CS 314: Operating Systems Laboratory
Lab 6

Nampally Pranav
190010026

---

1 Question 1

1.1 1.1

```
$ python2 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) --> 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

$ python2 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) --> 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

$ python2 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) --> 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)
```

Explanation for VA3 of Seed 3:
Here as Base=8916 and Limit=316, so allowed VAs range from 0 to 315.
And for VA= 67 (means it is valid as it lies in above range), Physical Addr= Base + VA= 8916+67=8983.
Similar approach is extended to other cases too.

1.2 1.2

The value after -l should be >=930 as we that the VA addresses asked have the maximum decimal value as 929 (9th one), so the bound is here kept as 930. We would want the bound/limit to be such that Physical addr(=virtual Addr +base) should be in bound here the bound has to a minimum of 930. Thus limit is set as 930.

```
$ python2 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)
```

1.3 1.3

Whatever be the Base value it doesn't affect the the decision as we just need the limit value to decide whether its a page fault or not. So, here as 100¡ all the VA decimal values, thus these are all page faults. Even if we change the base it doesn't matter here incase of page faults. The max value of physical addr=16K=16*1024=16384 And, we need base=max.phy.addr -limit=16384-100=16284

```
$ python2 relocation.py -s 0 -n 10 -l 100 -b 16284

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

Base-and-Bounds register information:

  Base   : 0x00003f9c (decimal 16284)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x00000360 (decimal:  864) --> SEGMENTATION VIOLATION
  VA  1: 0x00000308 (decimal:  776) --> SEGMENTATION VIOLATION
  VA  2: 0x000001ae (decimal:  430) --> SEGMENTATION VIOLATION
  VA  3: 0x00000109 (decimal:  265) --> SEGMENTATION VIOLATION
  VA  4: 0x0000020b (decimal:  523) --> SEGMENTATION VIOLATION
  VA  5: 0x0000019e (decimal:  414) --> SEGMENTATION VIOLATION
  VA  6: 0x00000322 (decimal:  802) --> SEGMENTATION VIOLATION
  VA  7: 0x00000136 (decimal:  310) --> SEGMENTATION VIOLATION
  VA  8: 0x000001e8 (decimal:  488) --> SEGMENTATION VIOLATION
  VA  9: 0x00000255 (decimal:  597) --> SEGMENTATION VIOLATION
```

1.4 1.4

I have considered the Address space size as 128m (-a) and Physical Memory Size as 4g (-p), which are large as required. Then, the max value of physical addr= 4g =4*1024*1024*1024=4294967296 And, we need max base=max.phy.addr -100(limit)=4294967296-100=4294967196. But, as explained in Q1.3, this base value doesn't change the decision regarding whether a VA causes a segmentation fault or gives a Valid PA.

```
$ python2 relocation.py -s 1 -n 10 -l 100 -b 4294967196 -a 128m -p 4g
```

```
ARG seed 1
ARG address space size 128m
ARG phys mem size 4g

Base-and-Bounds register information:

  Base   : 0xffffff9c (decimal 4294967196)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x01132d8f (decimal: 18034063) --> SEGMENTATION VIOLATION
  VA  1: 0x06c78b56 (decimal: 113740630) --> SEGMENTATION VIOLATION
  VA  2: 0x061c35de (decimal: 102512094) --> SEGMENTATION VIOLATION
  VA  3: 0x020a61a1 (decimal: 34234785) --> SEGMENTATION VIOLATION
  VA  4: 0x03f6a6ab (decimal: 66496171) --> SEGMENTATION VIOLATION
  VA  5: 0x03988ec5 (decimal: 60329669) --> SEGMENTATION VIOLATION
  VA  6: 0x05367660 (decimal: 87455328) --> SEGMENTATION VIOLATION
  VA  7: 0x064f4e30 (decimal: 105860656) --> SEGMENTATION VIOLATION
  VA  8: 0x00c03974 (decimal: 12597620) --> SEGMENTATION VIOLATION
  VA  9: 0x003a0e3d (decimal: 3804733) --> SEGMENTATION VIOLATION
```

1.5 1.5

The below graph shows the relation between the fraction of valid VAs to limit value:

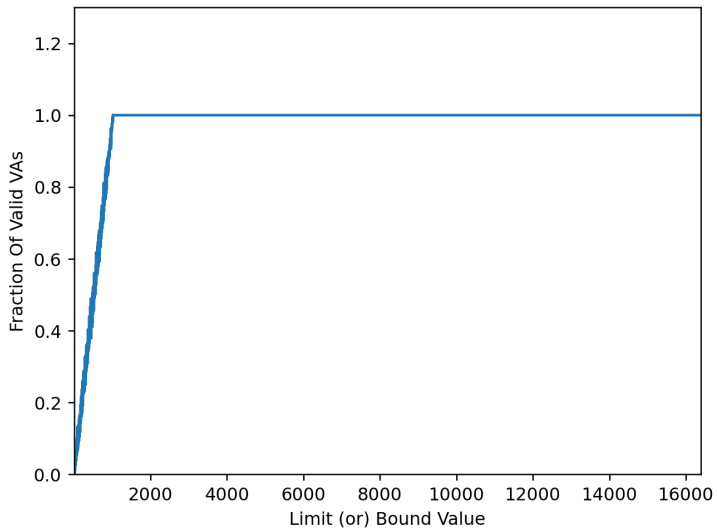

Figure 1: Graph showing the relation between the fraction of valid VAs to limit value

Explanation:
Given a Base Value as x, and a limit value l.
We can say that PA=VA+x.
This means that the valid VAs would be in the range 0 to l-1 and these correspondingly map in Physical Memory to PAs x to x+l-1.
We see that initially as limit value increases, then the fraction of Valid VAs also increases till limit value becomes large enough to cover all the VAs, then the the fraction will stagnate at 1 even though limit value increases.
This is because, initially the limit will be small and the available VAs can be large but only the VAs in range 0 to l-1 are valid, so as the l value increase so does the range of valid VAs increase. Thus, at some point the l value becomes large enough to encompass all the available VAs, then even if l increases the range of Valid VAs increases but all the available VAs are already covered so the saturation is reached.

2 Question 2

2.1 2.1

Here Address Space Size: 128, Physical Memory Size: 512, Value of Segment 0 Base Register: 0, Value of Segment 0 Limit Register: 20, Value of Segment 1 Base Register: 512, Value of Segment 1 Limit Register: 20

So for SEG0: Valid VAs are 0 to 19 for Phy.Addr: 0 to 19
For SEG1 valid VAs are 108 to 127 for Phys. Addr: 492 to 511
NOTE: In case of Segmentation Violation, consider 1st half [0 to 63] for SEG0 and remaining for SEG2[64 to 127]

```
For 1st case:
Seed No. = 0
VA  0: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492) as 492 + (108-108)
= 492
VA  1: 0x00000061 (decimal:   97) --> SEGMENTATION VIOLATION (SEG1)
VA  2: 0x00000035 (decimal:   53) --> SEGMENTATION VIOLATION (SEG0)
VA  3: 0x00000021 (decimal:   33) --> SEGMENTATION VIOLATION (SEG0)
VA  4: 0x00000041 (decimal:   65) --> SEGMENTATION VIOLATION (SEG1)


For 2nd Case
Seed No. = 1
VA  0: 0x00000011 (decimal:   17) --> VALID in SEG0: 0x00000011 (decimal:   17) as 0 + (17-0)= 17
VA  1: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492) as 492+(108-108)
= 492
VA  2: 0x00000061 (decimal:   97) --> SEGMENTATION VIOLATION (SEG1)
VA  3: 0x00000020 (decimal:   32) --> SEGMENTATION VIOLATION (SEG0)
VA  4: 0x0000003f (decimal:   63) --> SEGMENTATION VIOLATION (SEG0)

For 3rd Case:
Seed No. = 2
VA  0: 0x0000007a (decimal:  122) --> VALID in SEG1: 0x000001fa (decimal:  506) as 492+(122-108)
```

```
=492+14=506
VA  1: 0x00000079 (decimal:  121) --> VALID in SEG1: 0x000001f9 (decimal:  505) as 492+(121-108)
=492+13=505
VA  2: 0x00000007 (decimal:    7) --> VALID in SEG0: 0x00000007 (decimal:    7) as 0+ (7-0)=7
VA  3: 0x0000000a (decimal:   10) --> VALID in SEG0: 0x0000000a (decimal:   10) as 0+ (10-0)=10
VA  4: 0x0000006a (decimal:  106) --> SEGMENTATION VIOLATION (SEG1)
```

2.2 2.2

As explained above,
Here Address Space Size: 128, Physical Memory Size: 512, Value of Segment 0 Base Register: 0, Value of Segment 0 Limit Register: 20, Value of Segment 1 Base Register: 512, Value of Segment 1 Limit Register: 20

So for SEG0: Valid VAs are 0 to 19 for Phy.Addr: 0 to 19
For SEG1 valid VAs are 108 to 127 for Phys. Addr: 492 to 511

So,
The highest legal virtual address in segment 0: 19
The lowest legal virtual address in segment 1: 128 - 20 = 108
The lowest illegal virtual address in Entire Addr. Space: 20
The highest illegal virtual address in Entire Addr. Space: 107

Testing segmentation.py with -A flag can be done as follows:
python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1 -A 19,108,20,107

```
Then we see that,
NOTE: In case of Segmentation Violation, consider 1st half [0 to 63] for SEG0 and remaining
for SEG2[64 to 127]

$ python2 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1 -A 19,108,20,107
VA  0: 0x00000013 (decimal:   19) --> VALID in SEG0: 0x00000013 (decimal:   19) as 0+(19-0)=19
VA  1: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492) as 492+(108-108)
=492
VA  2: 0x00000014 (decimal:   20) --> SEGMENTATION VIOLATION (SEG0)
VA  3: 0x0000006b (decimal:  107) --> SEGMENTATION VIOLATION (SEG1)
```

2.3 2.3

To get such an order the valid VAs for SEG0 should be 0 to 1 and for SEG1 Valid VAs should be 14 to 15.
So, corresponding Phys. Addr for SEG0 should be: 0 to 1 and for SEG1 should be: 127 to 128
In such a case we would have
For SEG0 : Base as 0 and Bound/Limit as 2
For SEG2 : Base as 16 and Bound/Limit as 2

```
That is,
The command should be :
python2 segmentation.py -a 16 -p 128 -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 --b0 0 --l0 2
--b1 16 --l1 2
```

```
 Then we get the following output:
 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 2.4

We should have the limit value as 90% of address space size.
That is,
–l0 = 0.9 * (addr. space size)/2
–l1 = 0.9 * (addr. space size)/2

```
$ python2 segmentation.py -a 512 -p 1024 --b0 0 --l0 230 --b1 1024 --l1 230 -n 10 -c
ARG seed 0
ARG address space size 512
ARG phys mem size 1024

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 230

  Segment 1 base  (grows negative) : 0x00000400 (decimal 1024)
  Segment 1 limit                  : 230

Virtual Address Trace
  VA  0: 0x000001b0 (decimal:  432) --> VALID in SEG1: 0x000003b0 (decimal:  944)
  VA  1: 0x00000184 (decimal:  388) --> VALID in SEG1: 0x00000384 (decimal:  900)
  VA  2: 0x000000d7 (decimal:  215) --> VALID in SEG0: 0x000000d7 (decimal:  215)
  VA  3: 0x00000084 (decimal:  132) --> VALID in SEG0: 0x00000084 (decimal:  132)
  VA  4: 0x00000105 (decimal:  261) --> SEGMENTATION VIOLATION (SEG1)
  VA  5: 0x000000cf (decimal:  207) --> VALID in SEG0: 0x000000cf (decimal:  207)
  VA  6: 0x00000191 (decimal:  401) --> VALID in SEG1: 0x00000391 (decimal:  913)
```

```
  VA  7: 0x0000009b (decimal:  155) --> VALID in SEG0: 0x0000009b (decimal:  155)
  VA  8: 0x000000f4 (decimal:  244) --> SEGMENTATION VIOLATION (SEG0)
  VA  9: 0x0000012a (decimal:  298) --> VALID in SEG1: 0x0000032a (decimal:  810)
```

Here, we see that around 90% of VAs are Valid.

2.5 2.5

No Virtual will be valid when then limit for both the segments is set to 0 each and also bases can be initialized to 0 each.
That is,
-l=0
-L=0
which is same as,
–l0=0
–l1=0

Then there won't be any space allocated to any of the segments so all VAs will be invalid in such a case.

Command: python2 segmentation.py -b 0 -l 0 -B 0 -L 0

```
Example for 10 VAs:
$ python2 segmentation.py -b 0 -l 0 -B 0 -L 0 -n 10 -c
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) : 0x00000000 (decimal 0)
  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)
  VA  5: 0x0000019e (decimal:  414) --> SEGMENTATION VIOLATION (SEG0)
  VA  6: 0x00000322 (decimal:  802) --> SEGMENTATION VIOLATION (SEG1)
  VA  7: 0x00000136 (decimal:  310) --> SEGMENTATION VIOLATION (SEG0)
  VA  8: 0x000001e8 (decimal:  488) --> SEGMENTATION VIOLATION (SEG0)
  VA  9: 0x00000255 (decimal:  597) --> SEGMENTATION VIOLATION (SEG1)
```

3 Question 3

For Reference:

```
$ python2 paging-linear-size.py -v 20 -e 8 -p 4096 -c

ARG bits in virtual address 20
ARG page size 4096
ARG pte size 8

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

The number of bits in the virtual address: 20
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 8
Thus, a virtual address looks like this:

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

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 256.0
- The size of each page table entry, which is: 8
And then multiply them together. The final result:
  2048 bytes
  in KB: 2.0
  in MB: 0.001953125
```

With Lower No. of bits for VA:

```
$ python2 paging-linear-size.py -v 16 -e 8 -p 4096 -c

ARG bits in virtual address 16
ARG page size 4096
ARG pte size 8

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

The number of bits in the virtual address: 16
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 4
Thus, a virtual address looks like this:

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

where V is for a VPN bit and O is for an offset bit
```

```
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 16.0
- The size of each page table entry, which is: 8
And then multiply them together. The final result:
  128 bytes
  in KB: 0.125
  in MB: 0.0001220703125
```

With Higher No. of bits for VA:

```
$ python2 paging-linear-size.py -v 32 -e 8 -p 4096 -c

ARG bits in virtual address 32
ARG page size 4096
ARG pte size 8

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

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

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

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 1048576.0
- The size of each page table entry, which is: 8
And then multiply them together. The final result:
  8388608 bytes
  in KB: 8192.0
  in MB: 8.0
```

Conclusion:
This shows that as the no. of bits allocated for VA decrease, the Size of the Linear Page Table also decreases and vice versa. (multiplication factor is in terms of $2^{\text{change in VPN bits}}$ as Size of Page Table= $2^{\text{VPN bits count}} * (SizeofPTE)$)

With Lower PTE Size:

```
$ python2 paging-linear-size.py -v 20 -e 4 -p 4096 -c

ARG bits in virtual address 20
ARG page size 4096
ARG pte size 4

Recall that an address has two components:
```

```
[ Virtual Page Number (VPN) | Offset ]

The number of bits in the virtual address: 20
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 8
Thus, a virtual address looks like this:

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

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 256.0
- The size of each page table entry, which is: 4
And then multiply them together. The final result:
  1024 bytes
  in KB: 1.0
  in MB: 0.0009765625
```

With Higher PTE Size:

```
$ python2 paging-linear-size.py -v 20 -e 12 -p 4096 -c

ARG bits in virtual address 20
ARG page size 4096
ARG pte size 12

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

The number of bits in the virtual address: 20
The page size: 4096 bytes
Thus, the number of bits needed in the offset: 12
Which leaves this many bits for the VPN: 8
Thus, a virtual address looks like this:

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

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 256.0
- The size of each page table entry, which is: 12
And then multiply them together. The final result:
  3072 bytes
  in KB: 3.0
  in MB: 0.0029296875
```

Conclusion:
This shows that as the PTE Size decrease, the Size of the Linear Page Table also decreases and vice

versa. (multiplication factor is in terms of $2^{\text{change in VPN bits}}$ as Size of Page Table= $2^{\text{VPN bits count}} * (SizeofPTE)$)

With Lower Page Size:

```
$ python2 paging-linear-size.py -v 20 -e 8 -p 1024 -c

ARG bits in virtual address 20
ARG page size 1024
ARG pte size 8

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

The number of bits in the virtual address: 20
The page size: 1024 bytes
Thus, the number of bits needed in the offset: 10
Which leaves this many bits for the VPN: 10
Thus, a virtual address looks like this:

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

where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 1024.0
- The size of each page table entry, which is: 8
And then multiply them together. The final result:
  8192 bytes
  in KB: 8.0
  in MB: 0.0078125
```

With Higher Page Size:

```
$ python2 paging-linear-size.py -v 20 -e 8 -p 16384 -c

ARG bits in virtual address 20
ARG page size 16384
ARG pte size 8

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

The number of bits in the virtual address: 20
The page size: 16384 bytes
Thus, the number of bits needed in the offset: 14
Which leaves this many bits for the VPN: 6
Thus, a virtual address looks like this:

V V V V V V | O O O O O O O O O O O O O O
```

```
where V is for a VPN bit and O is for an offset bit
To compute the size of the linear page table, we need to know:
- The # of entries in the table, which is 2^(num of VPN bits): 64.0
- The size of each page table entry, which is: 8
And then multiply them together. The final result:
  512 bytes
  in KB: 0.5
  in MB: 0.00048828125
```

Conclusion:
This shows that as the Page Size decrease, the Size of the Linear Page Table also increases and vice versa. (multiplication factor is in terms of $2^{\text{change in VPN bits}}$ as Size of Page Table= $2^{\text{VPN bits count}} * (SizeofPTE)$).
And here as Page size increases we see that No. of Offset bits increase, hence the VPN bits decrease (as VPN bits +Offset Bits= Bits for VA [which is constant here]), if VPN bits increase then the Page Table Size will also increase (as Size of Page Table= $2^{\text{VPN bits count}} * (SizeofPTE)$).

4 Question 4

4.1 4.1

We know that,
Size of Page Table = Address Space Size / Page Size
So, we see that as Address Space Size increases then the Page Table size also increases.(as they are directly proportional)
And when Page size increases then the Page Table size decreases.(as they are inversely proportional)
Using big pages in general will cause wastage of space especially a scenario of Internal Fragmentation.

4.2 4.2

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

Case 2: python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 25
  VA 0x00003986 (decimal:14726) -->  Invalid (VPN 14 not valid)
  VA 0x00002bc6 (decimal:11206) --> 00004fc6 (decimal 20422) [VPN 10] (as 10*1024<11206<11*1024)
  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)

Case 3: python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 50
  VA 0x00003385 (decimal:13189) --> 00003f85 (decimal 16261) [VPN 12] (as 12*1024<11206<13*1024)
  VA 0x0000231d (decimal:8989) -->  Invalid (VPN 8 not valid)
```

```
  VA 0x000000e6 (decimal:230) --> 000060e6 (decimal 24806) [VPN 0] (as 0*1024<11206<1*1024)
  VA 0x00002e0f (decimal:11791) -->  Invalid (VPN 11 not valid)
  VA 0x00001986 (decimal:6534) --> 00007586 (decimal 30086) [VPN 6] (as 6*1024<11206<7*1024)

Case 4: python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 75
  VA 0x00002e0f (decimal:11791) --> 00004e0f (decimal 19983) [VPN 11] (as 11*1024<11206<12*1024)
  VA 0x00001986 (decimal:6534) --> 00007d86 (decimal 32134) [VPN 6] (as 6*1024<11206<7*1024)
  VA 0x000034ca (decimal:13514) --> 00006cca (decimal 27850) [VPN 13] (as 13*1024<11206<14*1024)
  VA 0x00002ac3 (decimal:10947) --> 00000ec3 (decimal 3779) [VPN 10] (as 10*1024<11206<11*1024)
  VA 0x00000012 (decimal:18) --> 00006012 (decimal 24594) [VPN 0] (as 0*1024<11206<1*1024)

Case 5: python2 paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 100
  VA 0x00002e0f (decimal:11791) --> 00004e0f (decimal 19983) [VPN 11] (as 11*1024<11206<12*1024)
  VA 0x00001986 (decimal:6534) --> 00007d86 (decimal 32134) [VPN 6] (as 6*1024<11206<7*1024)
  VA 0x000034ca (decimal:13514) --> 00006cca (decimal 27850) [VPN 13] (as 13*1024<11206<14*1024)
  VA 0x00002ac3 (decimal:10947) --> 00000ec3 (decimal 3779) [VPN 10] (as 10*1024<11206<11*1024)
  VA 0x00000012 (decimal:18) --> 00006012 (decimal 24594) [VPN 0] (as 0*1024<11206<1*1024)
```

Here, Addr space = 16K=2^{14},i.e. 14 bits in total. As Page size=1K=2^{10},i.e. 10 bits for offset. No. of Pages=$2^{14}/2^{10} = 2^4$,i.e. 4 bits for indexing pages

Explanation for Case 2:
VA2: 0x00002bc6: Binary 1010 1111000110.
So VPN: 1010=10, and Offset= 1111000110.
Then corresponding PFN for VPN(10) is: 0x80000013 in Binary: 00000000010011.
So Physical Addr=10011 1111000110=0x4fc6=20422 (replacing VPN in VA with corresponding PFN).

As we increase the percentage of pages that are allocated in each address space , we see that more memory access operations become valid however free space decreases.

4.3 4.3

The first and third cases are unrealistic. As in 1st case the address space page size is too small(external fragmentation), and in third case the address space page size is too big(internal fragmentation)

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

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

 Case 3: python2 paging-linear-translate.py -P 1m -a 256m -p 512m -v -s 3
  VA 0x0308b24d (decimal: 50901581) --> 1f68b24d (decimal 526955085) [VPN 48]
  VA 0x042351e6 (decimal: 69423590) -->  Invalid (VPN 66 not valid)
  VA 0x02feb67b (decimal: 50247291) --> 0a9eb67b (decimal 178173563) [VPN 47]
  VA 0x0b46977d (decimal: 189175677) -->  Invalid (VPN 180 not valid)
  VA 0x0dbcceb4 (decimal: 230477492) --> 1f2cceb4 (decimal 523030196) [VPN 219]
```

4.4 4.4

Some of the corners cases are shown below:

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

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


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

```
$ python2 paging-linear-translate.py -a 70k -p 60k -v
ARG seed 0
ARG address space size 70k
ARG phys mem size 60k
ARG page size 4k
ARG verbose True
ARG addresses -1

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

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

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

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

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

Other errors can include:
1. Address space is not multiple of page size.