Operating Systems Lab 7

Divy Jain (210010015)

March 11, 2024

Question 1

The program relocation.py allows you to see how address translations are performed in a system with base and bounds registers. See the README_base_bound for more details.

Solution

1. SEG denotes Segmentation Violation, VA and PA are Virtual Address and Physical Address. PA = VA + base (if limit > VA)

	Seed 1		Seed 2		Seed 3	
Valid Range	13884	14174 (+290)	15529	16029(+500)	8916	9232 (+316)
	VA	PA	VA	PA	VA	PA
0	782	SEG	57	15586	378	SEG
1	261	14145	86	15615	618	SEG
2	507	SEG	855	SEG	640	SEG
3	460	SEG	753	SEG	67	8983
4	667	SEG	685	SEG	13	8929

Table 1: VA to PA

2. Max value of Virtual addresses is 929, so we have to set bound to 930.

3. Since maximun pyhical address values is 16384 (16kb) the maximum value for base is 16284 if the bound is 100.

4.
```
python2 relocation.py -s 1 -a 64k -p 32m -c

ARG seed 1
ARG address space size 64k
ARG phys mem size 32m

Base-and-Bounds register information:

  Base   : 0x01b1e2d5 (decimal 28435157)
  Limit  : 18585

Virtual Address Trace
VA  0: 0x0000c386 (decimal: 50054) --> SEGMENTATION VIOLATION
VA  1: 0x0000414c (decimal: 16716) --> VALID: 0x01b22421 (decimal: 28451873)
VA  2: 0x00007ed4 (decimal: 32468) --> SEGMENTATION VIOLATION
VA  3: 0x00007311 (decimal: 29457) --> SEGMENTATION VIOLATION
VA  4: 0x0000a6ce (decimal: 42702) --> SEGMENTATION VIOLATION
```

   for address space of 64k and physical memory of 32mb, we will get minimum bound for all valid VA as $55537 + 1 = (maxVA + 1) = 50055$ and maximum base value for given bound value (18585) is $physicalmem - limit = 33554432 - 18585 = 33535847$.

5. plot is given below, min bound value = 0 and max bound value is 1k default.

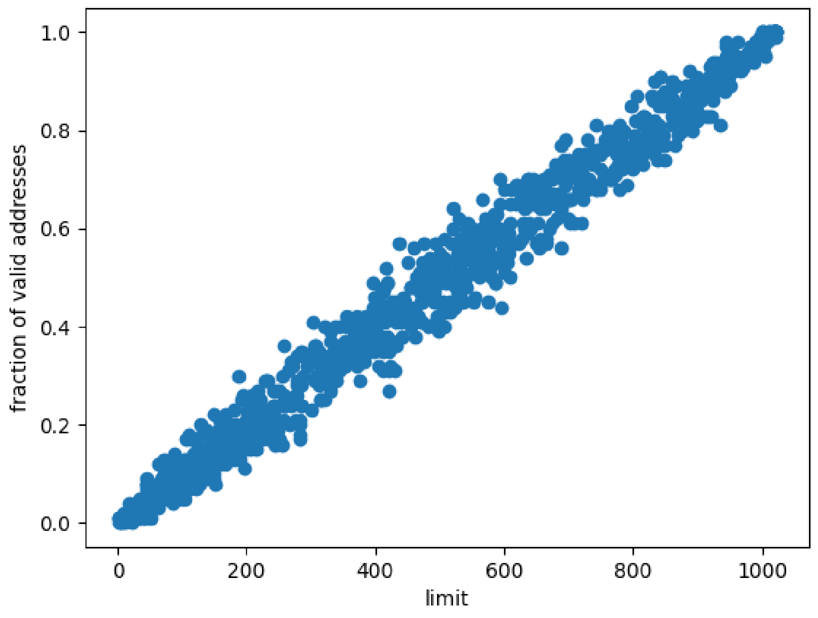

Figure 1: plot

Question 2

The program segmentation.py allows you to see how address translations are performed in a system with segmentation. See the README_segmentation for more details.

Solution

1. Translated addresses (SEG - Segmentation Violation)

	Seed 1		Seed 2		Seed 3	
Valid VA Range	0 - 19	108-127	0 - 19	108-127	0 - 19	108-127
	VA	PA	VA	PA	VA	PA
0	108	492	17	17	122	506
1	97	SEG	108	492	121	507
2	53	SEG	97	SEG	7	7
3	33	SEG	32	SEG	10	SEG
4	65	SEG	63	SEG	106	SEG

Table 2: VA to PA

2. highest valid (< 20) VA for SEG0 is 19. minmum valid VA for seg1 is 108. Lowest illegal address is 20. Highest illegal address is 107. The command for verification is below.

```
--------------- virtual address 0
 |     seg0     |
 |              |
 |              |
 |--------------|legal seg0 = 19
 |              |illegal after seg0 = 20
 |              |
 |              |
 |              |
 |(unallocated)|
 |              |
 |              |
 |              |illegal before seg 1 = 107
 |--------------|legal seg1 = 108
 |              |
 |     seg1     |
 |--------------| virtual address max (size of address space = 128 - 1 = 127)

python2 segmentation.py -a 128 -p 512 -c --b0 0 --l0 20 --b1 512 --l1 20 -A
    18,19,20,107,108,109
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: 0x00000012 (decimal:   18) --> VALID in SEG0: 0x00000012 (decimal:
    18)
VA  1: 0x00000013 (decimal:   19) --> VALID in SEG0: 0x00000013 (decimal:
    19)
VA  2: 0x00000014 (decimal:   20) --> SEGMENTATION VIOLATION (SEG0)
VA  3: 0x0000006b (decimal:  107) --> SEGMENTATION VIOLATION (SEG1)
VA  4: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:
    492)
VA  5: 0x0000006d (decimal:  109) --> VALID in SEG1: 0x000001ed (decimal:
    493)
```

3. we will have to run the following command, setting values of b0 = 0, l0 = 2, b1 = 128 and l1 = 2.

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

4. Virtual Addresses are uniformly generated between 0 to Address Space - 1 (inclusive). To get 90% of the virtual addresses correct we will have to make same amount of total segment space. For this, we can have b0 = 0, l0 = int(a*0.45), b1 = p, l1 = int(a*0.45), where a = address space size, p = physical memory . The important parameters are segments base and limit values.

5. if we run with zero limit values (l0, l1) then there will be no valid virtual addresses. The segment size in this case will be zero.

Question 3

The program paging-linear-size.py lets you figure out the size of a linear page table given a variety of input parameters. Compute how big a linear page table is with the characteristics such as different number of bits in the address space, different page size, different page table entry size. Explain your answers for various cases.

Solution

Page table size is affected by the following parameters. We will compare with the following base case.

```
python2 paging-linear-size.py -c
ARG bits in virtual address 32
ARG page size 4k
ARG pte size 4

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: 4
And then multiply them together. The final result:
  4194304 bytes
  in KB: 4096.0
  in MB: 4.0
```

Bits in Virtual Address The page table increases exponentially with the increase in the number of VPN bits in the address. VPN bits are number of bits in the virtual address minus the offset bits.

Page Table Entry size The page table size increases linearly with increase in page entry size.

Page size This is used in calculating the offset bits in the virtual address. Since more the offset bits, lesser are the Virtual Page Number bits(VPN), the Page table size decreases exponentially with increase in page size.

Formulas

$$\text{Offset} = log2(\text{Page Size})$$

$$\text{VAbits} = \text{VPN bits} + \text{Offset bits}$$

$$\text{Page Table Size} = 2^{(\text{VAbits}-\text{OffsetBits})} * \text{Page Table Entry Size}$$

Question 4

You will use the program, “paging-linear-translate.py” to see if you understand how simple virtual-to-physical address translation works with linear page tables. See the README_paging for more details.

Solution

1. As address space increases Page Table grows exponentially, but Page Table decreases exponentially as Page Size increase. We can not directly increase the Page size because this will increase the total space used by page table and most of space in the process (having one big page) will not be utilized.

2. As we increase the percentage of address space with allocated pages, we see that the more number of pages are valid (valid bit = 1) and number of Virtual Addresses are increasing.

3. below combination is unrealistic.

   ```
   python2 paging-linear-translate.py -P 1m -a 256m  -p 512m -v -c -s 3
   ```

   In this combination total memory is 512m and single page table size is 256m which is **half** of that total physical memory.

4. The program crashes or gives error message in following cases

   - If the address space is more the physical memory
   - If the page size is more than address space