CS314 : Operating Systems Lab

Lab 7

Sourabh Bhosale
200010004

March 5, 2023

1 Part 1

The program `relocation.py` allows you to see how address translations are performed in a system with base and bounds registers. See the README_base_bound for more details.

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.

```
CS314 OS lab/Submissions/Lab-7 $ python2 relocation.py -s 1 -c

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) --> 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
```

```
CS314 OS lab/Submissions/Lab-7 $ python2 relocation.py -s 2 -c

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

```
CS314 OS lab/Submissions/Lab-7 $ python2 relocation.py -s 3 -c

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) --> 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)
```

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?

As we can see from the figure below that for seed value 0 and 10 virtual addresses, we will need to set the value of `-l = 930` . We can infer this from the fact that the largest address of VA alloted here is 929, so we will need atleast one more than that to get the valid physical address mapped to it (generated virtual addresses are within bounds).

```
● CS314 OS lab/Submissions/Lab-7 $ python2 relocation.py -s 0 -n 10 -c -l 929

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

  Base-and-Bounds register information:

    Base   : 0x0000360b (decimal 13835)
    Limit  : 929

  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) --> SEGMENTATION VIOLATION

● CS314 OS lab/Submissions/Lab-7 $ python2 relocation.py -s 0 -n 10 -c -l 930

  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)
```

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?

We know that $if(base + limit) > psize$ then address space does not fit into physical memory with those base/bounds values. We have the given psize as 16k(16384) and limit as 100, so maximum value of the base can be 16284.

README-base_bound relocation.py 1 ×

relocation.py > ...

```
85  if base + limit > psize:
86      print 'Error: address space does not fit into physical memory with those base/bounds values.'
87      print 'Base + Limit:', base + limit, '  Psize:', psize
88      exit(1)
89
```

```
CS314 OS lab/Submissions/Lab-7 $ python2 relocation.py -s 1 -n 10 -l 100 -c -b 16284

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

Base-and-Bounds register information:

  Base   : 0x00003f9c (decimal 16284)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x00000089 (decimal:  137) --> SEGMENTATION VIOLATION
  VA  1: 0x00000363 (decimal:  867) --> SEGMENTATION VIOLATION
  VA  2: 0x0000030e (decimal:  782) --> SEGMENTATION VIOLATION
  VA  3: 0x00000105 (decimal:  261) --> SEGMENTATION VIOLATION
  VA  4: 0x000001fb (decimal:  507) --> SEGMENTATION VIOLATION
  VA  5: 0x000001cc (decimal:  460) --> SEGMENTATION VIOLATION
  VA  6: 0x0000029b (decimal:  667) --> SEGMENTATION VIOLATION
  VA  7: 0x00000327 (decimal:  807) --> SEGMENTATION VIOLATION
  VA  8: 0x00000060 (decimal:   96) --> VALID: 0x00003ffc (decimal: 16380)
  VA  9: 0x0000001d (decimal:   29) --> VALID: 0x00003fb9 (decimal: 16313)

CS314 OS lab/Submissions/Lab-7 $ python2 relocation.py -s 1 -n 10 -l 100 -c -b 16285

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

Base-and-Bounds register information:

  Base   : 0x00003f9d (decimal 16285)
  Limit  : 100

Error: address space does not fit into physical memory with those base/bounds values.
Base + Limit: 16385   Psize: 16384
```

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

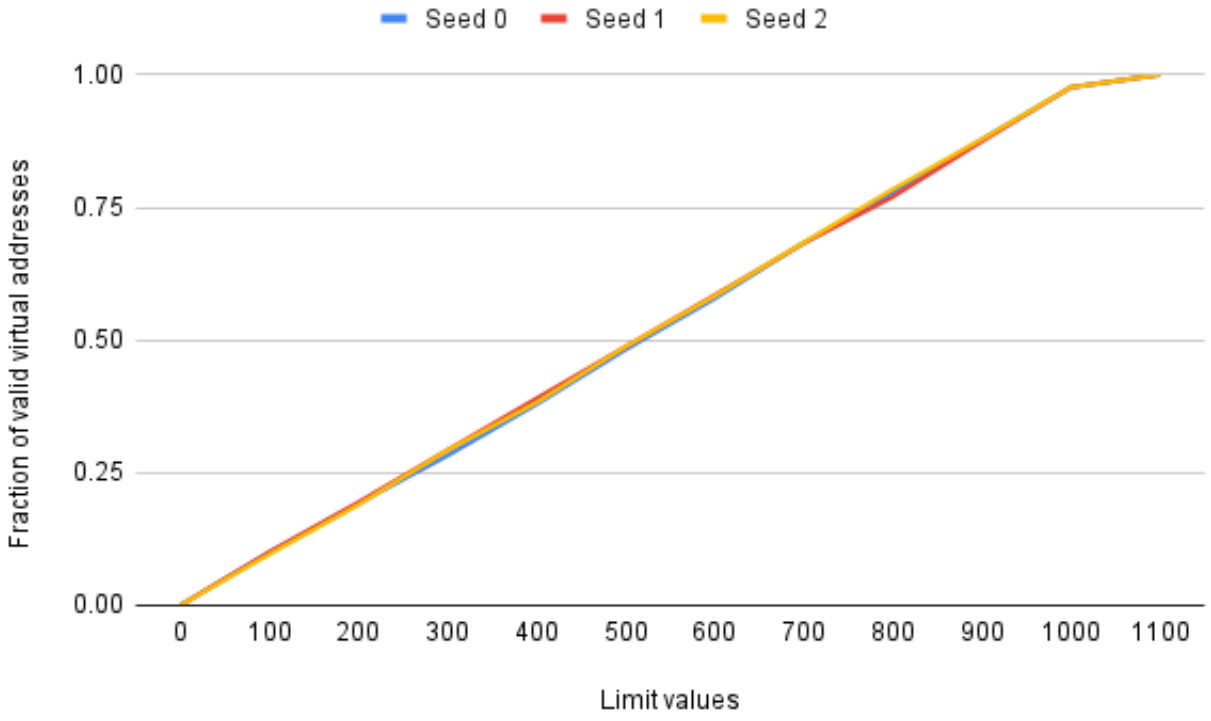
Here, we have set the -s to 1, -n to -10, -l to 10000, -p to 256m and -a to 10k for the first case and 50k for the second case. We can see that as limit size is almost close to address space size, all the VA are in bounds (Valid) and if we increase the -a to 50k, only few of them will be in bounds and rest of them will be out of bounds (SEGMENTATION VIOLATION) as limit size (10K) is smaller compared to address space size here.

```
● CS314 OS lab/Submissions/Lab-7 $ python2 relocation.py -s 1 -n 10 -l 10000 -c -a 10k -p 256m

  ARG seed 1
  ARG address space size 10k
  ARG phys mem size 256m

  Base-and-Bounds register information:

    Base   : 0x02265b1f (decimal 36068127)
    Limit  : 10000

  Virtual Address Trace
    VA  0: 0x000021e5 (decimal: 8677) --> VALID: 0x02267d04 (decimal: 36076804)
    VA  1: 0x00001e8d (decimal: 7821) --> VALID: 0x022679ac (decimal: 36075948)
    VA  2: 0x00000a33 (decimal: 2611) --> VALID: 0x02266552 (decimal: 36070738)
    VA  3: 0x000013d1 (decimal: 5073) --> VALID: 0x02266ef0 (decimal: 36073200)
    VA  4: 0x000011fa (decimal: 4602) --> VALID: 0x02266d19 (decimal: 36072729)
    VA  5: 0x00001a10 (decimal: 6672) --> VALID: 0x0226752f (decimal: 36074799)
    VA  6: 0x00001f8c (decimal: 8076) --> VALID: 0x02267aab (decimal: 36076203)
    VA  7: 0x000003c1 (decimal:  961) --> VALID: 0x02265ee0 (decimal: 36069088)
    VA  8: 0x00000122 (decimal:  290) --> VALID: 0x02265c41 (decimal: 36068417)
    VA  9: 0x0000216e (decimal: 8558) --> VALID: 0x02267c8d (decimal: 36076685)
```

```
● CS314 OS lab/Submissions/Lab-7 $ python2 relocation.py -s 1 -n 10 -l 10000 -c -a 50k -p 256m

  ARG seed 1
  ARG address space size 50k
  ARG phys mem size 256m

  Base-and-Bounds register information:

    Base   : 0x02265b1f (decimal 36068127)
    Limit  : 10000

  Virtual Address Trace
    VA  0: 0x0000a97c (decimal: 43388) --> SEGMENTATION VIOLATION
    VA  1: 0x000098c1 (decimal: 39105) --> SEGMENTATION VIOLATION
    VA  2: 0x00003303 (decimal: 13059) --> SEGMENTATION VIOLATION
    VA  3: 0x00006316 (decimal: 25366) --> SEGMENTATION VIOLATION
    VA  4: 0x000059e5 (decimal: 23013) --> SEGMENTATION VIOLATION
    VA  5: 0x00008251 (decimal: 33361) --> SEGMENTATION VIOLATION
    VA  6: 0x00009dbe (decimal: 40382) --> SEGMENTATION VIOLATION
    VA  7: 0x000012c5 (decimal: 4805) --> VALID: 0x02266de4 (decimal: 36072932)
    VA  8: 0x000005ab (decimal: 1451) --> VALID: 0x022660ca (decimal: 36069578)
    VA  9: 0x0000a727 (decimal: 42791) --> SEGMENTATION VIOLATION
```

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.

Here, we will keep all other values at their default values. We see linear trend in the graph as expected, as limit size increases fraction of randomly-generated virtual addresses number also increases. For the boundry values of limit, if it is 0 then fraction will be 0 and if it is 1024 (1k = address space size) then fraction will be 1.



2 Part 2

The program `segmentation.py` allows you to see how address translations are performed in a system with segmentation. See the README_segmentation for more details.

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?

```
● CS314 OS lab/Submissions/Lab-7 $ python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -c -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) --> 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)
```

```
● CS314 OS lab/Submissions/Lab-7 $ python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -c -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) --> 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)
```

```
● CS314 OS lab/Submissions/Lab-7 $ python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -c -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) --> 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. 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?

For all seeds :
Highest legal VA in segment 0 : 19
Lowest legal VA in segment 1 : 108
Highest illegal address (VA) : 107
Lowest illegal address (VA): 20
Highest illegal address (PA) : 491
Lowest illegal address (PA): 20

Command :

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

```
CS314 OS lab/Submissions/Lab-7 $ python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -c -s 0 -A 19,108,20,107
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)
```

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?

For segment 0, Base : 0 Bound : 2

For segment 1, Base : 128 Bound : 2

Base can have different values but we have to make sure of the bound value that it should be 2 also segment 0 and segment 1 shouldn't overlap.

```
CS314 OS lab/Submissions/Lab-7 $ 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=128 --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) : 0x00000080 (decimal 128)
  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: 0x0000007e (decimal:  126)
  VA 15: 0x0000000f (decimal:   15) --> VALID in SEG1: 0x0000007f (decimal:  127)
```

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?

For 90% of the randomly-generated virtual addresses to be valid we will have to take care of the situation such that roughly 90% are allocated to the physical memory. We can ensure this by $b + B = 0.9 * a$, where a is address space size and b & B are bounds for segment 0 & segment 1 respectively. Basically, total size of bounds should be 0.9 times the address space size.

```
CS314 OS lab/Submissions/Lab-7 $ python2 segmentation.py -a 10 -p 56 -c -b 0 -l 5 -B 512 -L 4 -n 10
ARG seed 0
ARG address space size 10
ARG phys mem size 56

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 5

  Segment 1 base  (grows negative) : 0x00000200 (decimal 512)
  Segment 1 limit                  : 4

Virtual Address Trace
  VA  0: 0x00000008 (decimal:    8) --> VALID in SEG1: 0x000001fe (decimal:  510)
  VA  1: 0x00000007 (decimal:    7) --> VALID in SEG1: 0x000001fd (decimal:  509)
  VA  2: 0x00000004 (decimal:    4) --> VALID in SEG0: 0x00000004 (decimal:    4)
  VA  3: 0x00000002 (decimal:    2) --> VALID in SEG0: 0x00000002 (decimal:    2)
  VA  4: 0x00000005 (decimal:    5) --> SEGMENTATION VIOLATION (SEG1)
  VA  5: 0x00000004 (decimal:    4) --> VALID in SEG0: 0x00000004 (decimal:    4)
  VA  6: 0x00000007 (decimal:    7) --> VALID in SEG1: 0x000001fd (decimal:  509)
  VA  7: 0x00000003 (decimal:    3) --> VALID in SEG0: 0x00000003 (decimal:    3)
  VA  8: 0x00000004 (decimal:    4) --> VALID in SEG0: 0x00000004 (decimal:    4)
  VA  9: 0x00000005 (decimal:    5) --> SEGMENTATION VIOLATION (SEG1)
```

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

For the simulator to have no addresses to be valid, we can st both the bounds to be 0, so that they won't have any allocation on physical memory (no translation to PA).

```
CS314 OS lab/Submissions/Lab-7 $ python2 segmentation.py -a 128 -p 512 -c -n 5 -b 0 -l 0 -B 512 -L 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                  : 0

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

Virtual Address Trace
  VA  0: 0x0000006c (decimal:  108) --> SEGMENTATION VIOLATION (SEG1)
  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)
```

3 Part 3

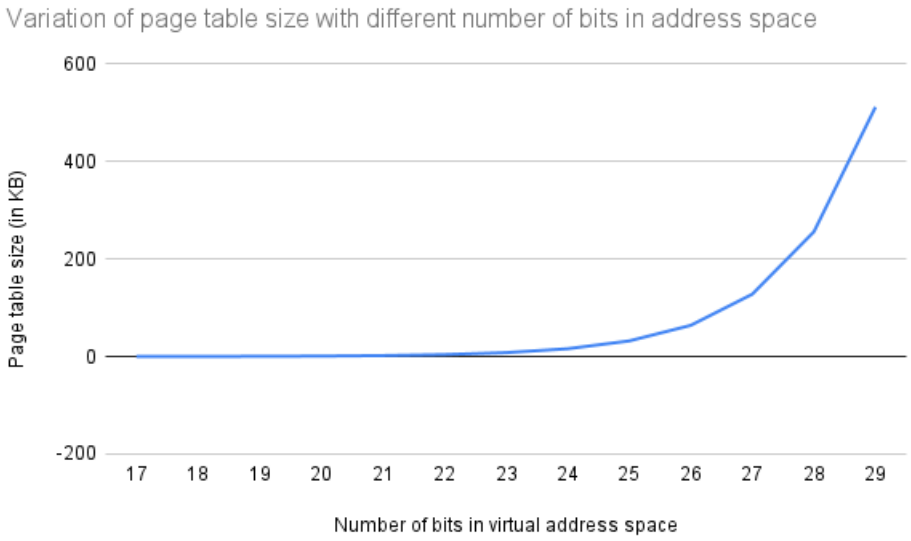
Q3.The program `paging-linear-size.py` lets you figure out the size of a linear page table given a variety of input parameters.

1. 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.

(1) Variation of page table size with different number of bits in address space

We can see that as we increase the address space by 1 bit then size of page table gets doubled (multiplied by 2).

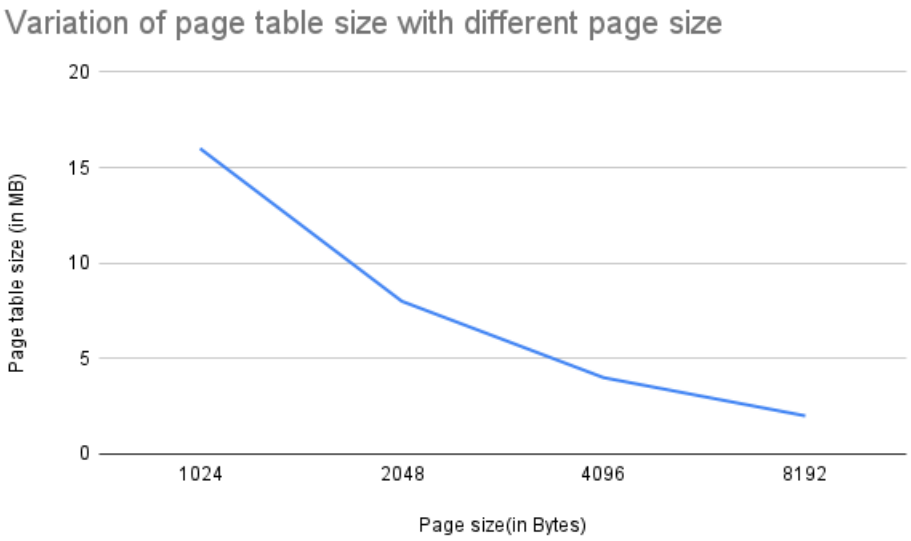
```
CS314 OS lab/Submissions/Lab-7 $ python2 paging-linear-size.py -v 24 -c
ARG bits in virtual address 24
ARG page size 4k
ARG pte size 4

Recall that an address has two components:
[ Virtual Page Number (VPN) | Offset ]

The number of bits in the virtual address: 24
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 12
Thus, a virtual address looks like this:

V V V V V V V V V V V V | O O O O O O O O O O O O

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 4096.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  16384 bytes
  in KB: 16.0
  in MB: 0.015625
```



(2) Variation of page table size with different page size

We can see that as we double the page size, the size of page table gets half (divided by 2) each time.

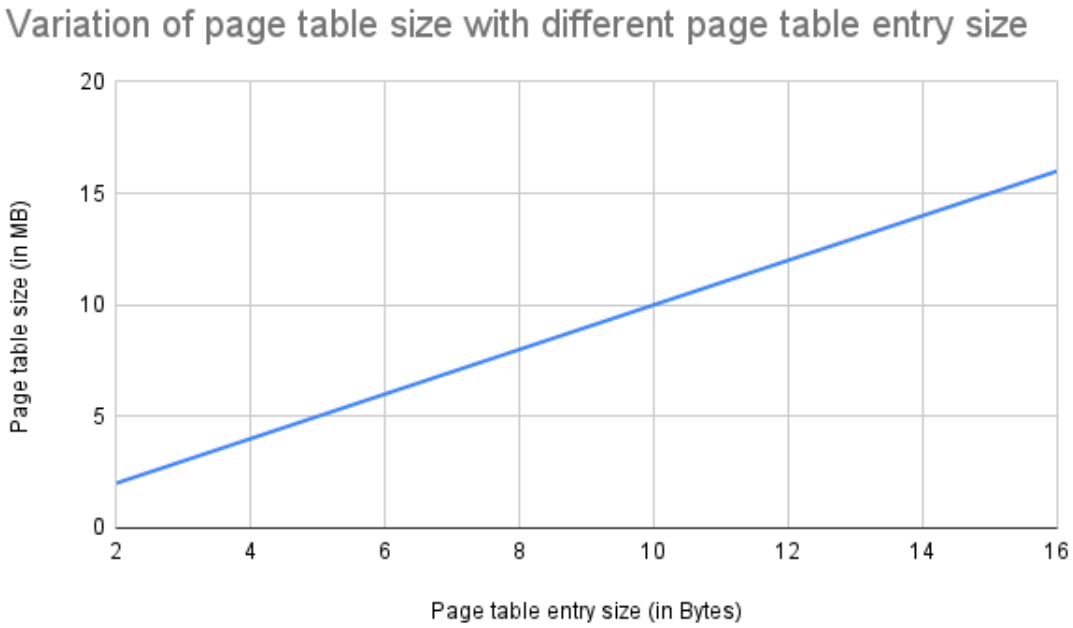
```
CS314 OS lab/Submissions/Lab-7 $ python2 paging-linear-size.py -c -p 4k
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 4

Recall that an address has two components:
[ Virtual Page Number (VPN) | Offset ]

The number of bits in the virtual address: 32
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 20
Thus, a virtual address looks like this:

V V V V V V V V V V V V V V V V V V V V | O O O O O O O O O O O O

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 1048576.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  4194304 bytes
  in KB: 4096.0
  in MB: 4.0
```



(3) Variation of page table size with different page table entry size

We can see that as we increase the page table entry size, the size of page table gets increased linearly.

```
CS314 OS lab/Submissions/Lab-7 $ python2 paging-linear-size.py -c -e 4
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 4

Recall that an address has two components:
[ Virtual Page Number (VPN) | Offset ]

The number of bits in the virtual address: 32
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 20
Thus, a virtual address looks like this:

V V V V V V V V V V V V V V V V V V V V | O O O O O O O O O O O O

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 1048576.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  4194304 bytes
  in KB: 4096.0
  in MB: 4.0
```

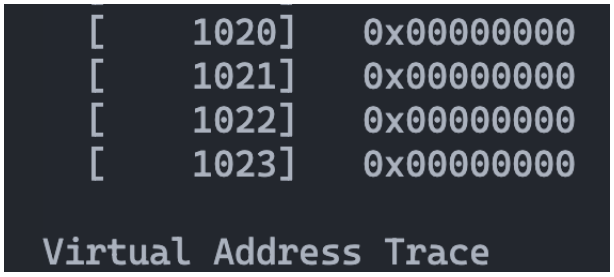

Variation of page table size with different page table entry size
Page table size (in MB)
20
15
10
5
0
2
4
6
8
10
12
14
16
Page table entry size (in Bytes)

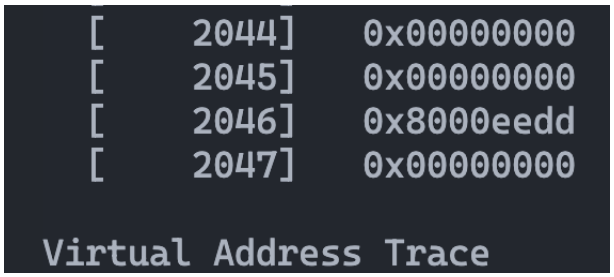
4 Part 4

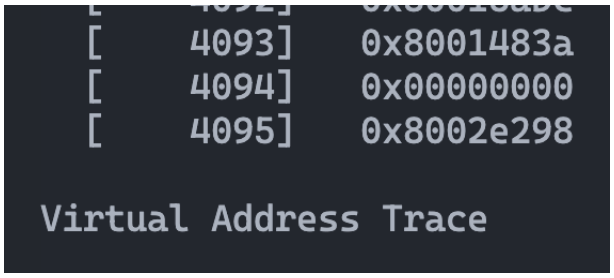
You will use the program, `paging-linear-translate.py` to see if you understand how simple virtual-to-physical address translation works with linear page tables. See the README_paging for more details.

1. Before doing any translations, lets 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. filled. First, to understand how linear page table size changes as the address space grows, run with these flags:

Linear page table size increases linearly with increase in address space. Below are the image results for a = 1m, 2m 4m respectively.

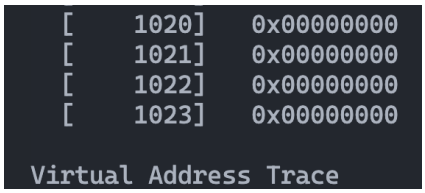


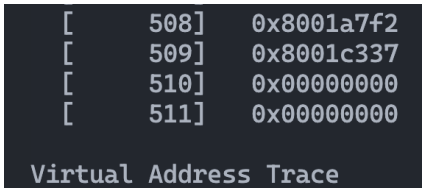


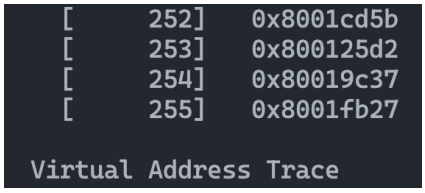


Then, to understand how linear page table size changes as page size grows.

As page size is doubled, linear page table size gets halved (divided by 2). Below are the image results for P = 1k, 2k 4k respectively.







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?

For changing the address space, page-table size grows linearly with that (as we need more pages to cover the whole address space) as we know the linear page table entries are sized to hold the physical memory address, so they grows in proportion with the physical address space. (no_of_entries = address_space_size / page_size)

For doubling the page size, page-table size should be halved as address space size still remains the same, so by doubling page size, we are reducing the number of pages in that address by half. And we know that, the number of entries in the linear page table grows in proportion to the size of the address space (we need less pages as they are bigger in size to cover the whole address space).

We should not use really big pages in general because it would be a lot of waste of memory, most processes use very little memory.

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

As you increase the percentage of pages that are allocated in each address space, the number of valid memory access operations increase.

For u = 0 : all the addresses are invalid as none of the pages are allocated.

For u = 25: Only VA 0x2bc6 is valid. Address space -> 16KB (2^{14}) so we have 14 bits in total. Page size -> 1KB, 10 bits needed to move around in our page. So, other 4 bits used as index to page table. Binary of 0x2bc6 is 10 1011 1100 0110. So, the VPN is 1010 mean the value of 10th index of Page Table is PFN. The Offset is 11 1100 0110. By Looking into page table we get 0x13(10011) at 10th index. However, we have to left shift it 10 times (number of offset bits) So, it becomes 0100 1100 0000 0000. Then, we have to OR this with out offset (0011 1100 0110 OR 0100 1100 0000 0000). It finally becomes 0x4fc6.

For u = 50: First valid VA is 0x3385. Binary of 0x3385 is 11 0011 1000 0101. So, the VPN is 1100 mean the value of 12th index of Page Table is PFN. The Offset is 11 1000 0101. By Looking into page table we get 0x0f(1111) at 12th index. However, we have to left shift it 10 times (number of offset bits) So, it becomes 0011 1100 0000 0000. Then, we have to OR this with out offset (11 1000 0101 OR 0011 1100 0000 0000). It finally becomes 0x3f85.

For u = 50: Second valid VA is 0x00e6. Binary of 0x00e6 00 0000 1110 0110. So, the VPN is 0000 mean the value of 0th index of Page Table is PFN. The Offset is 00 1110 0110. By Looking into page table we get 0x18(11000) at 0th index. However, we have to left shift it 10 times (number of offset bits) So, it becomes 0110 0000 0000 0000. Then, we have to OR this with out offset (00 1110 0110 OR 0110 0000 0000 0000). It finally becomes 0x60e6.

For u = 50: Third valid VA is 0x1986. Binary of 0x1986 01 1001 1000 0110. So, the VPN is 0110 mean the value of 6th index of Page Table is PFN. The Offset is 01 1000 0110. By Looking into page table we get 0x1d(11101) at 0th index. However, we have to left shift it 10 times (number of offset bits) So, it becomes 0111 0100 0000 0000. Then, we have to OR this with out offset (01 1000 0110 OR 0111 0100 0000 0000). It finally becomes 0x7586.

Similarly, we can show for the rest of cases of u, where the value is 75 and 100.

```
● CS314 OS lab/Submissions/Lab-7 $ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -c -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) -->  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)
```

```
● CS314 OS lab/Submissions/Lab-7 $ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -c -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) -->  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)
```

```
● CS314 OS lab/Submissions/Lab-7 $ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -c -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) --> 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]
```

```
● CS314 OS lab/Submissions/Lab-7 $ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -c -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) --> 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]
```

```
● CS314 OS lab/Submissions/Lab-7 $ python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -c -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) --> 00004e0f (decimal    19983) [VPN 11]
    VA 0x00001986 (decimal:     6534) --> 00007d86 (decimal    32134) [VPN 6]
    VA 0x000034ca (decimal:    13514) --> 00006cca (decimal    27850) [VPN 13]
    VA 0x00002ac3 (decimal:    10947) --> 00000ec3 (decimal     3779) [VPN 10]
    VA 0x00000012 (decimal:       18) --> 00006012 (decimal    24594) [VPN 0]
```

3.Now let's try some different random seeds, and some different (and sometimes quite crazy) address-space parameters. Which of these parameter combinations are unrealistic? Why?

The argument of the first command is unrealistic, as the size of page is too small. Also, only 4 pages at max can be present in address space, which is very less.

For second set of flags, only 4 pages at max can be present in address space. which is less in number compared to reality and hence can be said to be unrealistic but it is slightly more realistic than first case

The argument of the third command is unrealistic. the size of page is too large.

```
CS314 OS lab/Submissions/Lab-7 $ python2 paging-linear-translate.py -P 8 -a 32 -p 1024 -v -c -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) --> 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)
```

```
● CS314 OS lab/Submissions/Lab-7 $ python2 paging-linear-translate.py -P 8k -a 32k -p 1m -v -c -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) -->  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)
```

```
● CS314 OS lab/Submissions/Lab-7 $ python2 paging-linear-translate.py -P 1m -a 256m -p 512m -v -c -s 3
  ARG seed 3
  ARG address space size 256m
  ARG phys mem size 512m
  ARG page size 1m
  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]   0x800000bd
    [       2]   0x80000140
    [       3]   0x00000000
    [       4]   0x00000000
    [       5]   0x80000084
    [       6]   0x00000000
    [       7]   0x800000f0
    [     248]   0x800000f6
    [     249]   0x00000000
    [     250]   0x800001eb
    [     251]   0x00000000
    [     252]   0x00000000
    [     253]   0x00000000
    [     254]   0x80000159
    [     255]   0x00000000

  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]
```

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 mem-ory?

There can different cases where the given simulator program doesn't work :

- When we specify address-space size or physical memory size less than 1 (or negative).
- When we specify physical memory size lower than address space size.
- When we specify address space or physical memory not a multiple of the pagesize for this simulation.
- When we specify greater (more than 1 GB) sizes (address-space size physical memory size) for this simulation.
- When we specify page size greater than address-space.

```
⊗ CS314 OS lab/Submissions/Lab-7 $ python2 paging-linear-translate.py -P 8 -a 32 -p 8 -v -c
  ARG seed 0
  ARG address space size 32
  ARG phys mem size 8
  ARG page size 8
  ARG verbose True
  ARG addresses -1

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

```
⊗ CS314 OS lab/Submissions/Lab-7 $ python2 paging-linear-translate.py -P 8 -a 512m -p 1g -v -c
  ARG seed 0
  ARG address space size 512m
  ARG phys mem size 1g
  ARG page size 8
  ARG verbose True
  ARG addresses -1

  Error: must use smaller sizes (less than 1 GB) for this simulation.
```