CS 314 – Operating Systems Lab
Lab–7 Report

Student Name: Kavali Sri Vyshnavi Devi **Roll-No:** 200010023

1 Relocation

1.1

In this section, we need to generate the virtual address space with different values for seed and need to answer the questions generated.

- **In Bound -** VA < Limit and PA = Base + VA
- **Out of Bound -** VA > Limit

1.1.1 seed=1

In this subsection, we need to generate the virtual address space with seed=1 and the generated output is as follows which is being shown in the below image.

Command - python2.7 ./relocation.py -s 1

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

Base-and-Bounds register information:

  Base   : 0x0000363c (decimal 13884)
  Limit  : 290

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

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

Figure 1: Seed=1

In this figure the virtual address space is starting from 0x00000363c (decimal 13884) and have a size limit of 290 (in decimal). So, here the physical address space starts at 13884 and ends at 14174.

1. 0x0000030e (decimal: 782) is out of bounds as the size of space allotted to the process is only 290 and here 782 is greater than 290 which is allotted size and so it is **segmentation violation**.

2. 0x00000105 (decimal: 261) is not out of bounds as the size of space allotted to the process is 290 and here 261 is lesser than 290 which is allotted size and so the address in physical address space is **261+13884 = 14145 (0x00003741)**.

3. 0x000001fb (decimal: 507) is out of bounds as the size of space allotted to the process is only 290 and here 507 is greater than 290 which is allotted size and so it is **segmentation violation**.

4. 0x000001cc (decimal: 460) is out of bounds as the size of space allotted to the process is only 290 and here 460 is greater than 290 which is allotted size and so it is **segmentation violation**.

5. 0x0000029b (decimal: 667) is out of bounds as the size of space allotted to the process is only 290 and here 667 is greater than 290 which is allotted size and so it is **segmentation violation**.

1.1.2 seed=2

In this subsection, we need to generate the virtual address space with seed=2 and the generated output is as follows which is being shown in the below image.

Command - python2.7 ./relocation.py -s 2

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

Base-and-Bounds register information:

  Base   : 0x00003ca9 (decimal 15529)
  Limit  : 500

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

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

Figure 2: Seed=2

In this figure the virtual address space is starting from 0x00003ca9 (decimal 15529) and have a size limit of 500 (in decimal). So, here the physical address space starts at 15529 and ends at 16029.

1. 0x00000039 (decimal: 57) is not out of bounds as the size of space allotted to the process is 57 and here 57 is greater than 500 which is allotted size and so the address in physical address space is **57+15529 = 15586 (0x00003ce2)**.

2. 0x00000056 (decimal: 86) is not out of bounds as the size of space allotted to the process is 500 and here 86 is lesser than 500 which is allotted size and so the address in physical address space is **86+15529 = 15615 (0x00003cff)**.

3. 0x00000357 (decimal: 855) is out of bounds as the size of space allotted to the process is only 500 and here 855 is greater than 500 which is allotted size and so it is **segmentation violation**.

4. 0x000002f1 (decimal: 753) is out of bounds as the size of space allotted to the process is only 500 and here 753 is greater than 500 which is allotted size and so it is **segmentation violation**.

5. 0x000002ad (decimal: 685) is out of bounds as the size of space allotted to the process is only 500 and here 685 is greater than 500 which is allotted size and so it is **segmentation violation**.

1.1.3 seed=3

In this subsection, we need to generate the virtual address space with seed=3 and the generated output is as follows which is being shown in the below image.

Command - python2.7 ./relocation.py -s 3

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

Base-and-Bounds register information:

  Base   : 0x000022d4 (decimal 8916)
  Limit  : 316

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

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

Figure 3: Seed=3

In this figure the virtual address space is starting from 0x000022d4 (decimal 8916) and have a size limit of 316 (in decimal). So, here the physical address space starts at 8916 and ends at 9232.

1. 0x0000017a (decimal: 378) is out of bounds as the size of space allotted to the process is 316 and here 378 is greater than 316 which is allotted size and so it is **segmentation violation**.

2. 0x0000026a (decimal: 618) is not out of bounds as the size of space allotted to the process is 316 and here 618 is greater than 316 which is allotted size and so it is **segmentation violation**.

3. 0x00000280 (decimal: 640) is out of bounds as the size of space allotted to the process is only 316 and here 640 is greater than 316 which is allotted size and so it is **segmentation violation**.

4. 0x00000043 (decimal: 67) is not out of bounds as the size of space allotted to the process is only 316 and here 67 is lesser than 316 which is allotted size and so the address in physical address space is **67+8916=8983 (0x00002317)**.

5. 0x0000000d (decimal: 13) is not out of bounds as the size of space allotted to the process is only 316 and here 13 is lesser than 316 which is allotted size and so the address in physical address space is **13+8916=8929 (0x000022e1)**.

1.2

In this section we need to set seed value to 0 and number of virtual addresses to be generated in trace = 10 and the output is as shown in the below image.

Command - python2.7 ./relocation.py -s 0 -n 10 -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)
```

Figure 4: 1.2

Here, limit value should be greater than 929 (as minimum and less than 1024). Lower limit is **930** as the maximum memory address being accessed here is 929, there is only 1k (1024) virtual address size.

1.3

In this section we need to set seed value to 1, number of virtual addresses to be generated in trace = 10 and limit value = 100 the output is as shown in the below image.

Command - python2.7 ./relocation.py -s 1 -n 10 -l 100

Here, limit value is 100 and physical memory size = 16k that is equal to 16384. So, maximum value of base register needed to fit virtual address space in physical memory is 16384-100 that is equal to 16284.

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

Base-and-Bounds register information:

  Base   : 0x0000360b (decimal 13835)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x00000308 (decimal:  776) --> PA or segmentation violation?
  VA  1: 0x000001ae (decimal:  430) --> PA or segmentation violation?
  VA  2: 0x00000109 (decimal:  265) --> PA or segmentation violation?
  VA  3: 0x0000020b (decimal:  523) --> PA or segmentation violation?
  VA  4: 0x0000019e (decimal:  414) --> PA or segmentation violation?
  VA  5: 0x00000322 (decimal:  802) --> PA or segmentation violation?
  VA  6: 0x00000136 (decimal:  310) --> PA or segmentation violation?
  VA  7: 0x000001e8 (decimal:  488) --> PA or segmentation violation?
  VA  8: 0x00000255 (decimal:  597) --> PA or segmentation violation?
  VA  9: 0x000003a1 (decimal:  929) --> 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.
```

Figure 5: 1.3

1.4

Here, we need to generate virtual address trace with larger spaces (-a) and physical memories (-p).

Here, the values a=1g, p=16g, limit = 100 and base pointer is being set to the value = 16*1024*1024*1024 - 100 = 17179869034 (0x3ffffff6a).

Command - python2.7 ./relocation.py -s 1 -n 10 -l 100 -b 17179869034 -a 1g -p 16g

```
ARG seed 1
ARG address space size 1g
ARG phys mem size 16g

Base-and-Bounds register information:

  Base   : 0x3ffffff6a (decimal 17179869034)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x08996c7c (decimal: 144272508) --> PA or segmentation violation?
  VA  1: 0x363c5ab6 (decimal: 909925046) --> PA or segmentation violation?
  VA  2: 0x30e1aef0 (decimal: 820096752) --> PA or segmentation violation?
  VA  3: 0x10530d08 (decimal: 273878280) --> PA or segmentation violation?
  VA  4: 0x1fb5355e (decimal: 531969374) --> PA or segmentation violation?
  VA  5: 0x1cc4762b (decimal: 482637355) --> PA or segmentation violation?
  VA  6: 0x29b3b303 (decimal: 699642627) --> PA or segmentation violation?
  VA  7: 0x327a7181 (decimal: 846885249) --> PA or segmentation violation?
  VA  8: 0x0601cba3 (decimal: 100780963) --> PA or segmentation violation?
  VA  9: 0x01d071ef (decimal: 30437871) --> 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.
```

Figure 6: 16g

From, the virtual address traces generated one can say that all of them are not valid they are violating segmentation. In each of them the limit on the bound is 100 and the values generated in the virtual address trace are greater than the limit value that is 100.

Here, the values a=1g, p=32g, limit = 100 and base pointer is being set to the value = 32*1024*1024*1024 - 100 = 34359738368 (0x7fffff9c).

Command - python2.7 ./relocation.py -s 1 -n 10 -l 100 -b 34359738268 -a 1g -p 32g

```
ARG seed 1
ARG address space size 1g
ARG phys mem size 32g

Base-and-Bounds register information:

  Base   : 0x7ffffff9c (decimal 34359738268)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x08996c7c (decimal: 144272508) --> PA or segmentation violation?
  VA  1: 0x363c5ab6 (decimal: 909925046) --> PA or segmentation violation?
  VA  2: 0x30e1aef0 (decimal: 820096752) --> PA or segmentation violation?
  VA  3: 0x10530d08 (decimal: 273878280) --> PA or segmentation violation?
  VA  4: 0x1fb5355e (decimal: 531969374) --> PA or segmentation violation?
  VA  5: 0x1cc4762b (decimal: 482637355) --> PA or segmentation violation?
  VA  6: 0x29b3b303 (decimal: 699642627) --> PA or segmentation violation?
  VA  7: 0x327a7181 (decimal: 846885249) --> PA or segmentation violation?
  VA  8: 0x0601cba3 (decimal: 100780963) --> PA or segmentation violation?
  VA  9: 0x01d071ef (decimal: 30437871) --> 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.
```

Figure 7: 32g

From, the virtual address traces generated one can say that all of them are not valid they are violating segmentation. In each of them the limit on the bound is 100 and the values generated in the virtual address trace are greater than the limit value that is 100.

1.5

This graph is between valid fraction that means the number of virtual address traces in which there is no segmentation violation and the bound/limit value. From the graph one can say that when bound value increases that is the size of space allotted to a process increases valid fraction also increase as there will be high chances of memory being accessed lie in the size allotted to the process.

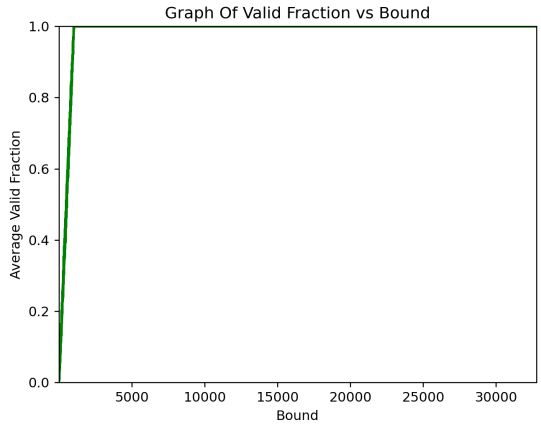

Figure 8: graph of valid fraction vs bound

2 Segmentation

2.1

In this subsection we are going to generate virtual address space of size 128, physical address space of size 512 and with different seed values.

- segment 0 (grows positive) - 0x00000000 (decimal 0), limit - 20 (range:0-19).
- segment 1 (grows negative) - 0x00000200 (decimal 512), limit - 20 (range:108-128).
- if vaddr < asize/2 – segment 0
  - In bound : vaddr < limit0 and paddr = vaddr + base0
  - Out of bound : vaddr >= limit0
- if vaddr >= asize/2 – segment 1
  - In bound : asize - limit1 <= vaddr and paddr = base1 + (vaddr - asize)
  - Out of bound : vaddr > asize - limit1

2.1.1 seed=0

In this subsection we are going to generate virtual address space of size 128, physical address space of size 512 and with seed value being set to 0.

Command - python2.7 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0

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

Segment register information:

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

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

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

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple address space with two segments: the top
bit of the virtual address can thus be used to check whether the virtual address
is in segment 0 (topbit=0) or segment 1 (topbit=1). Note that the base/limit pairs
given to you grow in different directions, depending on the segment, i.e., segment 0
grows in the positive direction, whereas segment 1 in the negative.
```

Figure 9: seed=0

Here, the base pointer 0 is point to address 0x00000000 (decimal 0) and base pointer 1 is pointing to 0x00000200 (decimal 512). Limit on first segment is 20 and limit on second segment is also 20.

1. Here, the given virtual address 0x0000006c (decimal: 108) is valid as it is under the range of **segment-1** that is from 108 to 128. So, the memory address being accessed is part of segment-1 and there is no segmentation violation. The translation of given virtual memory address to physical address space is equal to **512-128+108=492**.

2. Here, the given virtual address 0x00000061 (decimal: 97) is not valid as it does not lie under any range of **segment-0** that is from 0 to 19 and also not in the range of segment-1 that is from 108 to 128. So, the memory address being accessed is **violating segmentation** but lies in segment 1 as 97>64(asize/2).

3. Here, the given virtual address 0x00000035 (decimal: 53) is not valid as it does not lie under any range of **segment-0** that is from 0 to 19 and also not in the range of segment-1 that is from 108 to 128. So, the memory address being accessed is **violating segmentation** but lies in segment 0 as 53<64(asize/2).

4. Here, the given virtual address 0x00000021 (decimal: 33) is not valid as it does not lie under any range of **segment-0** that is from 0 to 19 and also not in the range of segment-1 that is from 108 to 128. So, the memory address being accessed is **violating segmentation** but lies in segment 0 as 33<64(asize/2).

5. Here, the given virtual address 0x00000041 (decimal: 65) is not valid as it does not lie under any range of **segment-0** that is from 0 to 19 and also not in the range of segment-1 that is from 108 to 128. So, the memory address being accessed is **violating segmentation** but lies in segment 1 as 65>=64(asize/2).

2.1.2 seed=1

In this subsection we are going to generate virtual address space of size 128, physical address space of size 512 and with seed value being set to 1.

Command - python2.7 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1

```
ARG seed 1
ARG address space size 128
ARG phys mem size 512

Segment register information:

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

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

Virtual Address Trace
  VA  0: 0x00000011 (decimal:   17) --> PA or segmentation violation?
  VA  1: 0x0000006c (decimal:  108) --> PA or segmentation violation?
  VA  2: 0x00000061 (decimal:   97) --> PA or segmentation violation?
  VA  3: 0x00000020 (decimal:   32) --> PA or segmentation violation?
  VA  4: 0x0000003f (decimal:   63) --> PA or segmentation violation?

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple address space with two segments: the top
bit of the virtual address can thus be used to check whether the virtual address
is in segment 0 (topbit=0) or segment 1 (topbit=1). Note that the base/limit pairs
given to you grow in different directions, depending on the segment, i.e., segment 0
grows in the positive direction, whereas segment 1 in the negative.
```

Figure 10: seed=1

1. Here, the given virtual address 0x00000011 (decimal: 17) is valid as it is under the range of **segment-0** that is from 0 to 19. So, the memory address being accessed is part of segment-0 and there is no segmentation violation. The translation of given virtual memory address to physical address space is equal to **0+17=17**.

2. Here, the given virtual address 0x0000006c (decimal: 108) is valid as it is under the range of **segment-1** that is from 108 to 128. So, the memory address being accessed is part of segment-1 and there is no segmentation violation. The translation of given virtual memory address to physical address space is equal to **512-128+108=492**.

3. Here, the given virtual address 0x00000061 (decimal: 97) is not valid as it does not lie under any range of segment-0 that is from 0 to 19 and also not in the range of segment-1 that is from 108 to 128. So, the memory address being accessed is **violating segmentation** but lies in segment 1 as 97>64 (asize/2).

4. Here, the given virtual address 0x00000020 (decimal: 32) is not valid as it does not lie under any range of segment-0 that is from 0 to 19 and also not in the range of segment-1 that is from 108 to 128. So, the memory address being accessed is **violating segmentation** but lies in segment 0 as 32<=64 (asize/2).

5. Here, the given virtual address 0x0000003f (decimal: 63) is not valid as it does not lie under any range of segment-0 that is from 0 to 19 and also not in the range of segment-1 that is from 108 to 128. So, the memory address being accessed is **violating segmentation** but lies in segment 0 as 63<=64 (asize/2).

2.1.3 seed=2

In this subsection we are going to generate virtual address space of size 128, physical address space of size 512 and with seed value being set to 2.

Command - python2.7 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2

1. Here, the given virtual address 0x0000007a (decimal: 122) is valid as it is under the range of **segment-1** that is from 108 to 128. So, the memory address being accessed is part of segment-1 and there is no segmentation violation. The translation of given virtual memory address to physical address space is equal to **512-128+122=506**.

2. Here, the given virtual address 0x00000079 (decimal: 121) is valid as it is under the range of **segment-1** that is from 108 to 128. So, the memory address being accessed is part of segment-1 and there is no segmentation violation. The translation of given virtual memory address to physical address space is equal to **512-128+121=505**.

3. Here, the given virtual address 0x00000007 (decimal: 7) is valid as it is under the range of **segment-0** that is from 0 to 19. So, the memory address being accessed is part of segment-0 and there is no segmentation violation. The translation of given virtual memory address to physical address space is equal to **0+7=7**.

4. Here, the given virtual address 0x0000000a (decimal: 10) is valid as it is under the range of **segment-0** that is from 0 to 19. So, the memory address being accessed is part of segment-0 and there is no segmentation violation. The translation of given virtual memory address to physical address space is equal to **0+10=10**.

5. Here, the given virtual address0x0000006a (decimal: 106) is not valid as it does not lie under any range of segment-0 that is from 0 to 19 and also not in the range of segment-1 that is from 108 to 128. So, the memory address being accessed is **violating segmentation** but lies in segment 1 as 106>=64.

```
ARG seed 2
ARG address space size 128
ARG phys mem size 512

Segment register information:

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

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

Virtual Address Trace
  VA  0: 0x0000007a (decimal:  122) --> PA or segmentation violation?
  VA  1: 0x00000079 (decimal:  121) --> PA or segmentation violation?
  VA  2: 0x00000007 (decimal:    7) --> PA or segmentation violation?
  VA  3: 0x0000000a (decimal:   10) --> PA or segmentation violation?
  VA  4: 0x0000006a (decimal:  106) --> PA or segmentation violation?

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple address space with two segments: the top
bit of the virtual address can thus be used to check whether the virtual address
is in segment 0 (topbit=0) or segment 1 (topbit=1). Note that the base/limit pairs
given to you grow in different directions, depending on the segment, i.e., segment 0
grows in the positive direction, whereas segment 1 in the negative.
```

Figure 11: seed=2

2.2

- Highest legal virtual address space in segment 0: 19.
- Lowest legal virtual address space in segment 1: 108 (128-20).
- Lowest illegal address in entire physical address space: 20 (0+20).
- Highest illegal address in entire physical address space: 491 (512-20-1).
- Lowest illegal address in entire address space (virtual): 20 (0+20).
- Highest illegal address in entire address space (virtual): 107 (128-20-1).
- Segment 0 physical addresses: 0-19
- Segment 1 physical addresses: 492-512

To verify whether these values are correct or not the below command is executed which include -c flag along with -A flag.

Command - python2.7 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 12: verify

From the above figure we can verify that the values are correct as the those values match with the exact result.

2.3

to get output as like valid, valid, violation, ..., violation, valid, valid we need to get valid for memory addresses 0,1 and 14,15 everything else should give invalid/violation as the output. To produce such a virtual address space we need the base pointer of segment 0 to 0 and limit of size on segment 0 equal to 2, base pointer for segment 1 as 16 and limit of size on segment1 is 2.

Command - python2.7 segmentation.py -a 16 -p 128 -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 –b0 0 –l0 2 –b1 16 –l1 2 -c

```
ARG seed 0
ARG address space size 16
ARG phys mem size 128

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 2

  Segment 1 base  (grows negative) : 0x00000010 (decimal 16)
  Segment 1 limit                  : 2

Virtual Address Trace
  VA  0: 0x00000000 (decimal:    0) --> VALID in SEG0: 0x00000000 (decimal:    0)
  VA  1: 0x00000001 (decimal:    1) --> VALID in SEG0: 0x00000001 (decimal:    1)
  VA  2: 0x00000002 (decimal:    2) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x00000003 (decimal:    3) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x00000004 (decimal:    4) --> SEGMENTATION VIOLATION (SEG0)
  VA  5: 0x00000005 (decimal:    5) --> SEGMENTATION VIOLATION (SEG0)
  VA  6: 0x00000006 (decimal:    6) --> SEGMENTATION VIOLATION (SEG0)
  VA  7: 0x00000007 (decimal:    7) --> SEGMENTATION VIOLATION (SEG0)
  VA  8: 0x00000008 (decimal:    8) --> SEGMENTATION VIOLATION (SEG1)
  VA  9: 0x00000009 (decimal:    9) --> SEGMENTATION VIOLATION (SEG1)
  VA 10: 0x0000000a (decimal:   10) --> SEGMENTATION VIOLATION (SEG1)
  VA 11: 0x0000000b (decimal:   11) --> SEGMENTATION VIOLATION (SEG1)
  VA 12: 0x0000000c (decimal:   12) --> SEGMENTATION VIOLATION (SEG1)
  VA 13: 0x0000000d (decimal:   13) --> SEGMENTATION VIOLATION (SEG1)
  VA 14: 0x0000000e (decimal:   14) --> VALID in SEG1: 0x0000000e (decimal:   14)
  VA 15: 0x0000000f (decimal:   15) --> VALID in SEG1: 0x0000000f (decimal:   15)
```

2.4

To get 90% of the randomly-generated virtual addresses are valid (not segmentation violations) we need to configure limit of a segment to be almost equal to 0.9*(size of virtual address space being generated. Parameters which are responsible to obtain this are -l, -L and -a.
For Example, let us consider a small virtual address space having a size of 32 bytes and 90% of it means 28.8 (approximately 28 bytes) and size of physical address space be equal to 256 bytes. Let us consider the limits of the segments as follows.

- limit 0 - 14
- limit 1 - 15

Command - python2.7 segmentation.py –l0 14 –l1 15 -c -a 32 -p 256 -s 0
Command - python2.7 segmentation.py –l0 14 –l1 15 -c -a 32 -p 256 -s 2

```
ARG seed 0
ARG address space size 32
ARG phys mem size 256

Segment register information:

  Segment 0 base  (grows positive) : 0x000000d8 (decimal 216)
  Segment 0 limit                  : 14

  Segment 1 base  (grows negative) : 0x000000d1 (decimal 209)
  Segment 1 limit                  : 15

Virtual Address Trace
  VA  0: 0x0000000d (decimal:   13) --> VALID in SEG0: 0x000000e5 (decimal:  229)
  VA  1: 0x00000008 (decimal:    8) --> VALID in SEG0: 0x000000e0 (decimal:  224)
  VA  2: 0x00000010 (decimal:   16) --> SEGMENTATION VIOLATION (SEG1)
  VA  3: 0x0000000c (decimal:   12) --> VALID in SEG0: 0x000000e4 (decimal:  228)
  VA  4: 0x00000019 (decimal:   25) --> VALID in SEG1: 0x000000ca (decimal:  202)
```

```
ARG seed 2
ARG address space size 32
ARG phys mem size 256

Segment register information:

  Segment 0 base  (grows positive) : 0x0000000e (decimal 14)
  Segment 0 limit                  : 14

  Segment 1 base  (grows negative) : 0x000000e4 (decimal 228)
  Segment 1 limit                  : 15

Virtual Address Trace
  VA  0: 0x00000017 (decimal:   23) --> VALID in SEG1: 0x000000db (decimal:  219)
  VA  1: 0x00000015 (decimal:   21) --> VALID in SEG1: 0x000000d9 (decimal:  217)
  VA  2: 0x00000009 (decimal:    9) --> VALID in SEG0: 0x00000017 (decimal:   23)
  VA  3: 0x00000013 (decimal:   19) --> VALID in SEG1: 0x000000d7 (decimal:  215)
  VA  4: 0x00000013 (decimal:   19) --> VALID in SEG1: 0x000000d7 (decimal:  215)
```

After executing on seed values 0 and 2 one can notice that the out of 10, 9 of them are not violating segmentation which is 90%.

2.5

To run simulator such than no virtual address is valid is to make limit of segment equal to 0 that means we need to set the flags -l and -L to 0.
One of the possible combination of values for base and bound registers are as follows.

- base 0 - 0

- base 1 - 10
- limit 0 - 0
- limit 1 - 0

Command - python2.7 segmentation.py –l0 0 –l1 0 –b0 0 –b1 10

From the below figure one can notice that all the virtual address traces generated are violating the segmentation.

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

Segment register information:

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

  Segment 1 base  (grows negative) : 0x0000000a (decimal 10)
  Segment 1 limit                  : 0

Virtual Address Trace
  VA  0: 0x00000360 (decimal:  864) --> SEGMENTATION VIOLATION (SEG1)
  VA  1: 0x00000308 (decimal:  776) --> SEGMENTATION VIOLATION (SEG1)
  VA  2: 0x000001ae (decimal:  430) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x00000109 (decimal:  265) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x0000020b (decimal:  523) --> SEGMENTATION VIOLATION (SEG1)
```

Figure 13: u=0

3

3.1 Different Number Of Bits

From the below we can observe if number of bits in the address space increase the page table size also increases.

3.1.1 Number Of Bits=16

To set number of bits to 16 for which we need to use the flag -v.

Command - python2.7 paging-linear-size.py -v 16

Number of bits in virtual address = 16
Page size = 4k = 4096 (need 12 bits)
page Table Entry size = 4
Offset = 12 bits
Number of VPN bits = total bits - Offset bits = 16-12=4
Size of page table = $2^{(numberofvpnbits)} * (pagetableentrysize) = 2^4 * 4 = 16*4 = 64$ bytes.

3.1.2 Number Of Bits=32

To set number of bits to 32 for which we need to use the flag -v.

Command - python2.7 paging-linear-size.py -v 32

Number of bits in virtual address = 32
Page size = 4k = 4096 (need 12 bits)
page Table Entry size = 4
Offset = 12 bits
Number of VPN bits = total bits - Offset bits = 32-12=20
Size of page table = $2^{(numberofvpnbits)} * (pagetableentrysize) = 2^{20} * 4$ = 1024*4 KB = 4096 KB.

3.2 Different Page Size

From the below we can observe if page size increase the page table size decreases as the number of offset bits increase and number of VPN bits decrease and leads to lesser size of the page table.

3.2.1 Page Size = 16

To set page size to 16 for which we need to use the flag -p.

Command - python2.7 paging-linear-size.py -p 16

Number of bits in virtual address = 32
Page size = 16 (need 4 bits)
page Table Entry size = 4
Offset = 4 bits
Number of VPN bits = total bits - Offset bits = 32-4=28
Size of page table = $2^{(numberofvpnbits)} * (pagetableentrysize) = 2^28 * 4 = 1048576$ KB.

3.2.2 Page Size = 32

To set page size to 32 for which we need to use the flag -v.

Command - python2.7 paging-linear-size.py -p 32

Number of bits in virtual address = 32
Page size = 32 (need 5 bits)
page Table Entry size = 4
Offset = 5 bits
Number of VPN bits = total bits - Offset bits = 32-5=27
Size of page table = $2^{(numberofvpnbits)} * (pagetableentrysize) = 2^{27} * 4 = 524288$ KB.

3.3 Different Page Table Entry Size

From the below we can observe if page table entry size increases the page table size increases as it will be the product of some constant and page table entry size. Here constant means $2^{(numberofvpnbits)}$.

3.3.1 Page Table Entry Size = 16

To set page table entry size to 16 for which we need to use the flag -e.

Command - python2.7 paging-linear-size.py -e 16

Number of bits in virtual address = 32
Page size = 4k (need 12 bits)
page Table Entry size = 16
Offset = 12 bits
Number of VPN bits = total bits - Offset bits = 32-12=20
Size of page table = $2^{(numberofvpnbits)} * (pagetableentrysize) = 2^20 * 16 = 16384$ KB.

3.3.2 Page Table Entry Size = 32

To set number of bits to 32 for which we need to use the flag -e.

Command - python2.7 paging-linear-size.py -e 32

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

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

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

Figure 14: u=0

Number of bits in virtual address = 32
Page size = 4k (need 12 bits)
page Table Entry size = 32
Offset = 12 bits
Number of VPN bits = total bits - Offset bits = 32-12=20
Size of page table = $2^{(numberofvpnbits)} * (pagetableentrysize) = 2^20 * 32 = 32768$ KB.

4

Default values if the flag is not are as mentioned below (this information is present in the readme file associated with the code given).

bits in virtual address - 32
page size - 4k
page table entry size - 4

4.1

Here, page table size will be equal to the ratio of address space to page size multiplied by page table entry size.

Size of page table = (size of each page table entry) × (Number of entries)
Number of entries in the table = address space size / page size

As the address space grows page table size increases and if page size increases page table size will be decreased. Using big pages in reality causes a lot of memory wastage which can be utilized for other processes if used less page size.

commands to understand how linear page table size changes as the address space grows

- python2.7 ./paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
- python2.7 ./paging-linear-translate.py -P 1k -a 2m -p 512m -v -n 0
- python2.7 ./paging-linear-translate.py -P 1k -a 4m -p 512m -v -n 0

After executing the above commands one can say that as we double the address space size the page table size is also becoming double which means that the page table size increases proportional to the address space size.

commands to understand how linear page table size changes as page size grows

- python2.7 ./paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
- python2.7 ./paging-linear-translate.py -P 2k -a 1m -p 512m -v -n 0
- python2.7 ./paging-linear-translate.py -P 4k -a 1m -p 512m -v -n 0

After executing the above commands one can say that as we double the page size the page table size is also becoming half which means that page table size decreases proportional to page size.

4.2

From all the observations below one can say that as the value that is set to the u flag increase the number of valid pages also increase.

- address space size - 16k (2^{14})
- physical address space size - 32k
- page size - 1k (2^{10})
- Number of bit in total - 14, Number of offset bits - 10, Number of vpn bits - 4

4.2.1 u=0

Let, the percentage of address space used to be 0 which can be set using the flag -u.

Command - python2.7 ./paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 0

Here, the output after executing the above command is as shown in the below figure.

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

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

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

Figure 15: u=0

- In the above figure 1st virtual address is 0x00003a39 (decimal: 14905) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 1110 that is 14 and VPN 14 is not valid as in the page table physical address corresponding to VPN 14 is 0x00000000 whose left most bit is not set.
- In the above figure 2nd virtual address is 0x00003ee5 (decimal: 16101) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 1111 that is 15 and VPN 15 is not valid as in the page table physical address corresponding to VPN 15 is 0x00000000 whose left most bit is not set.

- In the above figure 3rd virtual address is 0x000033da (decimal: 13274) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 1100 that is 12 and VPN 12 is not valid as in the page table physical address corresponding to VPN 12 is 0x00000000 whose left most bit is not set.
- In the above figure 4th virtual address is 0x000039bd (decimal: 14781) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 1100 that is 14 and VPN 14 is not valid as in the page table physical address corresponding to VPN 14 is 0x00000000 whose left most bit is not set.
- In the above figure 5th virtual address is 0x000013d9 (decimal: 5081) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 0100 that is 4 and VPN 4 is not valid as in the page table physical address corresponding to VPN 44 is 0x00000000 whose left most bit is not set.

4.2.2 u=25

Let, the percentage of address space used be 25 which can be set using the flag -u.

Command - python2.7 ./paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 25

Here, the output after executing the above command is as shown in the below figure.

```
Page Table (from entry 0 down to the max size)
  [       0]   0x80000018
  [       1]   0x00000000
  [       2]   0x00000000
  [       3]   0x00000000
  [       4]   0x00000000
  [       5]   0x80000009
  [       6]   0x00000000
  [       7]   0x00000000
  [       8]   0x80000010
  [       9]   0x00000000
  [      10]   0x80000013
  [      11]   0x00000000
  [      12]   0x8000001f
  [      13]   0x8000001c
  [      14]   0x00000000
  [      15]   0x00000000

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

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

Figure 16: u=25

- In the above figure 1st virtual address is 0x00003986 (decimal: 14726) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 1110 that is 14 and VPN 14 is not valid as in the page table physical address corresponding to VPN 14 is 0x00000000 whose left most bit is not set.
- In the above figure 2nd virtual address is 0x00002bc6 (decimal: 11206) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 1010 that is 10 and VPN 10 is valid as in the page table physical address corresponding to VPN 10 is 0x80000013 whose left most bit is equal to 1.

- In the above figure 3rd virtual address is 0x00001e37 (decimal: 7735) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 0111 that is 7 and VPN 7 is not valid as in the page table physical address corresponding to VPN 7 is 0x00000000 whose left most bit is not set.
- In the above figure 4th virtual address is 0x00000671 (decimal: 1649) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 0001 that is 1 and VPN 1 is not valid as in the page table physical address corresponding to VPN 1 is 0x00000000 whose left most bit is not set.
- In the above figure 5th virtual address is 0x00001bc9 (decimal: 7113) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 0110 that is 6 and VPN 6 is not valid as in the page table physical address corresponding to VPN 6 is 0x00000000 whose left most bit is not set.

4.2.3 u=50

Let, the percentage of address space used be 50 which can be set using the flag -u.

Command - python2.7 ./paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 50

Here, the output after executing the above command is as shown in the below figure.

```
Page Table (from entry 0 down to the max size)
  [       0]   0x80000018
  [       1]   0x00000000
  [       2]   0x00000000
  [       3]   0x8000000c
  [       4]   0x80000009
  [       5]   0x00000000
  [       6]   0x8000001d
  [       7]   0x80000013
  [       8]   0x00000000
  [       9]   0x8000001f
  [      10]   0x8000001c
  [      11]   0x00000000
  [      12]   0x8000000f
  [      13]   0x00000000
  [      14]   0x00000000
  [      15]   0x80000008

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

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

Figure 17: u=50

- In the above figure 1st virtual address is 0x00003385 (decimal: 13189) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 1100 that is 12 and VPN 12 is valid as in the page table physical address corresponding to VPN 12 is 0x8000000f whose left most bit is equal to 1.
- In the above figure 2nd virtual address is 0x0000231d (decimal: 8989) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 1000 that is 8 and VPN 8 is not valid as in the page table physical address corresponding to VPN 8 is 0x00000000 whose left most bit is not set.

- In the above figure 3rd virtual address is 0x000000e6 (decimal: 230) and it is known that page size = 1kb (1024 bytes) here, 230<1024 so it belongs to frame 0 which is valid as in the page table physical address corresponding to VPN 0 is 0x80000018 whose left most bit is equal to 1.
- In the above figure 4th virtual address is 0x00002e0f (decimal: 11791) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 1011 that is 11 and VPN 11 is not valid as in the page table physical address corresponding to VPN 11 is 0x00000000 whose left most bit is not set.
- In the above figure 5th virtual address is 0x00001986 (decimal: 6534) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 0110 that is 6 and VPN 6 is valid as in the page table physical address corresponding to VPN 6 is 0x8000001d whose left most bit is equal to 1.

4.2.4 u=75

Let, the percentage of address space used be 75 which can be set using the flag -u.

Command - python2.7 ./paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 75

Here, the output after executing the above command is as shown in the below figure.

```
Page Table (from entry 0 down to the max size)
  [       0]   0x80000018
  [       1]   0x80000008
  [       2]   0x8000000c
  [       3]   0x80000009
  [       4]   0x80000012
  [       5]   0x80000010
  [       6]   0x8000001f
  [       7]   0x8000001c
  [       8]   0x80000017
  [       9]   0x80000015
  [      10]   0x80000003
  [      11]   0x80000013
  [      12]   0x8000001e
  [      13]   0x8000001b
  [      14]   0x80000019
  [      15]   0x80000000

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

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

Figure 18: u=75

- In the above figure 1st virtual address is 0x00002e0f (decimal: 11791) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 1011 that is 11 and VPN 11 is valid as in the page table physical address corresponding to VPN 11 is 0x80000013 whose left most bit is equal to 1.
- In the above figure 2nd virtual address is 0x00001986 (decimal: 6534) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 0110 that is 6 and VPN 6 is valid as in the page table physical address corresponding to VPN 6 is 0x8000001f whose left most bit is equal to 1.

- In the above figure 3rd virtual address is 0x000034ca (decimal: 13514) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 1101 that is 13 and VPN 13 is valid as in the page table physical address corresponding to VPN 13 is 0x8000001b whose left most bit is equal to 1.
- In the above figure 4th virtual address is 0x00002ac3 (decimal: 10947) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 1010 that is 10 and VPN 10 is valid as in the page table physical address corresponding to VPN 10 is 0x80000003 whose left most bit is equal to 1.
- In the above figure 5th virtual address is 0x00000012 (decimal: 18) and it is known that page size = 1kb (1024 bytes) here, 18<1024 so it belongs to frame 0 which is valid as in the page table physical address corresponding to VPN 0 is 0x80000018 whose left most bit is equal to 1.

4.2.5 u=100

Let, the percentage of address space used be 100 which can be set using the flag -u.

Command - python2.7 ./paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 100

Here, the output after executing the above command is as shown in the below figure.

```
Page Table (from entry 0 down to the max size)
  [       0]   0x80000018
  [       1]   0x80000008
  [       2]   0x8000000c
  [       3]   0x80000009
  [       4]   0x80000012
  [       5]   0x80000010
  [       6]   0x8000001f
  [       7]   0x8000001c
  [       8]   0x80000017
  [       9]   0x80000015
  [      10]   0x80000003
  [      11]   0x80000013
  [      12]   0x8000001e
  [      13]   0x8000001b
  [      14]   0x80000019
  [      15]   0x80000000

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

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

Figure 19: u=100

- In the above figure 1st virtual address is 0x00002e0f (decimal: 11791) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 1011 that is 11 and VPN 11 is valid as in the page table physical address corresponding to VPN 11 is 0x80000013 whose left most bit is equal to 1.
- In the above figure 2nd virtual address is 0x00001986 (decimal: 6534) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 0110 that is 63 and VPN 6 is valid as in the page table physical address corresponding to VPN 6 is 0x8000001f whose left most bit is equal to 1.

- In the above figure 3rd virtual address is 0x000034ca (decimal: 13514) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 1101 that is 13 and VPN 13 is valid as in the page table physical address corresponding to VPN 13 is 0x8000001b whose left most bit is equal to 1.
- In the above figure 4th virtual address is 0x00002ac3 (decimal: 10947) and it is known that page size = 1kb (1024 bytes) here, after removing the last 10 bits the VPN number is 1010 that is 10 and VPN 10 is valid as in the page table physical address corresponding to VPN 10 is 0x80000003 whose left most bit is equal to 1.
- In the above figure 5th virtual address is 0x00000012 (decimal: 18) and it is known that page size = 1kb (1024 bytes) here, 18<1024 so it belongs to frame 0 which is valid as in the page table physical address corresponding to VPN 0 is 0x80000018 whose left most bit is equal to 1.

4.3

Trying out some different random seeds, and some different address-space parameters.

After executing all the below commands one can conclude that the first one we tried generated small page table and also it experiments on smaller sizes of address spaces. In case of 3rd one we tried generating large page table and also it experiments on bigger sizes of address spaces. Page table entries of 1st and 2nd are equal to 32/8 that is 4 where as it is 256/1 that is 256 in case of 3rd one. First one is unrealistically small and in case of 3rd one it is not good to have large page size as it leads to internal fragmentation.

4.3.1 -P 8 -a 32 -p 1024 -v -s 1

Here, we are setting address space size to 32, page size to 8 and physical address to 1024 with seed value = 1.

Command - python2.7 ./paging-linear-translate.py -P 8 -a 32 -p 1024 -v -s 1

Here, the output after executing the above command is as shown in the below figure.

4.3.2 -P 8k -a 32k -p 1m -v -s 2

Here, we are setting address space size to 32k, page size to 8k and physical address to 1m with seed value = 2.

Command - python2.7 ./paging-linear-translate.py -P 8k -a 32k -p 1m -v -s 2

Here, the output after executing the above command is as shown in the below figure.

4.3.3 -P 1m -a 256m -p 512m -v -s 3

Here, we are setting address space size to 256m, page size to 1m and physical address to 512m with seed value = 3.

Command - python2.7 ./paging-linear-translate.py -P 1m -a 256m -p 512m -v -s 3

Here, the output after executing the above command is as shown in the below figure.

```
Page Table (from entry 0 down to the max size)
  [       0]   0x00000000
  [       1]   0x80000018
  [       2]   0x00000000
  [       3]   0x00000000
  [       4]   0x00000000
  [       5]   0x80000019
  [       6]   0x00000000
  [       7]   0x00000000
  [       8]   0x8000000d
  [       9]   0x80000000
  [      10]   0x00000000
  [      11]   0x80000007
  [      12]   0x8000001c
  [      13]   0x00000000
  [      14]   0x00000000
  [      15]   0x8000001e

Virtual Address Trace
  VA 0x00001865 (decimal:     6245) --> PA or invalid address?
  VA 0x00000ddc (decimal:     3548) --> PA or invalid address?
  VA 0x00001b03 (decimal:     6915) --> PA or invalid address?
  VA 0x000001db (decimal:      475) --> PA or invalid address?
  VA 0x00000e30 (decimal:     3632) --> PA or invalid address?

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

Figure 20: -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

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

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

Figure 21: -P 8k -a 32k -p 1m -v -s 2

```
ARG seed 3
ARG address space size 256m
ARG phys mem size 512m
ARG page size 1m
ARG verbose True
ARG addresses -1
```

Figure 22: -P 1m -a 256m -p 512m -v -s 3

4.4

If the program is executing without specifying any command line arguments the default values are seed 0, address space size 16k, phys memory size 64k and page size 4k. Using the default values we can try out some other problems.

- Let us check what happens if virtual address required is greater than physical memory (64k).

  Command - python2.7 ./paging-linear-translate.py -a 0

  From the above picture one can say that as we try to assign space which is greater than physical memory size to virtual address it thrown an error saying **physical memory size must be GREATER than address space size (for this simulation)**.

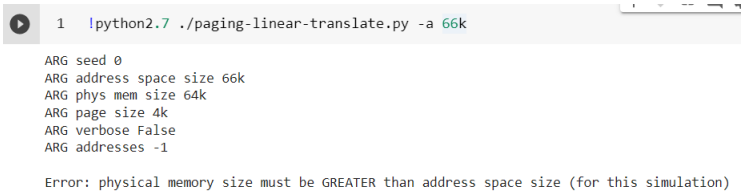

Figure 23: a=66k

- Let us check what happens if virtual address required is set to 0.

  Command - python2.7 ./paging-linear-translate.py -a 66k

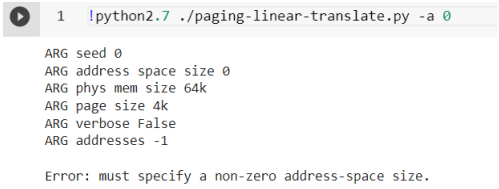

Figure 24: a=0

From the above picture one can say that as we try to assign 0 to virtual address it thrown an error saying **must specify a non-zero address-space size**.

- Let us check what happens if physical memory size is set to 0.

  Command - python2.7 ./paging-linear-translate.py -p 0

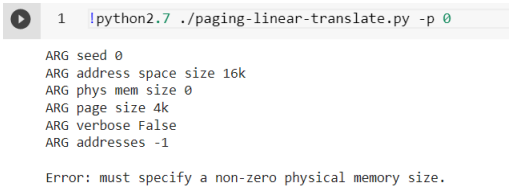

Figure 25: p=0

From the above picture one can say that as we try to assign 0 to physical memory size it thrown an error saying **must specify a non-zero physical memory size**.

- Let us check what happens if page size is set to 0.

  Command - python2.7 ./paging-linear-translate.py -P 0

  From the above picture one can say that as we try to assign 0 to page size it thrown an error saying **float division by zero**. If page size is 0 then virtual address can't be divided into pages and if tried then gives float division by zero error.

```
[40]  1  !python2.7 ./paging-linear-translate.py -P 0

ARG seed 0
ARG address space size 16k
ARG phys mem size 64k
ARG page size 0
ARG verbose False
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
```

Figure 26: P=0

- Let us check what happens if page size is set to a value which is greater than virtual address space size (16k by default).

  Command - python2.7 ./paging-linear-translate.py -P 32k -v -c

```
1  !python2.7 ./paging-linear-translate.py -P 32k -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


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)

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
```

Figure 27: P>p

From the above picture one can say that as we try to assign 32k to a page size it gives an error **index out of range** as we try to access something which is greater that 16k assuming it is present in the page of size 32k.

- Let us check what happens if page size is set to a value which is not power of 2.

  Command - python2.7 ./paging-linear-translate.py -P 18k -v -c

```
1  !python2.7 ./paging-linear-translate.py -P 18k -v -c

ARG seed 0
ARG address space size 16k
ARG phys mem size 64k
ARG page size 18
ARG verbose True
ARG addresses -1

Error in argument: page size must be a power of 2
```

Figure 28: P!=power of 2

From the above picture one can say that as we try to assign 18k to a page size it gives an error as page size should be in power of 2 which is not happening in this case. So, it throws an error saying **page size must be a power of 2**.