CS314: Lab Assignment 7

B Siddharth Prabhu
200010003@iitdh.ac.in

05 March 2023

1 Address Translation with Base and Bounds Registers

The program `relocation.py` allows us to see how address translations are performed in a system with base and bounds registers. The file `README_base_bound` contains more details.

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

The seed randomizes the choices of base, bound (limit), and Virtual Address Trace. References to any address within the bounds generated would be considered legal; references above this value are out of bounds and thus the hardware would raise an exception. Figures (1), (2), and (3) depict the observed translations.

```
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7/cs314oslaboratory7$
python3 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
```

Figure 1: Seed set to 1

```
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7/cs314oslaboratory7$
python3 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
```

Figure 2: Seed set to 2

```
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7/cs314oslaboratory7$
python3 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)
```

Figure 3: Seed set to 3

(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?

The given flags run the code with seed set to 0, and result in some segmentation violations due to out-of-bound references, as seen in Figure (4). To ensure that all the generated virtual addresses are within bounds, we set the limit to 1 more than the maximum encountered virtual address in the trace, which results in `930` . The code is run with bounds set to this value as shown in Figure (5).

```
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7/cs314oslaboratory7$
 python3 relocation.py -s 0 -n 10 -c

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
```

Figure 4: Seed set to 0, Limit default

```
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7/cs314oslaboratory7$
 python3 relocation.py -s 0 -n 10 -l 930 -c

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)
```

Figure 5: Seed set to 0, Limit 930

(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 the physical memory size is 16 kB, which is $16 \times 1024 = 16384$ Bytes. Also, note that addresses within the bound are less than the sum of 'base' and 'limit'. In the given flags, limit has been set to 100. Hence, we have the equation, $base + 100 \leq 16384$. Hence, the base is at most `16284` . The code has been run with and without the base set to this value, as shown in Figures (6) and (7).

```
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7/cs314oslaboratory7$
 python3 relocation.py -s 1 -n 10 -l 100 -c

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) --> SEGMENTATION VIOLATION
  VA  1: 0x0000030e (decimal:  782) --> SEGMENTATION VIOLATION
  VA  2: 0x00000105 (decimal:  261) --> SEGMENTATION VIOLATION
  VA  3: 0x000001fb (decimal:  507) --> SEGMENTATION VIOLATION
  VA  4: 0x000001cc (decimal:  460) --> SEGMENTATION VIOLATION
  VA  5: 0x0000029b (decimal:  667) --> SEGMENTATION VIOLATION
  VA  6: 0x00000327 (decimal:  807) --> SEGMENTATION VIOLATION
  VA  7: 0x00000060 (decimal:   96) --> VALID: 0x000008f9 (decimal: 2297)
  VA  8: 0x0000001d (decimal:   29) --> VALID: 0x000008b6 (decimal: 2230)
  VA  9: 0x00000357 (decimal:  855) --> SEGMENTATION VIOLATION
```

Figure 6: Code run with given flags

```
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7/cs314oslaboratory7$
 python3 relocation.py -s 1 -n 10 -l 100 -b 16285 -c

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
```

Figure 7: Base set to 16285, which fails

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

On running with larger address spaces, we must keep in mind that in this simulation, size of address space should be smaller than size of virtual memory, and that `-l` could be modified to cover more addresses in the bound.

In Figure (8), we have set base to 50000, with address space of 1 MB and physical address space of 1 GB. In Figure (9), we recreated the previous question, but with address space set to 3 kB while physical memory is of 1 GB.

```
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7/cs314oslaboratory7$
 python3 relocation.py -s 1 -n 10 -l 500000 -a 1m -p 1g -c

ARG seed 1
ARG address space size 1m
ARG phys mem size 1g

Base-and-Bounds register information:

  Base   : 0x08996c7c (decimal 144272508)
  Limit  : 500000

Virtual Address Trace
  VA  0: 0x000d8f16 (decimal: 888598) --> SEGMENTATION VIOLATION
  VA  1: 0x000c386b (decimal: 800875) --> SEGMENTATION VIOLATION
  VA  2: 0x000414c3 (decimal: 267459) --> VALID: 0x089d813f (decimal: 144539967)
  VA  3: 0x0007ed4d (decimal: 519501) --> SEGMENTATION VIOLATION
  VA  4: 0x0007311d (decimal: 471325) --> VALID: 0x08a09d99 (decimal: 144743833)
  VA  5: 0x000a6cec (decimal: 683244) --> SEGMENTATION VIOLATION
  VA  6: 0x000c9e9c (decimal: 827036) --> SEGMENTATION VIOLATION
  VA  7: 0x00018072 (decimal: 98418) --> VALID: 0x089aecee (decimal: 144370926)
  VA  8: 0x0000741c (decimal: 29724) --> VALID: 0x0899e098 (decimal: 144302232)
  VA  9: 0x000d5f4b (decimal: 876363) --> SEGMENTATION VIOLATION
```

Figure 8: Flags: `-s 1 -n 10 -l 500000 -a 1m -p 1g -c`

```
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7/cs314oslaboratory7$
 python3 relocation.py -s 1 -n 10 -l 100 -a 3k -p 1g -c

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

Base-and-Bounds register information:

  Base   : 0x08996c7c (decimal 144272508)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x00000a2b (decimal: 2603) --> SEGMENTATION VIOLATION
  VA  1: 0x0000092a (decimal: 2346) --> SEGMENTATION VIOLATION
  VA  2: 0x0000030f (decimal:  783) --> SEGMENTATION VIOLATION
  VA  3: 0x000005f1 (decimal: 1521) --> SEGMENTATION VIOLATION
  VA  4: 0x00000564 (decimal: 1380) --> SEGMENTATION VIOLATION
  VA  5: 0x000007d1 (decimal: 2001) --> SEGMENTATION VIOLATION
  VA  6: 0x00000976 (decimal: 2422) --> SEGMENTATION VIOLATION
  VA  7: 0x00000120 (decimal:  288) --> SEGMENTATION VIOLATION
  VA  8: 0x00000057 (decimal:   87) --> VALID: 0x08996cd3 (decimal: 144272595)
  VA  9: 0x00000a07 (decimal: 2567) --> SEGMENTATION VIOLATION
```

Figure 9: Flags: `-s 1 -n 10 -l 100 -a 3k -p 1g -c`

(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.

We compute this fraction over varying limit values, for different seeds, by modifying the code to include a counter variable that increments for every valid value, and is finally used to calculate the required fraction. Larger datasets are better to make judgements, so we generate 10000 virtual addresses each time, using `-n 10000` . We observe that the required function, that maps value of bounds register to fraction of valid generated virtual addresses is roughly:

$$f(l) = \begin{cases} 0 & \text{if } l \leq 0 \\ 0.0489 \times l & \text{if } l \in (0, 1024) \\ 1 & \text{if } l \geq 1024 \end{cases}$$

This claim is substantiated by Figure (10), where the plots for all three considered seeds are extremely similar, to the point where width of the plot must be reduced to discern any difference.

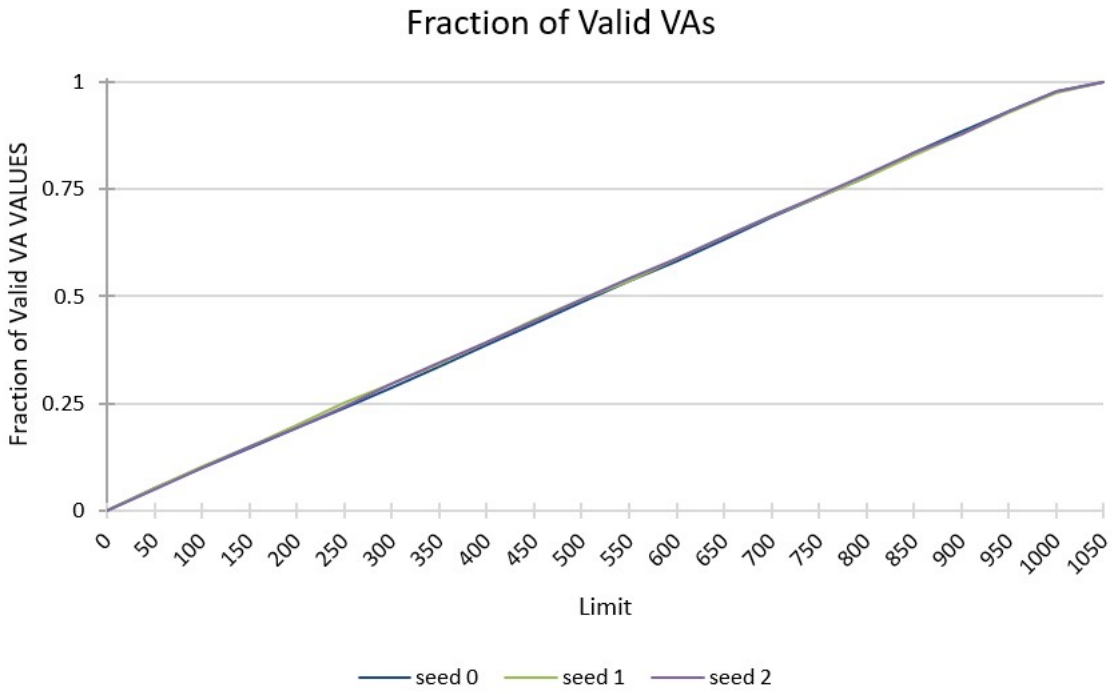

Figure 10: Plot of Fraction of Valid VAs against varying limit values

2 Address Translation with Segmentation

The program `segmentation.py` allows us to see how address translations are performed in a system with segmentation. The file `README_segmentation` contains 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?

```
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
```

In all cases, the start of the address space maps to segment 0 in the physical memory, which has base 0 and limit 20. The last addresses in the address space map to segment 1, which has base 512 and limit 20. Therefore, $\{0, 1, \ldots, 19\}$ (VA) map to $\{0, 1, \ldots, 19\}$ (PA), and $\{108, 110, \ldots, 127\}$ (VA) map to $\{492, 494, \ldots, 511\}$ (PA). So, observe that only the addresses in $\{0, 1, \ldots, 19\} \cup \{108, 110, \ldots, 127\}$ (Let us refer to this as set S) would have valid addresses in the physical memory.

For seed 0, the virtual addresses generated (in decimal form) are 108, 97, 53, 33, 65. Of these, only 108 belongs to the set S. As per the deductions above, the translation is $(108 \rightarrow 492)$.

For seed 1, the virtual addresses generated (in decimal form) are 17, 108, 97, 32, 63. Of these, 17 and 108 belong to the set S. As per the deductions above, the translation is $(17 \rightarrow 17)$ and $(108 \rightarrow 492)$.

For seed 2, the virtual addresses generated (in decimal form) are 122, 121, 7, 10, 106. Of these, 122, 121, 7, and 10 belong to the set S. As per the deductions above, the translation is $(122 \rightarrow 506)$, $(121 \rightarrow 505)$, $(7 \rightarrow 7)$, $(10 \rightarrow 10)$.

We can slightly formalize these findings by devising a function that gives the physical address on passing the virtual address as input:

$$f(v) = \begin{cases} v & \text{if } v \in \{0, 1, ..., 19\} \text{ (i.e., Segment 0)} \\ 512 - (128 - v) & \text{if } v \in \{108, 110, ..., 127\} \text{ (i.e., Segment 1)} \\ \text{undefined} & \text{otherwise} \end{cases}$$

So, the range of f (set of legal physical addresses) is $\{0, 1, \dots, 19\} \cup \{492, 493, \dots, 511\}$.

(2) 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?

The following observations are made, quite easily based on the function we defined in the previous question:

- Highest legal virtual address in segment 0 is `19`.
- Lowest legal virtual address in segment 1 is `108`.
- Lowest illegal (virtual) address in this address space is `20`.
- Highest illegal (virtual) address in this address space is `107`.

Additionally, regarding accessible physical addresses using the function:

- Lowest illegal (physical) address in this address space is `20`.
- Highest illegal (physical) address in this address space is `491`.

To test if these are correct, we can run the following code:

```
python3 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1 -A 19,108,20,107 -c
```

We observe that the answers above are correct, as shown in Figure (11).

```
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7/cs314oslaboratory7$
 python3 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1 -A 19,108,20,107 -c
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: 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)
```

Figure 11: Verifying Answers to Q2.2

(3) Consider 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 ?
```

We must set the base and bounds as follows for the specified address stream (valid, valid, violation, ..., violation, valid, valid):

- Segment 0 : Base = 0 , Bound = 2
- Segment 1 : Base = 128 , Bound = 2

We can check if this works by executing the program as follows:

```
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
```

The output obtained is shown in Figure (12). (Backslash in the above code is to be able to directly paste in terminal).

```
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7/cs314oslaboratory7/q2$
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 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)
```

Figure 12: Verifying Answers to Q2.3

(4) Assume we want to generate a problem where roughly 90% of the randomly-generated virtual addresses are valid. How should you configure the simulator to do so? Which parameters are important to getting this outcome?

To ensure that roughly 90% of the randomly-generated virtual addresses are valid, we must set valid memory size to be equal to 90% of the whole virtual memory size. In the context of segmentation in a system with virtual address space of size a, if b and B are the bounds for each of the segments, then we need to satisfy the equation: $b + B \approx 0.9 \times a$. In practice, we observe that the sum $b + B$ must be a tiny bit greater than $0.9 \times a$. An example of the same would be:

```
python3 segmentation.py -a 10000 -p 128m -s 1 \
    --b0 0 --l0 4547 --b1 128 --l1 4547 -n 10 -c
```

```
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7/cs314oslaboratory7/q2$
 python3 segmentation.py -a 10000 -p 128m -s 1 --b0 0 --l0 4547 --b1 128 --l1 4547 -n 10 -c
ARG seed 1
ARG address space size 10000
ARG phys mem size 128m

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 4547

  Segment 1 base  (grows negative) : 0x00000080 (decimal 128)
  Segment 1 limit                  : 4547

Virtual Address Trace
  VA  0: 0x0000053f (decimal: 1343) --> VALID in SEG0: 0x0000053f (decimal: 1343)
  VA  1: 0x0000211a (decimal: 8474) --> VALID in SEG1: 0x-0000576 (decimal: -1398)
  VA  2: 0x00001dd5 (decimal: 7637) --> VALID in SEG1: 0x-00008bb (decimal: -2235)
  VA  3: 0x000009f6 (decimal: 2550) --> VALID in SEG0: 0x000009f6 (decimal: 2550)
  VA  4: 0x0000135a (decimal: 4954) --> SEGMENTATION VIOLATION (SEG0)
  VA  5: 0x0000118e (decimal: 4494) --> VALID in SEG0: 0x0000118e (decimal: 4494)
  VA  6: 0x00001973 (decimal: 6515) --> VALID in SEG1: 0x-0000d1d (decimal: -3357)
  VA  7: 0x00001ecf (decimal: 7887) --> VALID in SEG1: 0x-00007c1 (decimal: -1985)
  VA  8: 0x000003aa (decimal:  938) --> VALID in SEG0: 0x000003aa (decimal:  938)
  VA  9: 0x0000011b (decimal:  283) --> VALID in SEG0: 0x0000011b (decimal:  283)

f=  0.9
```

Figure 13: Verifying Answers to Q2.4

(Note that the code has been edited to print the required fraction.)

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

Yes, this can easily be done, simply by setting the bounds of all segments to zero. This ensures that not a single virtual address is valid. The same can be observed in Figure (14). Code for the same is as follows:

```
python3 segmentation.py -a 10000 -p 128m -s 1 --b0 0 --l0 0 --b1 128 --l1 0 -n 10 -c
```

```
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7/cs314oslaboratory7/q2$
 python3 segmentation.py -a 10000 -p 128m -s 1 --b0 0 --l0 0 --b1 128 --l1 0 -n 10 -c
ARG seed 1
ARG address space size 10000
ARG phys mem size 128m

Segment register information:

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

  Segment 1 base  (grows negative) : 0x00000080 (decimal 128)
  Segment 1 limit                  : 0

Virtual Address Trace
  VA  0: 0x0000053f (decimal: 1343) --> SEGMENTATION VIOLATION (SEG0)
  VA  1: 0x0000211a (decimal: 8474) --> SEGMENTATION VIOLATION (SEG1)
  VA  2: 0x00001dd5 (decimal: 7637) --> SEGMENTATION VIOLATION (SEG1)
  VA  3: 0x000009f6 (decimal: 2550) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x0000135a (decimal: 4954) --> SEGMENTATION VIOLATION (SEG0)
  VA  5: 0x0000118e (decimal: 4494) --> SEGMENTATION VIOLATION (SEG0)
  VA  6: 0x00001973 (decimal: 6515) --> SEGMENTATION VIOLATION (SEG1)
  VA  7: 0x00001ecf (decimal: 7887) --> SEGMENTATION VIOLATION (SEG1)
  VA  8: 0x000003aa (decimal:  938) --> SEGMENTATION VIOLATION (SEG0)
  VA  9: 0x0000011b (decimal:  283) --> SEGMENTATION VIOLATION (SEG0)

f=  0.0
```

Figure 14: Verifying Answers to Q2.5

(Note that the code has been edited to print the required fraction.)

3 Linear Page Tables

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

(1) Check how linear page table size varies with #bits in the address space.

The observations of how linear page table size varies with number of bits in the address space can be made by running code such as the following:

```
python3 paging-linear-size.py -v 16 -c
python3 paging-linear-size.py -v 32 -c
python3 paging-linear-size.py -v 64 -c
```

Plotting a graph for the observations, and increasing the number of datapoints at which the page table size is computed, we get the image in Figure (15). Beyond size 1033, an overflow error is thrown by Python due to the exponential blowup in the size of the Page Table. We can formulate a function that is a mapping from number of bits in address space to (linear) page table size in kilobytes, as follows:

$$f(b) = \begin{cases} 2^{b-20} & \text{if } b \in \{12, 13, ...\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that page size is fixed at 4096 bits, which needs 12 bits as minimum number of bits for offset. The code mistakenly assumes that negative number of bits is allowed for number of bits in Virtual Page Number. (The function defined makes sense, since **adding 1 bit doubles the size of addressable space.**)

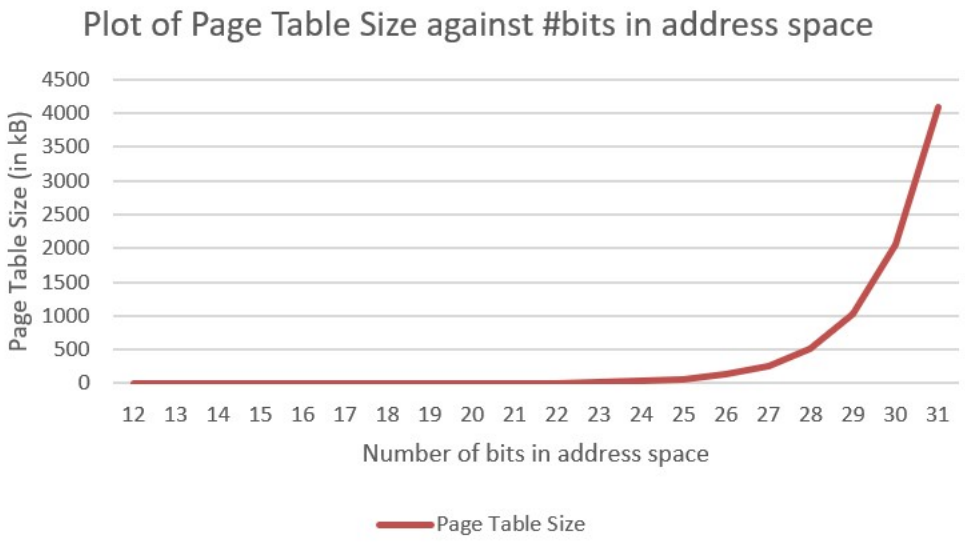

Figure 15: Verifying Answers to Q3.1

(2) Check how linear page table size varies with the page size.

The observations of how linear page table size varies with page size can be made by running code such as the following:

```
python3 paging-linear-size.py -p 16 -c
python3 paging-linear-size.py -p 32 -c
python3 paging-linear-size.py -p 64 -c
```

Plotting a graph for the observations, and increasing the number of data points at which the table size is computed (and taking logarithm with base 2 on each axis for convenience), we get the graph in Figure (16). There is observed to be an exponential decrease in the size of page table with exponential increase in page size. The product of page table size and page size is observed to stay constant at 16777216 when unit is kilobytes. We can formulate a function that is a mapping from page size to page table size, as follows:

$$f(p) = \begin{cases} \frac{16777216}{p} & \text{if } p \in \{x \mid x = 2^n,\ n \in \mathbb{N}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that number of bits in virtual address is fixed at 32 bits here. If the page table size goes beyond $2^{32} = 4294967296$, then the code mistakenly allows negative number of bits for Virtual Page Number. (The above function makes sense since **doubling page size would halve the number of pages, which would in turn halve the page table size**.)

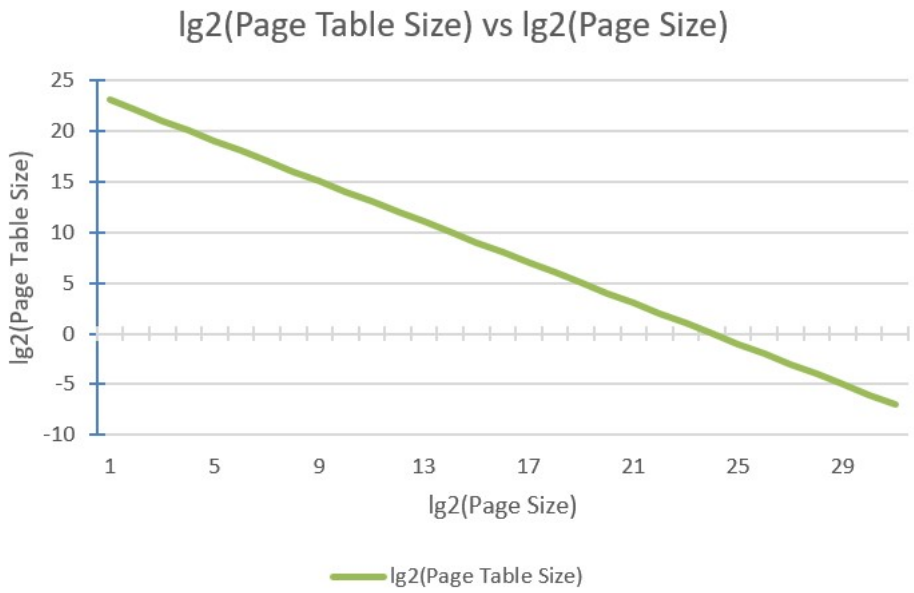

Figure 16: Verifying Answers to Q3.2

(3) Check how linear page table size varies with the page table entry size.

The observations of how linear page table size varies with page table entry size can be made by running code such as the following:

```
python3 paging-linear-size.py -e 4 -c
python3 paging-linear-size.py -e 8 -c
python3 paging-linear-size.py -e 16 -c
```

Plotting a graph for the observations, and increasing the number of data points at which the page table size is computed, we get the plot in Figure (17). The number of entries in the table, is 2^b, where b is the number of bits in Virtual Page Number, which finally evaluates to 1048576. The relation is observed to be linear, and a function can be devised to map page table entry size, e, to page table size, as follows:

$$f(e) = \begin{cases} 1024 \times e & \text{if } p \in \mathbb{N} \cup \{0\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that the code mistakenly allows negative size of page table entry. (The function defined above makes sense since **the quantities are directly related**.)

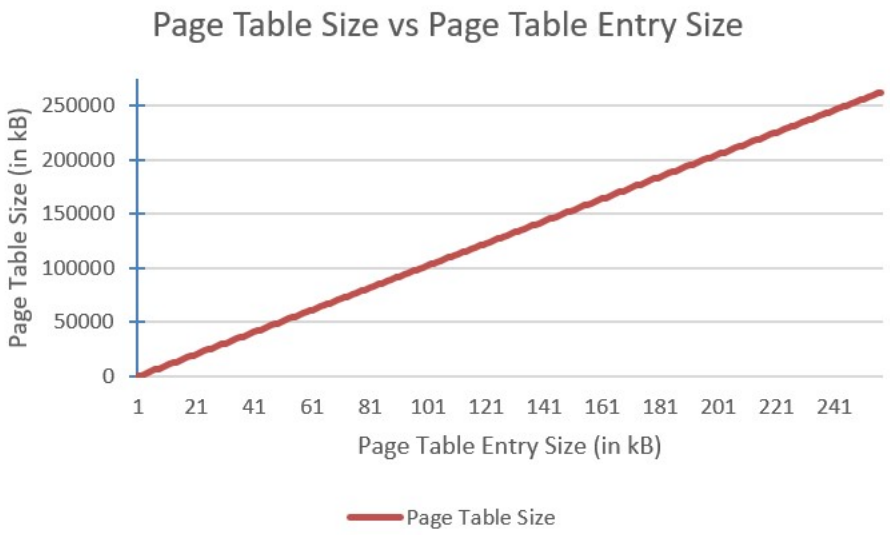

Figure 17: Verifying Answers to Q3.3

(Note: This follows from: PT Size = PTE Size $\times$ $2^{\#\text{bits in VPN}}$, where #bits in VPN is 20)

4 Address Translation with Linear Page Tables

The python script `paging-linear-translate.py` helps us understand how simple address translation works with linear page tables. The file `README_paging_linear_translate` contains more details.

(1) Compute the size of linear page tables as different parameters change. State 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?

Expected Trends: Page table size should increase as the address space increases, since we need more pages to cover the whole address space. Also, when page size increases, the page table size must decrease, as we need less pages to cover the whole address space. Larger page sizes would lead to under-utilization of pages, and wastage of memory.

First, to understand how linear page table size changes as the address space grows, we run with these flags:

```
-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
```

A bash script was made for simplicity in running the code, and we observe that the fraction of filled page table entries decreases very gradually, (0.5068, 0.5054, 0.5034) as seen in Figure (18).

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
```

A bash script was made for easily running the code, and we observe that the fraction of filled pages stays very close to 0.5, as seen in Figure (19).

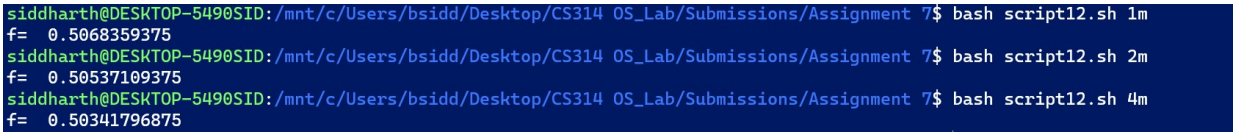

Figure 18: Verifying Answers to Q4.1

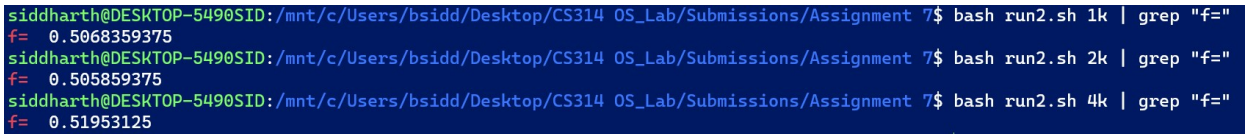

Figure 19: Verifying Answers to Q4.1

(2) Do some translations; change the number of pages that are allocated to the address space with the `-u` flag, as depicted in code below. What happens as you increase the percentage of pages that are allocated in each address space?

```
-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
```

The `-u` flag is used to set the percentage of pages that are allocated in each address space. A bash script was made for simplicity in running the code, and we observe in Figure (20) that the **fraction of valid pages increases** as we increase the percentage of pages allocated. The Figures (21), (22), (23), (24), and (25) depict the Page Table entries for different `-u` inputs. (The respective commands are provided as captions.) Some points to be noted about each case are as follows:

- `-u 0` : None of the virtual addresses have been successfully translated. (None of the corresponding VPNs were not allocated pages.)
- `-u 25` : The address 0x2BC6 has been successfully translated to 0x4FC6. In binary, 0x2BC6 is 10 1011 1100 0110. This is of 14 bits since the address space is 16KB. Then, VPN is 1010 (first 4 bits), since there are 16 pages (This is 10 in decimal), and the offset is 11 1100 0110. So, corresponding PFN from the page table is 0x13 (10011 in binary). We have to left shift this 10 times, since there are 10 bits in offset. This comes to 0100 1100 0000 0000. Finally, OR this with our offset, 0000 0011 1100 0110. The result is 1011 0000 0011 1001. This is 0x4FC6 in hexadecimal, which is the same as what is obtained in Figure (22).
- `-u 50` : There are three successful translations here: ($0x3385 \rightarrow 0x3F85$), ($0x00E6 \rightarrow 0x60E6$), ($0x1986 \rightarrow 0x7D86$). Details of the same are in a below-mentioned table.
- `-u 75` : There are five successful translations here: ($0x2EOF \rightarrow 0x4E0F$), ($0x1986 \rightarrow 0x7D86$), ($0x34CA \rightarrow 0x6CCA$), ($0x2AC3 \rightarrow 0x0EC3$), and ($0x0012 \rightarrow 0x6012$).
- `-u 100` : There are five successful translations here: ($0x2EOF \rightarrow 0x4E0F$), ($0x1986 \rightarrow 0x7D86$), ($0x34CA \rightarrow 0x6CCA$), ($0x2AC3 \rightarrow 0x0EC3$), and ($0x0012 \rightarrow 0x6012$).

The Table (1) depicts the translation of valid addresses for each value of `-u` considered. Note that the address $0x1986$ maps to two different physical addresses in the case of `-u` being set to 50, as opposed to 75 or 100. This is because of the PFN being different in the generated page table.

```
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7$ bash run3.sh 0
f=  0.0
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7$ bash run3.sh 25
f=  0.375
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7$ bash run3.sh 50
f=  0.5625
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7$ bash run3.sh 75
f=  1.0
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7$ bash run3.sh 100
f=  1.0
```

Figure 20: Verifying Answers to Q4.2

```
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

f=  0.0
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)
```

Figure 21: Output for `bash run3.sh 0`

```
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

f=  0.375
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)
```

Figure 22: Output for `bash run3.sh 25`

```
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

f=  0.5625
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]
```

Figure 23: Output for `bash run3.sh 50`

```
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

f=  1.0
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]
```

Figure 24: Output for `bash run3.sh 75`

`-u`	VA (VPN | Offset)	PFN ($<<10$)	Result	In Hex
25	0x2BC6 (10 10|11 1100 0110)	0100 1100 0000 0000	1011 0000 0011 1001	0x4FC6
50	0x3385 (11 00|11 1000 0101)	0011 1100 0000 0000	0011 1111 1000 0101	0x3F85
50	0x00E6 (00 00|00 1110 0110)	0110 0000 0000 0000	0110 0000 1110 0110	0x60E6
50	0x1986 (01 10|01 1000 0110)	0111 0100 0000 0000	0110 0101 1000 0110	0x7586
75/100	0x1986 (01 10|01 1000 0110)	0111 1100 0000 0000	0110 1101 1000 0110	0x7D86
75/100	0x2E0F (10 11|10 0000 1111)	0100 1100 0000 0000	0100 1110 0000 1111	0x4E0F
75/100	0x34CA (11 01|00 1100 1010)	0110 1100 0000 0000	0110 1100 1100 1010	0x6CCA
75/100	0x2AC3 (10 10|10 1100 0011)	0000 1100 0000 0000	0000 1110 1100 0011	0x0EC3
75/100	0x0012 (00 00|00 0001 0010)	0110 0000 0000 0000	0110 0000 0001 0010	0x6012

Table 1: Address Translation

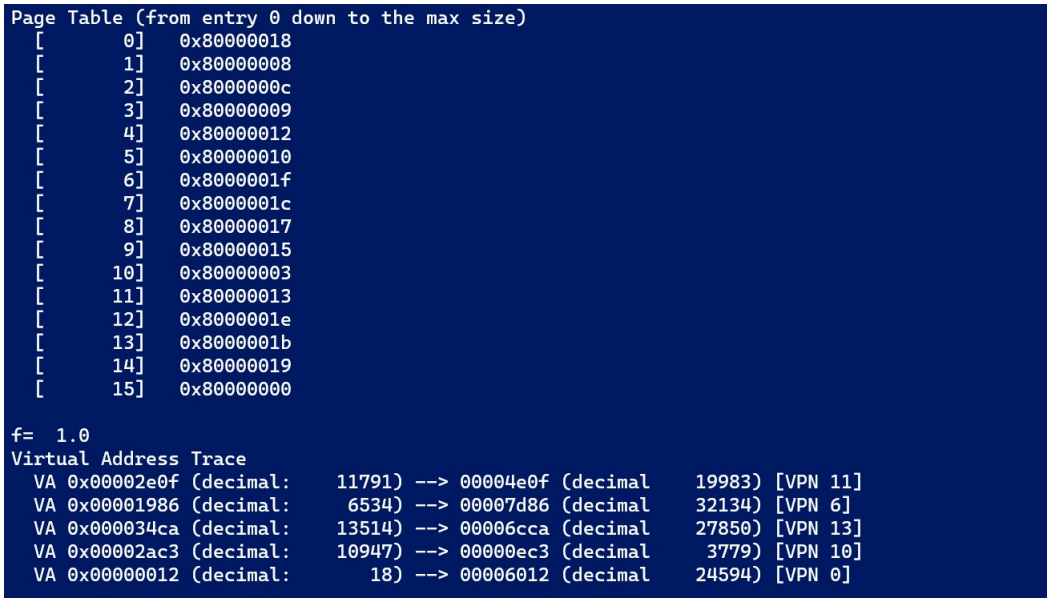

Figure 25: Output for `bash run3.sh 100`

(3) Try some different random seeds, and some different (and sometimes quite crazy) address-space parameters, for variety, like the code given below. Which of these parameter combinations are unrealistic? Why?

```
-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
```

An observation we can make is that the second set of parameters is the same as the first; just scaled by a factor of 2^{10}. In these two cases, the ratio of Page Size to Virtual Address Space to Physical Address Space is $1 : 4 : 128$. This means that the address space can only have 4 pages, which is quite small. Meanwhile, the third set of parameters has the corresponding ratio $1 : 256 : 512$. The Virtual Address space would have 256 pages in this case, each of size 1MB, which is quite high for a page. Hence, remarks that could be made are:

- In the first set of flags, 8B is a very low value to allot to a page. Also, only at max 4 pages can be present in address space. This is fine for the sake of simulation, but if we're going for realism, this is too few, and hence is unrealistic.
- In the second set of flags, only at max 4 pages can be present in address space. This is fine for the sake of simulation, but if we're going for realism, this is too few, and hence is unrealistic (albeit, slightly more realistic than the previous case).
- In the third set of flags, the size of a page is 1MB. This is quite large, and would take up too much memory. (Generally, page sizes are around 256kB.) This could also lead to wastage and under-utilization of memory due to emptiness within pages. Hence, this is unrealistic.

Thus, the degree to which each case is unrealistic can be ordered as $3 > 1 > 2$.

Check Figures (26), (27), and (28) to see the output on running with the above flags. (For the third case, only the last part is included in the figure since it is too large.)

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

f=  0.25
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)
```

Figure 26: Output for `-P 8 -a 32 -p 1024 -v -s 1`

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

f=  0.5
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)
```

Figure 27: Output for `-P 8k -a 32k -p 1m -v -s 2`

```
  [     244]   0x80000049
  [     245]   0x800000f5
  [     246]   0x800000ef
  [     247]   0x800001a4
  [     248]   0x800000f6
  [     249]   0x00000000
  [     250]   0x800001eb
  [     251]   0x00000000
  [     252]   0x00000000
  [     253]   0x00000000
  [     254]   0x80000159
  [     255]   0x00000000

f=  0.46875
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]
```

Figure 28: Output for `-P 1m -a 256m -p 512m -v -s 3`

(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?

If the address space size is bigger than physical memory, the error thrown is that of Figure (29). If non-positive values are set for `-a` or `-p`, or if either of them exceed 1GB, then an error is thrown, as depicted in Figure (30). Also, if either physical memory or address space isn't a multiple of the page size, then the error thrown is as shown in Figure (31). Also, the address space should be a power of 2, or it will throw the error displayed in Figure (32).

```
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7$ python3 paging-linear-transl
ate.py -P 1m -a 1g -p 512m
ARG seed 0
ARG address space size 1g
ARG phys mem size 512m
ARG page size 1m
ARG verbose False
ARG addresses -1

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

Figure 29: Output for `-a > -p`

```
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7$ python3 paging-linear-transl
ate.py -p 0 | grep "Error"
Error: must specify a non-zero physical memory size.
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7$ python3 paging-linear-transl
ate.py -a 0 | grep "Error"
Error: must specify a non-zero address-space size.
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7$ python3 paging-linear-transl
ate.py -p 2g | grep "Error"
Error: must use smaller sizes (less than 1 GB) for this simulation.
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7$ python3 paging-linear-transl
ate.py -a 2g | grep "Error"
Error: physical memory size must be GREATER than address space size (for this simulation)
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7$ python3 paging-linear-transl
ate.py -a 2g -p 4g | grep "Error"
Error: must use smaller sizes (less than 1 GB) for this simulation.
```

Figure 30: More possible errors

```
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7$ python3 paging-linear-transl
ate.py -P 251 -a 12 -p 251 | grep "Error"
Error in argument: address space must be a multiple of the pagesize
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7$ python3 paging-linear-transl
ate.py -P 251 -a 251 -p 12m | grep "Error"
Error in argument: physical memory must be a multiple of the pagesize
```

Figure 31: More possible errors

```
siddharth@DESKTOP-5490SID:/mnt/c/Users/bsidd/Desktop/CS314 OS_Lab/Submissions/Assignment 7$ python3 paging-linear-transl
ate.py -P 25 -a 1000 -p 2000 | grep "Error"
Error in argument: address space must be a power of 2
```

Figure 32: More possible errors

Remark: The code at line 101 of `paging-linear-translate.py` won't ever be executed, because two error checks prior to the program reaching this point would have checked effectively the same thing. By this point, the parameters are all non-zero, and the address space size is a multiple of the page size. Also, the address space has been checked to have a size equal to a power of 2. Factors of such a number would just be lower powers of 2, which means that when the code reached line 101, page size is already established to be a power of 2.