Operating Systems Lab

Lab-7 Report

Pavan Kumar V Patil
200030041

March 5, 2023

Question 1

1.

For maximum access of memory for process is bound address

$$Bound\_address = base + limit$$

If process accesses more than limit ($VA\_access > limit$) gives segmentation violation. If process accesses less than limit ($VA\_access < limit$), physical address mapping is $base + VA\_access$.

For seed 1

Base : 13884 and Limit : 290

Virtual Address Trace		
VA's	**VA access**	**PA or Segmentation violation**
VA 0	782	Segmentation violation
VA 1	261	PA: 14145
VA 2	507	Segmentation violation
VA 3	460	Segmentation violation
VA 4	667	Segmentation violation

Table 1: Seed 1

For seed 2

Base : 15529 and Limit : 500

Virtual Address Trace		
VA's	**VA access**	**PA or Segmentation violation**
VA 0	57	PA: 15586
VA 1	86	PA: 15615
VA 2	855	Segmentation violation
VA 3	753	Segmentation violation
VA 4	685	Segmentation violation

Table 2: Seed 2

For seed 3

Base : 8916 and Limit : 316

Virtual Address Trace		
VA's	**VA access**	**PA or Segmentation violation**
VA 0	378	Segmentation violation
VA 1	618	Segmentation violation
VA 2	640	Segmentation violation
VA 3	67	PA: 8983
VA 4	13	PA: 8929

Table 3: Seed 3

2.

For running flags -s 0 -n 10, Base is 13835 and VA accesses are 776,430,265,523,414,802,310,488,597 and 929. Among all VA accesses 929 is the highest, so if we set **Limit as 930** then no VA's are outside the bound.

3.

For running flags -s 1 -n 10 -l 100, Limit is 100 and size of Physical memory is 16k which is 16384 bytes, The maximum base is **16384 - 100 = 16284**

4.

For running flags **-s 0 -n 10 -a 2m -p 32m**

```
pavan@pavan-OMEN-Laptop-15-ek0xxx:~/Downloads/cs314oslaboratory7$ python2 relocation.py -s 0 -n 10 -a 2m -p 32m

ARG seed 0
ARG address space size 2m
ARG phys mem size 32m

Base-and-Bounds register information:

  Base   : 0x01841299 (decimal 25432729)
  Limit  : 967008

Virtual Address Trace
  VA  0: 0x000d7552 (decimal: 882002) --> PA or segmentation violation?
  VA  1: 0x0008490b (decimal: 542987) --> PA or segmentation violation?
  VA  2: 0x00105c5c (decimal: 1072220) --> PA or segmentation violation?
  VA  3: 0x000cf538 (decimal: 849208) --> PA or segmentation violation?
  VA  4: 0x001914e0 (decimal: 1643744) --> PA or segmentation violation?
  VA  5: 0x0009b4bc (decimal: 636092) --> PA or segmentation violation?
  VA  6: 0x000f4048 (decimal: 999496) --> PA or segmentation violation?
  VA  7: 0x0012ab10 (decimal: 1223440) --> PA or segmentation violation?
  VA  8: 0x001d0f42 (decimal: 1904450) --> PA or segmentation violation?
  VA  9: 0x00102665 (decimal: 1058405) --> PA or segmentation violation?
```

Figure 1: -s 0 -n 10 -a 2m -p 32m

Among all VA accesses 1984450 is the highest, so if we set **Limit as 1984451** then no VA's are outside the bound.

For running flags **-s 1 -n 10 -l 100 -a 2m -p 32m**

```
pavan@pavan-OMEN-Laptop-15-ek0xxx:~/Downloads/cs314oslaboratory7$ python2 relocation.py -s 1 -n 10 -l 100 -a 2m -p 32m

ARG seed 1
ARG address space size 2m
ARG phys mem size 32m

Base-and-Bounds register information:

  Base   : 0x0044cb63 (decimal 4508515)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x001b1e2d (decimal: 1777197) --> PA or segmentation violation?
  VA  1: 0x001870d7 (decimal: 1601751) --> PA or segmentation violation?
  VA  2: 0x00082986 (decimal: 534918) --> PA or segmentation violation?
  VA  3: 0x000fda9a (decimal: 1039002) --> PA or segmentation violation?
  VA  4: 0x000e623b (decimal: 942651) --> PA or segmentation violation?
  VA  5: 0x0014d9d9 (decimal: 1366489) --> PA or segmentation violation?
  VA  6: 0x00193d38 (decimal: 1654072) --> PA or segmentation violation?
  VA  7: 0x000300e5 (decimal: 196837) --> PA or segmentation violation?
  VA  8: 0x0000e838 (decimal: 59448) --> PA or segmentation violation?
  VA  9: 0x001abe96 (decimal: 1752726) --> PA or segmentation violation?
```

Figure 2: -s 1 -n 10 -l 100 -a 2m -p 32m

Limit is 100 and size of Physical memory is 32m which is 33554432 bytes, The maximum base is **33554432 - 100 = 33554332**

5.

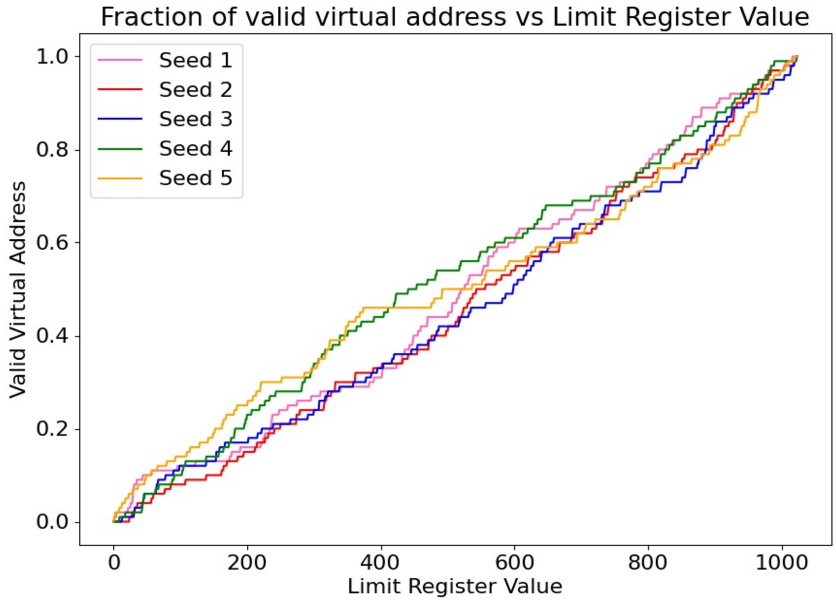


Figure 3: Graph of Fraction of valid address vs Limit register Value

Question 2

1.

If process accesses memory more than segment1 limit and less than VA - segment2 limit [$(VA\_access > segment1\_limit)$ and $(VA\_access < VA - segment2\_limit)$] gives segmentation violation. If process accesses less than segment1 limit $(VA\_access < limit)$ and more than $(VA\_access > VA - segment2\_limit)$. Physical address mapping, if it falls in segment1 $segment1\_base + VA\_access$ and if it falls in segment2 $segment2\_base - (VA - VA\_access)$.

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

Segment 0 base: 0 and Segment 0 limit: 20
Segment 1 base: 512 and Segment 1 limit: 20

Virtual Address Trace		
VA's	**VA access**	**PA or Segmentation violation**
VA 0	108	PA: 492
VA 1	97	Segmentation violation
VA 2	53	Segmentation violation
VA 3	33	Segmentation violation
VA 4	65	Segmentation violation

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

Segment 0 base: 0 and Segment 0 limit: 20
Segment 1 base: 512 and Segment 1 limit: 20

Virtual Address Trace		
VA's	**VA access**	**PA or Segmentation violation**
VA 0	17	PA: 17
VA 1	108	PA: 492
VA 2	97	Segmentation violation
VA 3	32	Segmentation violation
VA 4	63	Segmentation violation

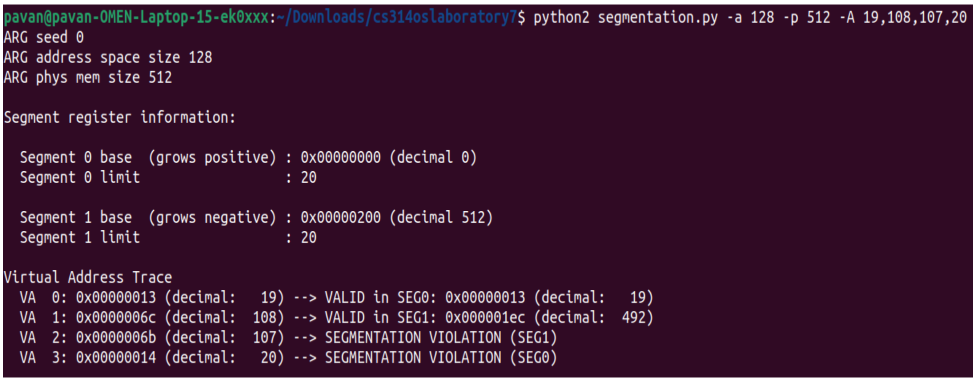
for segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2

Segment 0 base: 0 and Segment 0 limit: 20
Segment 1 base: 512 and Segment 1 limit: 20

Virtual Address Trace		
VA's	**VA access**	**PA or Segmentation violation**
VA 0	122	PA: 506
VA 1	121	PA: 505
VA 2	7	PA: 7
VA 3	10	PA: 10
VA 4	106	Segmentation violation

2.

Highest Legal virtual address in segment 0: **19**
Lowest Legal virtual address in segment 1: **108**
Highest illegal virtual address is: **107**
Lowest illegal virtual address is: **20**
We run command **segmentation.py -a 128 -p 512 -A 19,108,107,20 -b 0 -l 20 -B 512 -L 20 -c** to check out the answer



```
pavan@pavan-OMEN-Laptop-15-ek0xxx:~/Downloads/cs314oslaboratory7$ python2 segmentation.py -a 128 -p 512 -A 19,108,107,20
ARG seed 0
ARG address space size 128
ARG phys mem size 512

Segment register information:

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

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

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

3.

To make first 2 as valid access and last 2 as valid access and other as violation, we have to run command **segmentation.py -a 16 -p 128 -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 -b 0 -l 2 -B 128 -L 2**

Therefore answer is:
b0 = 0
l0 = 2
b1 = 128
l1 = 2

4.

To configure to get 90% of the randomly-generated VA are we have set l0 and l1 such that they cover 90% of VA's for example:
./segmentation.py -a 100 -p 512 -n 1000 –b0 0 –l0 45 –b1 100 –l1 45 -c
l0 and l1 parameters are important to getting this outcome.

5.

Yes, we can run simulator such that no Virtual addresses are valid. To do so we have to keep both the segmentation limit as **0 (zero)**.

1 Question 3

case1: paging-linear-size.py -v 32 -e 4 -p 4k

Bits in virtual address: 32
Page size: 4k
Page Table Entry size: 4
The number of bits in the virtual address: **32**
The page size: **4096 bytes**. Thus, the number of bits needed in the offset: **12**
The size of memory = $\mathbf{2^{32}}$
The number of Page entries = $\boldsymbol{2^{32}/page\_size(2^{12})} = \mathbf{2^{20}}$
The size of page table = no. of page entries * page table entry size($\mathbf{2^{20} * 4 = 4194304}$)

case2: paging-linear-size.py -v 64 -e 4 -p 4k

Bits in virtual address: 64
Page size: 4k
Page Table Entry size: 4
The number of bits in the virtual address: **64**
The page size: **4096 bytes**. Thus, the number of bits needed in the offset: **12**
The size of memory = $\mathbf{2^{64}}$
The number of Page entries = $\boldsymbol{2^{64}/page\_size(2^{12})} = \mathbf{2^{52}}$
The size of page table = no. of page entries * page table entry size($\mathbf{2^{52} * 4 = 18014398509481984}$)

case3: paging-linear-size.py -v 64 -e 8 -p 16k

Bits in virtual address: 64
Page size: 16k
Page Table Entry size: 8
The number of bits in the virtual address: **64**
The page size: **16384 bytes**. Thus, the number of bits needed in the offset: **14**
The size of memory = $\mathbf{2^{64}}$
The number of Page entries = $\boldsymbol{2^{64}/page\_size(2^{14})} = \mathbf{2^{50}}$
The size of page table = no. of page entries * page table entry size($\mathbf{2^{50} * 8 = 9007199254740992}$)

Question 4

1.

paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0

Page size: 1k = $\mathbf{2^{10}}$
Number of page entries = $\boldsymbol{virtual\_space\_address}(2^{20})/\boldsymbol{page\_size}(2^{10}) = \mathbf{2^{10}}$
Page entry size = **4**
Size of Page Table = $\mathbf{2^{10} * 4 = 4096}$ **bytes**

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

Page size: 1k = $\mathbf{2^{10}}$
Number of page entries = $\boldsymbol{virtual\_space\_address}(2^{21})/\boldsymbol{page\_size}(2^{10}) = \mathbf{2^{11}}$
Page entry size = **4**
Size of Page Table = $\mathbf{2^{11} * 4 = 8192}$ **bytes**

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

Page size: 1k = $\mathbf{2^{10}}$
Number of page entries = $\boldsymbol{virtual\_space\_address}(2^{22})/\boldsymbol{page\_size}(2^{10}) = \mathbf{2^{12}}$
Page entry size = **4**
Size of Page Table = $\mathbf{2^{12} * 4 = 16384}$ **bytes**

As virtual space **doubles** the size of page table **doubles**.

paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0

Page size: 1k = $\mathbf{2^{10}}$
Number of page entries = $\boldsymbol{virtual\_space\_address}(2^{20})/\boldsymbol{page\_size}(2^{10}) = \mathbf{2^{10}}$
Page entry size = **4**
Size of Page Table = $\mathbf{2^{10} * 4 = 4096}$ **bytes**

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

Page size: 2k = $\mathbf{2^{11}}$
Number of page entries = $\boldsymbol{virtual\_space\_address}(2^{20})/\boldsymbol{page\_size}(2^{11}) = \mathbf{2^{9}}$
Page entry size = **4**
Size of Page Table = $\mathbf{2^{9} * 4 = 2048}$ **bytes**

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

Page size: 4k = $\mathbf{2^{12}}$
Number of page entries = $\boldsymbol{virtual\_space\_address}(2^{20})/\boldsymbol{page\_size}(2^{12}) = \mathbf{2^{8}}$
Page entry size = **4**
Size of Page Table = $\mathbf{2^{8} * 4 = 1024}$ **bytes**

As page size **doubles**, page table size **halves**

Table size will decrease as page size increase. But the problem with larger page would be **internal fragmentation** in the pages.

2.

Page size is 1k, log(1k) = **10** bits are required to describe every address. Address space size is 16k. There can be 16 pages in total, so log(16) = **4** more bits are required to describe a page number.

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

All the addresses are invalid as none of the pages are allocated.

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

For VA: 11206 the binary form is **10101111000110**. The starting four bits value is 10, therefore the VPN is **10**

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

For VA: 13189 the binary form is **11001110000101**. The starting four bits value is 12, therefore the VPN is **12**
For VA: 230 the binary form is **00000011100110**. The starting four bits value is 0, therefore the VPN is **0**
For VA: 6534 the binary form is **1100110000110**. The starting four bits value is 6, therefore the VPN is **6**

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

For VA: 11791 the binary form is **10111000001111**. The starting four bits value is 11, therefore the VPN is **11**
For VA: 13514 the binary form is **11010011001010**. The starting four bits value is 13, therefore the VPN is **13**
For VA: 6534 the binary form is **1100110000110**. The starting four bits value is 6, therefore the VPN is **6**
For VA: 10947 the binary form is **10101011000011**. The starting four bits value is 10, therefore the VPN is **10**
For VA: 18 the binary form is **00000000010010**. The starting four bits value is 0, therefore the VPN is **0**

As, the percentage of pages that are allocated or usage of address space is increased more and more memory access operations become valid and free space decreases

3.

The **-P 8 -a 32 -p 1024 -v -s 1** is **unrealistic** because the page size, virtual address, physical memory is very small.
The **-P 1m -a 256m -p 512m -v -s 3** is **unrealistic** because the page size is very big.

4.

For the following cases program doesn't work:

- Physical memory size is less or equal to the address space size.
- Page is bigger than address space size.
- Page size or address space size is not power of 2. This is because it will result in discontinuity in addresses.
- Page or address space size is zero

When address space is bigger than physical memory, the proper mapping can't be done to physical memory and system working fails.