Shashank P
200010048

CS314 Operating Systems Lab
Lab 7

1 Relocation.py

The segmentation violation and address translation are shown below one after the other.

$$\textbf{Virtual\_Address} \geq \textbf{limit}$$

$$\textbf{Physical\_Address} = \textbf{Base} + \textbf{Virtual\_Address}$$

1.1 Seed 1, 2 and 3

For $Seed = 1$ we have, $Base = 13884$ and $Limit = 290$

Virtual Address	**Physical Address**
782	segmentation violation
261	14145
507	segmentation violation
460	segmentation violation
667	segmentation violation

Table 1: Seed: 1

For $Seed = 2$ we have, $Base = 15529$ and $Limit = 500$

Virtual Address	**Physical Address**
57	15586
86	15615
855	segmentation violation
753	segmentation violation
685	segmentation violation

Table 2: Seed: 2

For $Seed = 3$ we have, $Base = 8916$ and $Limit = 316$

Virtual Address	**Physical Address**
378	segmentation violation
618	segmentation violation
640	segmentation violation
67	8983
13	8929

Table 3: Seed: 3

1.2 Limit Register Value

The highest address access was at virtual address **929**; therefore, setting the limit register to **930** would be sufficient, as shown below.

```
$ python 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) --> 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 1: Limit Register

1.3 Maximum Base Register

Since the limit register is set to **100** and the physical memory size is **16k** = **16384**, the maximum base register such that the entire address space fits $16384 - 100 = 16284$.

```
$ python relocation.py -s 1 -n 10 -l 100 -b 16285

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

Base-and-Bounds register information:

  Base   : 0x00003f9d (decimal 16285)
  Limit  : 100

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

Figure 2: $Base = \mathbf{16285}$

1.4 Larger Address and Physical Spaces

On increasing parameters I observed the following:

1. Keeping Physical Space constant and increasing just the address space, the number of requests out of bound was the same as even the **Limit** register value increased. The address space could only be increased up to the physical memory size.

2. In the above case on keeping the limit register fixed, most of the requests were going out of bound as even though the address space is huge (say $2m$) the limit is only 100, and the probability that among the $2m$ space only the first 100 are requested is very small.

3. On keeping address space fixed and increasing physical memory did not affect the answers except for the third part, where we explicitly used the size of physical address space.

4. Physical address space did not affect the results because even though the physical address space is huge, the process only knows about the address space it can access and nothing else, which in this case was kept constant

1.5 Plot

For the plots, I have used a large physical memory (say **10m**) and address space size to be **1k**. The number of requests was kept to **500**. Then I increased the limit from **1 - 1k**. The number of valid requests was averaged over **5** seeds and plotted as shown below.

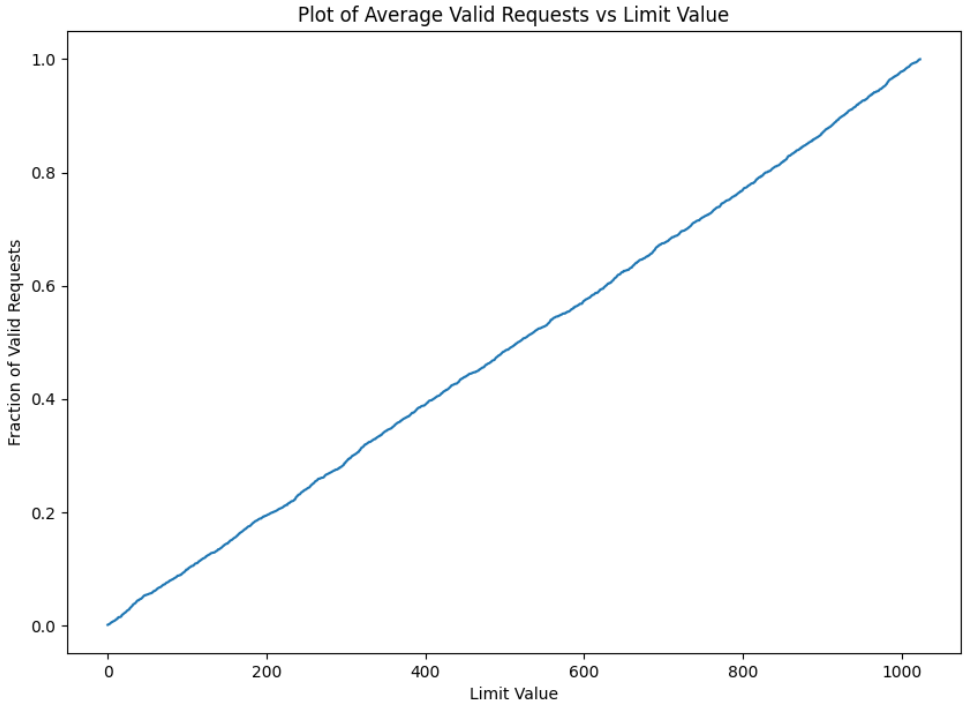

Figure 3: Fraction of Valid Requests vs Limit Value

The code used to run the simulation is shown below.

```
for seed in {1..5}
do
    for l in {1..1024}
    do
        python relocation.py -s $seed -l $l -n 500 -a 1k -p 10m -c >> log.txt
    done
done
```

From the plot, we can observe that as the limit register value increases, more and more requests start to become valid, which is as expected.

2 Segementation.py

The condition for segmentation violation is given below:

$$Virtual\_Address \geq limit\_0 \quad and$$
$$Virtual\_Address < Address\_Space\_Size - limit\_1$$

To translate a valid virtual address to a physical address, we can do the following:

$$Physical\_Address = Base\_0 + Virtual\_Address \quad (Segement\ 0)$$
$$Physical\_Address = Base\_1 - (Address\_Space - Virtual\_Address) \quad (Segement\ 1)$$

2.1 Small Address Spaces

First is **segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0**

Virtual Address	**Segment No.**	**Physical Address**
108	Segment 1	492
97	segmentation violation	-
53	segmentation violation	-
33	segmentation violation	-
65	segmentation violation	-

Table 4: Seed: 0

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

Virtual Address	**Segment No.**	**Physical Address**
17	Segment 0	17
108	Segment 1	492
97	segmentation violation	-
32	segmentation violation	-
63	segmentation violation	-

Table 5: Seed: 1

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

Virtual Address	**Segment No.**	**Physical Address**
122	Segment 1	506
121	Segment 1	505
7	Segment 0	7
10	Segment 0	10
106	segmentation violation	-

Table 6: Seed: 2

2.2 Extreme Addresses

Using the conditions for segmentation violation:

1. Highest legal in Segment 0: **19**
2. Lowest legal in Segment 1: **108**
3. Highest illegal in entire address space: **107**
4. Lowest illegal in entire address space: **20**

Using **-A** flag, I can manually provide the translation requests. The command and the output are shown below, and the output is verified.

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

Figure 4: Output for Given Extreme Values

2.3 Base and Bound Registers

Given initial parameters **-a 16 -p 128 -A 0,1,2,3 ... 15** and the output to be **valid, valid, invalid, ..., invalid, valid, valid**. We must set limit registers to **2** and choose appropriate base registers. $b0 = 0,\ l0 = 2,\ b1 = 128,\ l1 = 2$.

The output is shown below and the output is verified.

```
$ ./segmentation.py -a 16 -p 128 --b0 0 --l0 2 --b1 128 --l1 2 -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 -c
ARG seed 0
ARG address space size 16
ARG phys mem size 128

Segment register information:

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

  Segment 1 base  (grows negative) : 0x00000080 (decimal 128)
  Segment 1 limit                  : 2

Virtual Address Trace
  VA  0: 0x00000000 (decimal:    0) --> VALID in SEG0: 0x00000000 (decimal:    0)
  VA  1: 0x00000001 (decimal:    1) --> VALID in SEG0: 0x00000001 (decimal:    1)
  VA  2: 0x00000002 (decimal:    2) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x00000003 (decimal:    3) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x00000004 (decimal:    4) --> SEGMENTATION VIOLATION (SEG0)
  VA  5: 0x00000005 (decimal:    5) --> SEGMENTATION VIOLATION (SEG0)
  VA  6: 0x00000006 (decimal:    6) --> SEGMENTATION VIOLATION (SEG0)
  VA  7: 0x00000007 (decimal:    7) --> SEGMENTATION VIOLATION (SEG0)
  VA  8: 0x00000008 (decimal:    8) --> SEGMENTATION VIOLATION (SEG1)
  VA  9: 0x00000009 (decimal:    9) --> SEGMENTATION VIOLATION (SEG1)
  VA 10: 0x0000000a (decimal:   10) --> SEGMENTATION VIOLATION (SEG1)
  VA 11: 0x0000000b (decimal:   11) --> SEGMENTATION VIOLATION (SEG1)
  VA 12: 0x0000000c (decimal:   12) --> SEGMENTATION VIOLATION (SEG1)
  VA 13: 0x0000000d (decimal:   13) --> SEGMENTATION VIOLATION (SEG1)
  VA 14: 0x0000000e (decimal:   14) --> VALID in SEG1: 0x0000007e (decimal:  126)
  VA 15: 0x0000000f (decimal:   15) --> VALID in SEG1: 0x0000007f (decimal:  127)
```

Figure 5: Output for the given sequence

2.4 90% Valid Requests

We can achieve this by tuning the **address_space size** and limit registers. The Physical memory should be larger than the address space, and Base registers pointing in physical memory should not overlap the 2 segments.

1. We can set the address space size to **128 bytes** and physical memory to **512 bytes**.
2. Let Base_0 be **0** and Base_1 be **512**.
3. Now if we set both the limit registers to **45%** of address space size, only $2 * 45 = 90\%$ of the address space is accessible. Therefore on average 90% of the requests are satisfied.

We can set **l0** $= 0.45 * 128 = 58$ bytes and **l1** $= 58$ bytes.

2.5 All invalid requests

We can run the simulation such that all requests are invalid by setting limit registers to **0** and all other parameters to be the same as before. In this case, the segmentation violation condition will always be true. And none of the address spaces will be accessible.

3 paging-linear-size.py

To calculate the page table size, we first check how many bits among the virtual address are required for Virtual Page Number (VPN).

1. Bits for $VPN = VA\_bits - Offset$
2. Here, bits in offset can be calculated by taking $log_2(page\_table\_size)$; this is because these many bits are required to resolve addresses per page table.
3. Next, we can find the number of entries in the page table by using $2^{bits\_in\_VPN}$
4. Final size can be obtained by multiplying the size of each entry.

The final formula is:

$$Size \ = \ 2^{\ bits\_in\_VA \ - \ log_2(\ page\_size\ )} * per\_entry\_size$$

Using the above formula, we can test some cases:

1. The page size should be set to a power of **2** for consistent results.
2. Also, for consistent results $bits\_in\_VA \geq log_2(\ page\_size\ )$, otherwise number of entries in the page table becomes fractional.
3. For **v = 32, e = 4, p = 4k** size of page table $S = 4194304$
4. Doubling e, **v = 32, e = 8, p = 4k** size of page table $S = 8388608$ also doubles which is consistent with the result.
5. Increasing v, by 3 **v = 35, e = 4, p = 4k** size of page table $S = 33554432$ becomes **8** times more which is again consistent with the result.
6. Increasing p, by 8 times **v = 32, e = 4, p = 32k** size of page table $S = 524288$ becomes **8** times less which is again consistent with the result.
7. For **v = 9, e = 4, p = 4k** I got $S = 0$, which is inconsistent as the initial input does not satisfy the required constraints. (Point 2)
8. For **v = 9, e = 4, p = 4055**, I encountered an error as the size of the page table was not the power of 2**.** (Point 1)

4 paging-linear-translate.py

4.1 Variation of Page size

Given page size P and address space size a and assuming each entry to be of size **4 bytes**, the number of entries can be calculated as:

$$Entries = \frac{a}{P}$$

This can be multiplied by the size per entry (say **4 bytes**) to obtain the actual size. Note that here **a** should be divisible by **P** or else we get fractional entries.

4.1.1 Variation with Address space size

On testing with different initial values, we can observe the following:

1. For **-P 1k -a 1m -p 512m -v -n 0**, Entries = 1024
2. For **-P 1k -a 2m -p 512m -v -n 0**, Entries = 2048
3. For **-P 1k -a 4m -p 512m -v -n 0**, Entries = 4096

We can see that number of entries gets doubled as the address space doubles. The results can be verified with the formula given at the beginning of the section. To get the size, we can multiply the number of entries with **4 bytes** to get 4096 bytes, 8192 bytes and 16384 bytes, respectively.

4.1.2 Variation with Page size

On testing with different initial values, we can observe the following:

1. For **-P 1k -a 1m -p 512m -v -n 0**, Entries = 1024
2. For **-P 2k -a 1m -p 512m -v -n 0**, Entries = 512
3. For **-P 4k -a 1m -p 512m -v -n 0**, Entries = 256

We can see that number of entries gets halved as the page size doubles. The results can be verified with the formula given at the beginning of the section. To get the size, we can multiply the number of entries with **4 bytes** to get 4096 bytes, 2048 bytes and 1024 bytes, respectively.

If we use large page sizes, say equal to address space size. There is only **1** entry in the page table and will therefore resemble the base and bound system of memory allocation. A contiguous block will be provided in the physical memory. This can lead to **internal fragmentation**, i.e., most entries in between will be unallocated in the physical memory.

4.2 Allocated Space Percent

On changing the value of **u** that is a fraction of allotted space from 0 to 100 and generating around **10 requests** for address translation, the following results are obtained:

1. For **-P 1k -a 16k -p 32k -v -u 0**, Valid requests = 0.
2. For **-P 1k -a 16k -p 32k -v -u 25**, Valid requests = 2.

3. For **-P 1k -a 16k -p 32k -v -u 50**, Valid requests = 6.
4. For **-P 1k -a 16k -p 32k -v -u 75**, Valid requests = 10.
5. For **-P 1k -a 16k -p 32k -v -u 100**, Valid requests = 10.

As we can see, the number of possible randomly generated requests being valid **increases** as a fraction of allotted space **increases**.

4.3 Unrealistic Parameters

Among the given initial parameters:

1. -P 8 -a 32 -p 1024 -v -s 1
2. -P 8k -a 32k -p 1m -v -s 2
3. -P 1m -a 256m -p 512m -v -s 3

All the above cases are possible theoretically as none of the size constraints is being broken. However, in a practical sense, the first option seems unrealistic because the page size is too small, which will cause space overhead in storing the page table itself (Section 4.1). The address space available for each process is also too small to run any reasonable application. The others are reasonable values of parameters.

4.4 Breaking the simulation

On trying out different values, the simulation fails in the following cases:

- When the address space size exceeds the physical memory, we cannot map between the virtual and physical addresses. Therefore, it breaks. Ex. **-a 512m -p 256m**

```
ARG address space size 512m
ARG phys mem size 256m
ARG page size 4k
ARG verbose False
ARG addresses -1

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

- When address space size is not a multiple of page size, we get a fractional number of entries in the page table, which again breaks the simulation. Ex. **-P 4555 -a 256k -p 1m**

```
ARG address space size 256k
ARG phys mem size 1m
ARG page size 4555
ARG verbose False
ARG addresses -1

Error in argument: address space must be a multiple of the pagesize
```

- Similarly, even the physical memory size has to be a multiple of page size, as otherwise, we obtain a fractional number of physical frames. Ex. **-P 500 -a 1000 -p 1m**

```
ARG address space size 1000
ARG phys mem size 1m
ARG page size 500
ARG verbose True
ARG addresses -1

Error in argument: physical memory must be a multiple of the pagesize
```

- The address space has to be a power of 2 for binary indexing. Ex. **-P 500 -a 1000 -p 4000**

```
ARG address space size 1000
ARG phys mem size 4000
ARG page size 500
ARG verbose True
ARG addresses -1

Error in argument: address space must be a power of 2
```

- The simulation also fails when negative values are passed in as arguments.