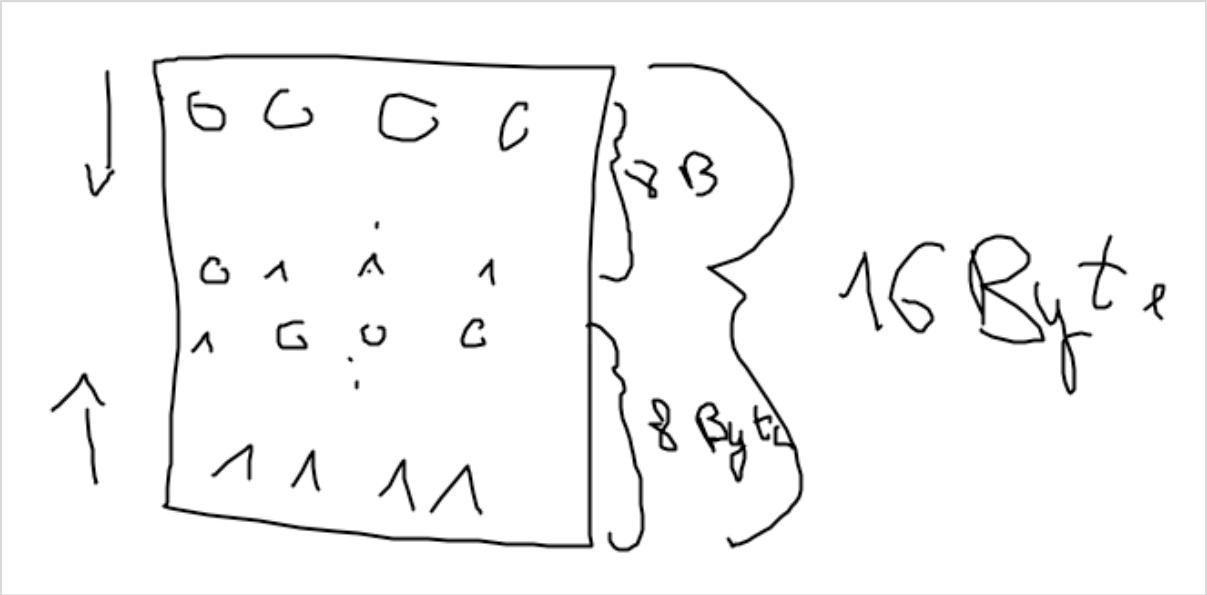
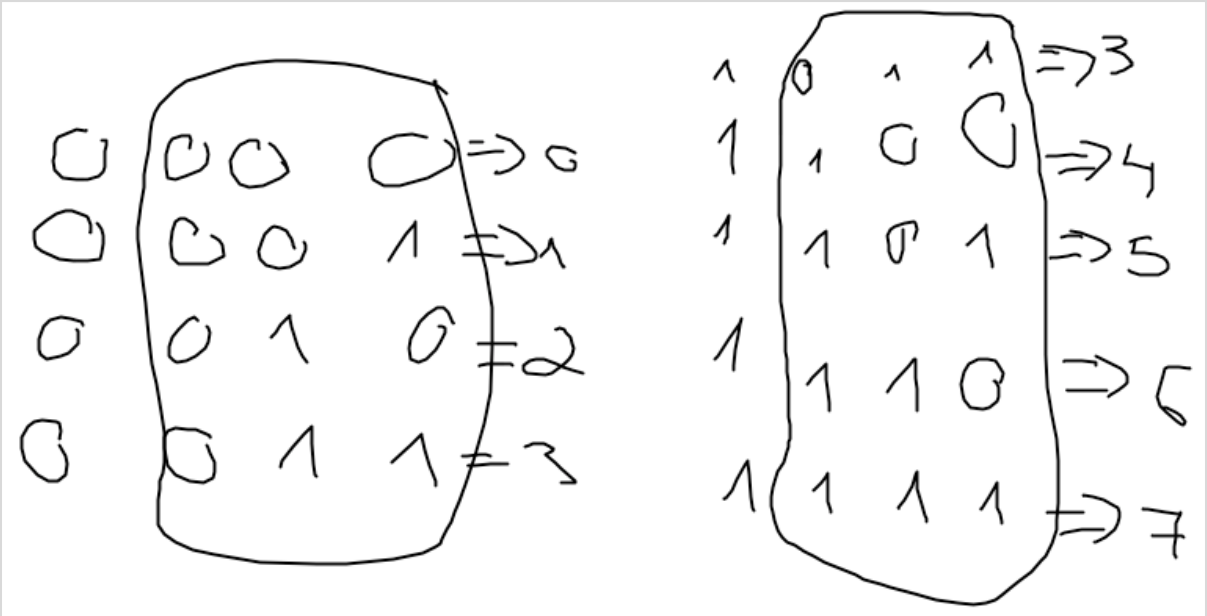
#uni/semester3/Betriebssysteme/chapter16/simulator-fragen

1. First let's use a tiny address space to translate some addresses. Here's a simple set of parameters with a few different random seeds; can you translate the addresses?
   ➜ Als erstes muss man sich die size des address spaces anschauen. Diese sagt wie viel Bits die virtuelle Adresse hat!



   ➜ Ich glaube hier liegt die maximale size eines Segments bei 8 Byte, wenn wir nur zwei Segments haben!
   ➜ Hat man in diesem Beispiel zwei Segmente, einer der positiv grows mit limit 4 Byte. Und das andere, dass negativ grows und mit limit 5 Byte. Für das erste Segment, sind die ersten 4 virtuellen Adressen gültig, also VA 0,1,2,3. Beim zweiten sind die letzten 5 Bytes, also VA 11, 12, 13, 14, 15



JETZT zu den eigentlichen Aufgaben!

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

Address space: 128 Byte ➜ 1000000 ➜ 7 Bit virtuelle adresse
Max size segment = 2 hoch 6 = 64

17: 001 0001 ➜ Offset < limit? ➜ erstes bit sagt in welchem segment
➜ 17 < 20
➜ ja, PA (physical Address) = Base + offset = 17
108: 110 1100 ➜ neg Offset = Offset - 2 hoch offset bits
➜ neg Offset = 44 - 64 = -20
➜ abs(neg Offset) <= limit?
➜ 20 <= 20?
➜ Ja, PA = -20 + 512 = 492
97: 110 0001 ➜ neg Offset = 33 - 64 = -31
➜ abs(-31) <= 20?
➜ nein, segmentation fault
32: 010 0000 ➜ 32 < 20?
➜ nein Segmentation fault
63: 011 1111 ➜ 56 < 20?
➜ nein, Segmentation fault

➜ Die Umwandlung von VA auf binary, habe ich mit der Hexa zahl gemacht. Ist einfacher!

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

Adress space: 128 ➜ 7 Bit VA
Max size segment = 2 hoch 6 = 64

0x00000011: 001 0001 ➜ 17 < 20
➜ PA = 0 + 17 = 17
0x0000006c: 110 1100 ➜ 44 - 64 = -20

➜ 20 <= 20

➜ PA = 492

0x00000061: 110 0001 ➜ 33 - 64 = -31

➜ 31 <= 20

➜ segmentation fault

0x00000020: 010 0000 ➜ 32 < 20

➜ segmentation fault

0x0000003f: 011 1111 ➜ 55 < 20

➜ segmentation fault

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

Adress space: 128 ➜ 7 Bit VA

Max size segment = 2 hoch 6 = 64

0x0000007a: 111 1010 ➜ 58 - 64 = -6

➜ 6 <= 20

➜ PA = 506

0x00000079: 111 1001 ➜ 57 - 64 =-7

➜ 7 <= 20

➜ PA = 512 - 7 = 505

0x00000007: 000 0111 ➜ 7 < 20

➜ PA = 0 + 7 = 7

0x0000000a: 000 1010 ➜ 10 < 20

➜ PA = 0 + 10 = 10

0x0000006a: 110 1010 ➜ 42 - 64 = -22

➜ 22 <= 20

➜ segmentation fault

2. Now, let's see if we understand this tiny address space we've constructed (using the parameters from the question above). What is the highest legal virtual address in segment 0? What about the lowest legal virtual address in segment 1? What are the lowest and highest illegal addresses in this entire address space? Finally, how would you run segmentation.py with the -A flag to test if you are right?

   What is the highest legal virtual address in segment 0?

➜ Wir haben ja ein address space von 128, also hat eine Virtuelle Adresse 7 Bits

➜ Erste bit sagt, ob es sich um segment 0 oder segment 1 handelt.

➜ Man muss auch limit beachten, also Antwort ist VA: 19 (also limit - 1)

What about the lowest legal virtual address in segment 1?

➜ 111 1111 ➜ letzte Adresse vom Adressbereich, aber erste Adresse vom segment 1

➜ Wir wissen, dass abs(neg offeset) <= 20 sein muss.

➜ neg offset = -20 ist die letzte Adresse vom segment 1

➜ neg offset = offset - max segment size

➜ -20 = offset - 64

➜ offset = 44 (101100)

➜ VA = 110 1100 ➜ VA: 108 (Also address space size - limit)

What are the lowest and highest illegal addresses in this entire address space?

➜ Lowest ist 19 + 1 = 20

➜ Highest ist 108 - 1 = 107

Finally, how would you run segmentation.py with the -A flag to test if you are right?

```
./segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -A
17,18,19,20,21,106,107,108,109,110,111 -c
```

```
vladb@VladB ~/G/h/S/B/H/HW16-Segmentation (master)>
./segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -A 17,18,19,20,21,106,107,108,109,
110,111 -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: 0x00000011 (decimal:   17) --> VALID in SEG0: 0x00000011 (decimal:   17)
  VA  1: 0x00000012 (decimal:   18) --> VALID in SEG0: 0x00000012 (decimal:   18)
  VA  2: 0x00000013 (decimal:   19) --> VALID in SEG0: 0x00000013 (decimal:   19)
  VA  3: 0x00000014 (decimal:   20) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x00000015 (decimal:   21) --> SEGMENTATION VIOLATION (SEG0)
  VA  5: 0x0000006a (decimal:  106) --> SEGMENTATION VIOLATION (SEG1)
  VA  6: 0x0000006b (decimal:  107) --> SEGMENTATION VIOLATION (SEG1)
  VA  7: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492)
  VA  8: 0x0000006d (decimal:  109) --> VALID in SEG1: 0x000001ed (decimal:  493)
  VA  9: 0x0000006e (decimal:  110) --> VALID in SEG1: 0x000001ee (decimal:  494)
  VA 10: 0x0000006f (decimal:  111) --> VALID in SEG1: 0x000001ef (decimal:  495)
```

3. Let's say we have a tiny 16-byte address space in a 128-byte physical memory. What base

and bounds would you set up so as to get the simulator to generate the following translation results for the specified address stream: valid, valid, violation, ..., violation, valid, valid? Assume the following parameters:

```
vladb@VladB ~/G/h/S/B/H/HW16-Segmentation (master) [1]>
./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)
```

4. Assume we want to generate a problem where roughly 90% of the randomly-generated virtual addresses are valid (not segmentation violations). How should you configure the simulator to do so? Which parameters are important to getting this outcome?
   ➜ Bei 16 Byte haven wir 16 Adressen. 10 % davon wären 2 Adressen. Also eine Konfiguration, wo 14 Adressen valid sind, zum Beispiel 7 im segment 0 und 7 in segment 1. Vorsicht, wenn man über die Mitte geht nicht, weil dann wurde das Bit segment nicht mehr passen.

```
./segmentation.py -a 16 -p 128 -b 0 -l 8 -B 16 -L 6 -c
```

```
vladb@VladB ~/G/h/S/B/H/HW16-Segmentation (master)>
./segmentation.py -a 16 -p 128 -b 0 -l 8 -B 16 -L 6 -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                  : 8

  Segment 1 base  (grows negative) : 0x00000010 (decimal 16)
  Segment 1 limit                  : 6

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

```
vladb@VladB ~/G/h/S/B/H/HW16-Segmentation (master)>
./segmentation.py -a 64 -p 128 -b 0 -l 29 -B 64 -L 29 -c
ARG seed 0
ARG address space size 64
ARG phys mem size 128

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 29

  Segment 1 base  (grows negative) : 0x00000040 (decimal 64)
  Segment 1 limit                  : 29

Virtual Address Trace
  VA  0: 0x00000036 (decimal:   54) --> VALID in SEG1: 0x00000036 (decimal:   54)
  VA  1: 0x00000030 (decimal:   48) --> VALID in SEG1: 0x00000030 (decimal:   48)
  VA  2: 0x0000001a (decimal:   26) --> VALID in SEG0: 0x0000001a (decimal:   26)
  VA  3: 0x00000010 (decimal:   16) --> VALID in SEG0: 0x00000010 (decimal:   16)
  VA  4: 0x00000020 (decimal:   32) --> SEGMENTATION VIOLATION (SEG1)
```

5. Can you run the simulator such that no virtual addresses are valid? How?

   ➜ Beide limits auf 0

```
./segmentation.py  -a 128 -p 512 -b 0 -l 0 -B 512 -L 0 -c
```

```
./segmentation.py  -a 64 -p 512 -b 63 -l 0 -B 64 -L 1 -c -s 8
```

➜ wenn man das Verhältnis nicht beachtet. Wahrscheinlich liegt es daran, dass dass es schaut sich das erste Bit an, und dieses Bit entscheidet was berechnet und vergliechen wird.

```
./segmentation.py  -a 64 -p 512 -b 0 -l 3 -B 10 -L 2 -c -s 3
```