a 1m -p 512m -v -n 0  -->   512 × (size of pte)
-P 4k -a 1m -p 512m -v -n 0  -->   256 × (size of pte)
```

The above two reasoning can be summed up in the following equation:

{Number of page table entries} = {address space size} / {page size}

If we use really big pages in general, then it would lead to problem of internal fragmentation. Meaning, not all the portion of the big page would be used, leading to wastage of memory inside an allocated page.

4.2

Now let's do some translations. Start with some small examples, and change the number of pages that are allocated to the address space with the -u flag. For example:

```
-P 1k -a 16k -p 32k -v -u 0
-P 1k -a 16k -p 32k -v -u 25
-P 1k -a 16k -p 32k -v -u 50
-P 1k -a 16k -p 32k -v -u 75
-P 1k -a 16k -p 32k -v -u 100
```

What happens as you increase the percentage of pages that are allocated in each address space?

Answer: Command:

```
python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 0
```

Output and calculated translations:

```
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) -->  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)
```

Command:

```
python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 25
```

Output and calculated translations:

```
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) -->  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)
```

Command:

```
python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 50
```

Output and calculated translations:

```
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) --> 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]
```

Command:

```
python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 75
```

Output and calculated translations:

```
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) --> 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]
```

Command:

```
python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 100
```

Output and calculated translations:

```
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) --> 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]
```

As the percentage of pages that are allocated in each address space increases, the number of valid pages increases.

4.3

Now let's try some different random seeds, and some different (and sometimes quite crazy) address-space parameters, for variety:

```
-P 8  -a 32   -p 1024 -v -s 1
-P 8k -a 32k  -p 1m   -v -s 2
-P 1m -a 256m -p 512m -v -s 3
```

Which of these parameter combinations are unrealistic? Why?

Answer: Command:

```
python2 paging-linear-translate.py -P 8 -a 32 -p 1024 -v -s 1
```

Output and calculated translations:

```
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) --> 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)
```

Command:

```
python2 paging-linear-translate.py -P 8k -a 32k -p 1m -v -s 2
```

Output and calculated translations:

```
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) -->  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)
```

Command:

```
python2 paging-linear-translate.py -P 1m -a 256m -p 512m -v -s 3
```

Output and calculated translations:

```
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]
```

The first command parameters are too small, they are in bytes. The third command parameters are too large, the page size is too large and address space is half of that of physical address memory.

4.4

Use the program to try out some other problems. Can you find the limits of where the program doesn't work anymore? For example, what happens if the address-space size is bigger than physical memory?

Answer: • The address space size cannot be larger than physical memory, as we cannot fit complete address space in physical memory. Command and output:

```
$ python2 paging-linear-translate.py -p 64k -a 65k -v -c
ARG seed 0
ARG address space size 65k
ARG phys mem size 64k
ARG page size 4k
ARG verbose True
ARG addresses -1

Error: physical memory size must be GREATER than address space size
(for this simulation)
```

• The page size cannot be zero (0), because it represents no part of memory. Command and output:

```
$ python2 paging-linear-translate.py -P 0 -v -c
ARG seed 0
ARG address space size 16k
ARG phys mem size 64k
ARG page size 0
ARG verbose True
ARG addresses -1

Traceback (most recent call last):
  File "paging-linear-translate.py", line 85, in <module>
    mustbemultipleof(asize, pagesize, 'address space must be a multiple
    of the pagesize')
  File "paging-linear-translate.py", line 14, in mustbemultipleof
    if (int(float(bignum)/float(num)) != (int(bignum) / int(num))):
ZeroDivisionError: float division by zero
```

• The address space size cannot be zero (0), because it represents no part of memory. Command and output:

```
$ python2 paging-linear-translate.py -a 0 -v -c
ARG seed 0
ARG address space size 0
ARG phys mem size 64k
ARG page size 4k
ARG verbose True
ARG addresses -1

Error: must specify a non-zero address-space size.
```

• The physical memory size cannot be zero (0), because it represents no part of memory. Command and output:

```
$ python2 paging-linear-translate.py -p 0 -v -c
ARG seed 0
ARG address space size 16k
ARG phys mem size 0
ARG page size 4k
ARG verbose True
ARG addresses -1

Error: must specify a non-zero physical memory size.
```

• The page size cannot be larger than size of address space, as it would lead to invalid memory access. Command and part of output:

```
$ python2 paging-linear-translate.py -P 32k -a 16k -v -c
ARG seed 0
ARG address space size 16k
ARG phys mem size 64k
ARG page size 32k
ARG verbose True
ARG addresses -1

Page Table (from entry 0 down to the max size)

Virtual Address Trace
Traceback (most recent call last):
  File "paging-linear-translate.py", line 174, in <module>
    if pt[vpn] < 0:
IndexError: array index out of range
```