Ch15

1. Run with seeds 1, 2, and 3, and compute whether each virtual address generated by the process is in or out of bounds. If in bounds, compute the translation.

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: 261 + 13884 = 14145)
  VA 2: 0x000001fb (decimal: 507) --> SEGMENTATION VIOLATION
  VA 3: 0x000001cc (decimal: 460) --> SEGMENTATION VIOLATION
  VA 4: 0x0000029b (decimal: 667) --> SEGMENTATION VIOLATION

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: 57 + 15529 = 15586)
  VA 1: 0x00000056 (decimal: 86) --> VALID: 0x00003cff (decimal: 86 + 15529 = 15615)
  VA 2: 0x00000357 (decimal: 855) --> SEGMENTATION VIOLATION
  VA 3: 0x000002f1 (decimal: 753) --> SEGMENTATION VIOLATION
  VA 4: 0x000002ad (decimal: 685) --> SEGMENTATION VIOLATION

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: 8916 + 67 = 8983)
  VA 4: 0x0000000d (decimal: 13) --> VALID: 0x000022e1 (decimal: 8916 + 13 = 8929)

2. Run with these flags: -s 0 -n 10. What value do you have set -l (the bounds register) to in order to ensure that all the generated virtual addresses are within bounds?

1e18@Zuojuns-MacBook-Air vm-mechanism % ./relocation.py -s 0 -n 10 -l 930 -c

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)
The value of the bounds register should be 930 at least.

3. Run with these flags: -s 1 -n 10 -l 100. What is the maximum value that base can be set to, such that the address space still fits into physical memory in its entirety?
16k - 100 = 16 * 1024b - 100 = 16248
The maximum value should be 16248
./relocation.py -s 1 -n 10 -l 100 -b 16284 -c

ARG seed 1
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: 0x00000089 (decimal: 137) --> SEGMENTATION VIOLATION
  VA 1: 0x00000363 (decimal: 867) --> SEGMENTATION VIOLATION
  VA 2: 0x0000030e (decimal: 782) --> SEGMENTATION VIOLATION
  VA 3: 0x00000105 (decimal: 261) --> SEGMENTATION VIOLATION
  VA 4: 0x000001fb (decimal: 507) --> SEGMENTATION VIOLATION
  VA 5: 0x000001cc (decimal: 460) --> SEGMENTATION VIOLATION
  VA 6: 0x0000029b (decimal: 667) --> SEGMENTATION VIOLATION
  VA 7: 0x00000327 (decimal: 807) --> SEGMENTATION VIOLATION
  VA 8: 0x00000060 (decimal: 96) --> VALID: 0x00003ffc (decimal: 16380)
  VA 9: 0x0000001d (decimal: 29) --> VALID: 0x00003fb9 (decimal: 16313)

4. Run some of the same problems above, but with larger address spaces (-a) and physical memories (-p).
1e18@Zuojuns-MacBook-Air vm-mechanism % ./relocation.py -s 1 -n 10 -l 100 -b 16284 -p 64k -a 2k -c

ARG seed 1
ARG address space size 2k
ARG phys mem size 64k

Base-and-Bounds register information:

  Base   : 0x00003f9c (decimal 16284)
  Limit : 100

Virtual Address Trace
  VA 0: 0x00000113 (decimal: 275) --> SEGMENTATION VIOLATION
  VA 1: 0x000006c7 (decimal: 1735) --> SEGMENTATION VIOLATION
  VA 2: 0x0000061c (decimal: 1564) --> SEGMENTATION VIOLATION
  VA 3: 0x0000020a (decimal: 522) --> SEGMENTATION VIOLATION
  VA 4: 0x000003f6 (decimal: 1014) --> SEGMENTATION VIOLATION
  VA 5: 0x00000398 (decimal: 920) --> SEGMENTATION VIOLATION
  VA 6: 0x00000536 (decimal: 1334) --> SEGMENTATION VIOLATION
  VA 7: 0x0000064f (decimal: 1615) --> SEGMENTATION VIOLATION
  VA 8: 0x000000c0 (decimal: 192) --> SEGMENTATION VIOLATION
  VA 9: 0x0000003a (decimal: 58) --> VALID: 0x00003fd6 (decimal: 16342)

1e18@Zuojuns-MacBook-Air vm-mechanism % ./relocation.py -s 1 -n 10 -l 100 -b 16284 -p 64k -a 3k -c

ARG seed 1
ARG address space size 3k
ARG phys mem size 64k

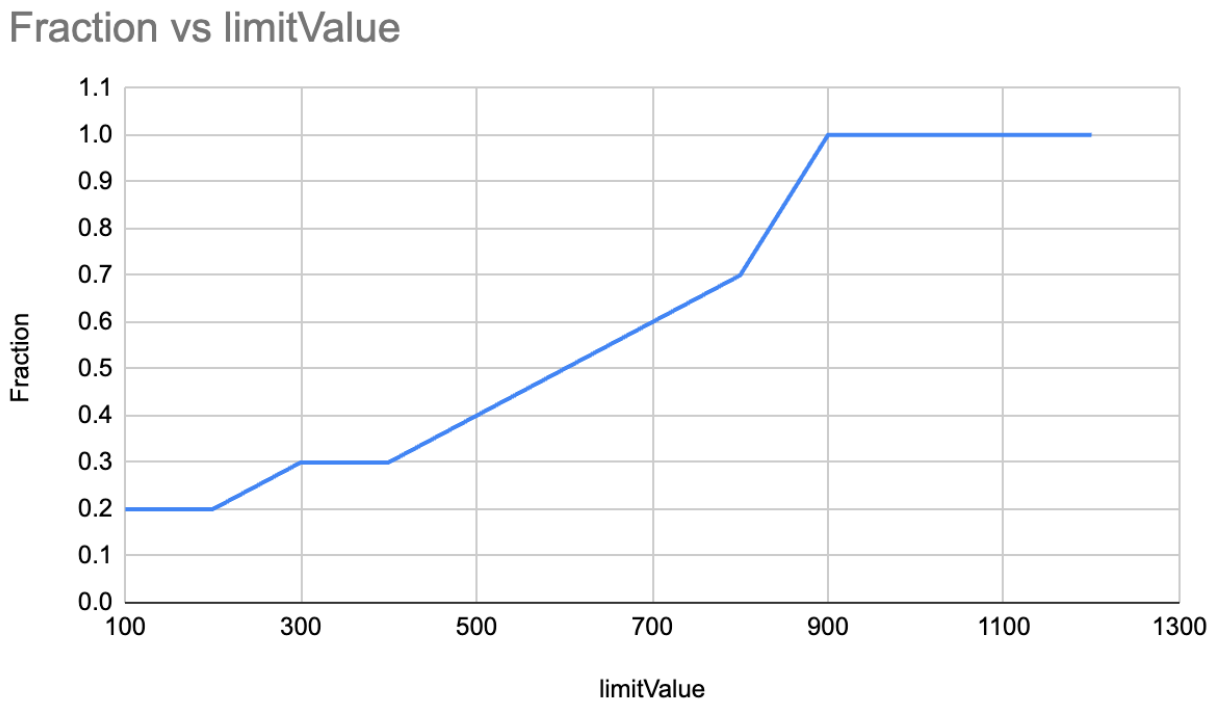
Base-and-Bounds register information:

  Base   : 0x00003f9c (decimal 16284)
  Limit : 100

Virtual Address Trace
  VA 0: 0x0000019c (decimal: 412) --> SEGMENTATION VIOLATION
  VA 1: 0x00000a2b (decimal: 2603) --> SEGMENTATION VIOLATION
  VA 2: 0x0000092a (decimal: 2346) --> SEGMENTATION VIOLATION
  VA 3: 0x0000030f (decimal: 783) --> SEGMENTATION VIOLATION
  VA 4: 0x000005f1 (decimal: 1521) --> SEGMENTATION VIOLATION
  VA 5: 0x00000564 (decimal: 1380) --> SEGMENTATION VIOLATION
  VA 6: 0x000007d1 (decimal: 2001) --> SEGMENTATION VIOLATION
  VA 7: 0x00000976 (decimal: 2422) --> SEGMENTATION VIOLATION
  VA 8: 0x00000120 (decimal: 288) --> SEGMENTATION VIOLATION
  VA 9: 0x00000057 (decimal: 87) --> VALID: 0x00003ff3 (decimal: 16371)

5. What fraction of randomly-generated virtual addresses are valid, as a function of the value of the bounds register? Make a graph from running with different random seeds, with limit values ranging from 0 up to the maximum size of the address space.
./relocation.py -s 1 -n 10 -l 100 -b 15184 -c
Test from limit value is 100 to 1200

limitValue	100	200	300	400	500	600	700	800	900	1000	1200
Fraction	0.2	0.2	0.3	0.3	0.4	0.5	0.6	0.7	1	1	1



Ch16

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?

1e18@Zuojuns-MacBook-Air vm-segmentation % ./segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0 -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: 0x0000006c (decimal: 108) --> VALID in SEG1: 0x000001ec (decimal: 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)


1e18@Zuojuns-MacBook-Air vm-segmentation % ./segmentation.py -a 128 -p 512 -b 0 -l 20 -B
512 -L 20 -s 1 -c
ARG seed 1
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: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  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)

1e18@Zuojuns-MacBook-Air vm-segmentation %
1e18@Zuojuns-MacBook-Air vm-segmentation % ./segmentation.py -a 128 -p 512 -b 0 -l 20 -B
512 -L 20 -s 3 -c
ARG seed 3
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: 0x0000001e (decimal:   30) --> SEGMENTATION VIOLATION (SEG0)
  VA  1: 0x00000045 (decimal:   69) --> SEGMENTATION VIOLATION (SEG1)
```

VA 2: 0x0000002f (decimal: 47) --> SEGMENTATION VIOLATION (SEG0)
VA 3: 0x0000004d (decimal: 77) --> SEGMENTATION VIOLATION (SEG1)
VA 4: 0x00000050 (decimal: 80) --> SEGMENTATION VIOLATION (SEG1)

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?

The hightest legal virtual address in segment 0 is 19.
The lowest legal virtual address in segment 1 is 128 - 20 =108.
The lowest and highest illegal addresses in this virtual address space are 20, and 107.
The lowest and highest illegal addresses in this physical address space are 20, and 491.

./segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2 -A 0,19,20,107,108,127 -c

3. Let's saywe 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: segmentation.py -a 16 -p 128 -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 --b0 ? --l0 ? --b1 ? --l1 ?

1e18@Zuojuns-MacBook-Air vm-segmentation % ./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)

-b BASE0, --b0=BASE0  value of segment 0 base register
-l LEN0, --l0=LEN0    value of segment 0 limit register
-B BASE1, --b1=BASE1  value of segment 1 base register
-L LEN1, --l1=LEN1    value of segment 1 limit register
```

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?
(-l + -L) should be 90% * -a
./segmentation.py -a 100 -p 512 -b 0 -l 45 -B 512 -L 45 -n 20 -c

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

./segmentation.py -a 100 -p 512 -b 0 -l 0 -B 512 -L 0 -n 20 -c
The limits of two registers are 0.

Ch17:
1. First run with the flags -n 10 -H 0 -p BEST -s 0 to generate a few random allocations and frees. Can you predict what alloc()/ free() will return? Can you guess the state of the free list after each request? What do you notice about the free list over time?

Without coalescing, the fragments in the free list become more and more over time.

```
1e18@Zuojuns-MacBook-Air vm-freespace % ./malloc.py -n 10 -H 0 -p BEST -s 0 -c
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
```

listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute True

ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [Size 1]: [addr:1003 sz:97]

Free(ptr[0])
returned 0
Free List [Size 2]: [addr:1000 sz:3][addr:1003 sz:97]

ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [Size 2]: [addr:1000 sz:3][addr:1008 sz:92]

Free(ptr[1])
returned 0
Free List [Size 3]: [addr:1000 sz:3][addr:1003 sz:5][addr:1008 sz:92]

ptr[2] = Alloc(8) returned 1008 (searched 3 elements)
Free List [Size 3]: [addr:1000 sz:3][addr:1003 sz:5][addr:1016 sz:84]

Free(ptr[2])
returned 0
Free List [Size 4]: [addr:1000 sz:3][addr:1003 sz:5][addr:1008 sz:8][addr:1016 sz:84]

ptr[3] = Alloc(8) returned 1008 (searched 4 elements)
Free List [Size 3]: [addr:1000 sz:3][addr:1003 sz:5][addr:1016 sz:84]

Free(ptr[3])
returned 0
Free List [Size 4]: [addr:1000 sz:3][addr:1003 sz:5][addr:1008 sz:8][addr:1016 sz:84]

ptr[4] = Alloc(2) returned 1000 (searched 4 elements)
Free List [Size 4]: [addr:1002 sz:1][addr:1003 sz:5][addr:1008 sz:8][addr:1016 sz:84]

ptr[5] = Alloc(7) returned 1008 (searched 4 elements)
Free List [Size 4]: [addr:1002 sz:1][addr:1003 sz:5][addr:1015 sz:1][addr:1016 sz:84]

2. How are the results different when using a WORST fit policy to search the free list (-p WORST)?What changes?

Always find the largest chunk in the free list, it's easily to create more fragments in the free list.

```
1e18@Zuojuns-MacBook-Air vm-freespace % ./malloc.py -n 10 -H 0 -p WORST -s 0 -c
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy WORST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute True

ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1003 sz:97 ]

Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1003 sz:97 ]

ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1008 sz:92 ]

Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:92 ]

ptr[2] = Alloc(8) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]

Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[3] = Alloc(8) returned 1016 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1024 sz:76 ]

Free(ptr[3])
returned 0
```

Free List [Size 5]: [addr:1000 sz:3][addr:1003 sz:5][addr:1008 sz:8][addr:1016 sz:8][addr:1024 sz:76]

ptr[4] = Alloc(2) returned 1024 (searched 5 elements)
Free List [Size 5]: [addr:1000 sz:3][addr:1003 sz:5][addr:1008 sz:8][addr:1016 sz:8][addr:1026 sz:74]

ptr[5] = Alloc(7) returned 1026 (searched 5 elements)
Free List [Size 5]: [addr:1000 sz:3][addr:1003 sz:5][addr:1008 sz:8][addr:1016 sz:8][addr:1033 sz:67]

3. What about when using FIRST fit (-p FIRST)? What speeds up when you use first fit?
The number of free spaces are less than before. And it searched less elements.

1e18@Zuojuns-MacBook-Air vm-freespace % ./malloc.py -n 10 -H 0 -p FIRST -s 0 -c
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy FIRST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute True

ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [Size 1]: [addr:1003 sz:97]

Free(ptr[0])
returned 0
Free List [Size 2]: [addr:1000 sz:3][addr:1003 sz:97]

ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [Size 2]: [addr:1000 sz:3][addr:1008 sz:92]

Free(ptr[1])
returned 0
Free List [Size 3]: [addr:1000 sz:3][addr:1003 sz:5][addr:1008 sz:92]

ptr[2] = Alloc(8) returned 1008 (searched 3 elements)
Free List [Size 3]: [addr:1000 sz:3][addr:1003 sz:5][addr:1016 sz:84]

Free(ptr[2])
returned 0
Free List [Size 4]: [addr:1000 sz:3][addr:1003 sz:5][addr:1008 sz:8][addr:1016 sz:84]

ptr[3] = Alloc(8) returned 1008 (searched 3 elements)
Free List [Size 3]: [addr:1000 sz:3][addr:1003 sz:5][addr:1016 sz:84]

Free(ptr[3])
returned 0
Free List [Size 4]: [addr:1000 sz:3][addr:1003 sz:5][addr:1008 sz:8][addr:1016 sz:84]

ptr[4] = Alloc(2) returned 1000 (searched 1 elements)
Free List [Size 4]: [addr:1002 sz:1][addr:1003 sz:5][addr:1008 sz:8][addr:1016 sz:84]

ptr[5] = Alloc(7) returned 1008 (searched 3 elements)
Free List [Size 4]: [addr:1002 sz:1][addr:1003 sz:5][addr:1015 sz:1][addr:1016 sz:84]

4. For the above questions, how the list is kept ordered can affect the time it takes to find a free location for some of the policies. Use the different free list orderings (-l ADDRSORT, -l SIZESORT+, -l SIZESORT-) to see how the policies and the list orderings interact.

./malloc.py -n 10 -H 0 -p FIRST -l SIZESORT+ -s 0 -c

The first sort has better performance with size sort. Because it always use the first fit chunk, the when the free chunks sort by order, the task always can find the most fit chunk in a faster speed.
When first sort uses SIZESORT-, it would have the same result as worst fit, but with better time efficiency.

For best fit and worst fit, there is not much difference with size sort. Because they still need traverse all of the free list.

5. Coalescing of a free list can be quite important. Increase the number of random allocations (say to -n 1000). What happens to larger allocation requests over time? Run with and without coalescing (i.e., without and with the -C flag). What differences in outcome do you see? How big is the free list over time in each case? Does the ordering of the list matter in this case?

./malloc.py -n 1000 -r 30 -c
ptr[736] = Alloc(26) returned -1 (searched 31 elements)

Free List [Size 31]: [addr:1000 sz:4][addr:1004 sz:4][addr:1016 sz:2][addr:1018 sz:5][addr:1023 sz:9][addr:1032 sz:1][addr:1033 sz:1][addr:1034 sz:1][addr:1035 sz:5][addr:1040 sz:1][addr:1041 sz:1][addr:1042 sz:3][addr:1045 sz:1][addr:1046 sz:10][addr:1056 sz:1][addr:1057 sz:1][addr:1058 sz:1][addr:1059 sz:2][addr:1061 sz:1][addr:1062 sz:2][addr:1064 sz:7][addr:1071 sz:1][addr:1072 sz:10][addr:1082 sz:1][addr:1083 sz:1][addr:1084 sz:1][addr:1085 sz:1][addr:1086 sz:4][addr:1090 sz:6][addr:1096 sz:1][addr:1097 sz:3]

./malloc.py -n 1000 -r 30 -c -C
Free(ptr[540])
returned 0
Free List [Size 1]: [addr:1000 sz:100]

With coalescing, the free list in the end is much less fragmented than the situation without -C. The size of free list with coalescing is 1, and the size of free list without coalescing is 31. For first fit, SIZESORT+ would help increase performance.

6. What happens when you change the percent allocated fraction -P to higher than 50? What happens to allocations as it nears 100? What about as the percent nears 0?
./malloc.py -c -n 1000 -P 50
ptr[590] = Alloc(9) returned -1 (searched 31 elements)
Free List [Size 31]: [addr:1000 sz:2][addr:1002 sz:1][addr:1006 sz:1][addr:1007 sz:1][addr:1008 sz:5][addr:1013 sz:1][addr:1014 sz:1][addr:1015 sz:1][addr:1016 sz:5][addr:1021 sz:1][addr:1022 sz:3][addr:1031 sz:1][addr:1032 sz:2][addr:1034 sz:3][addr:1037 sz:4][addr:1041 sz:1][addr:1042 sz:2][addr:1052 sz:1][addr:1053 sz:6][addr:1059 sz:2][addr:1061 sz:1][addr:1068 sz:1][addr:1069 sz:3][addr:1072 sz:5][addr:1077 sz:3][addr:1080 sz:1][addr:1081 sz:5][addr:1086 sz:3][addr:1089 sz:5][addr:1094 sz:2][addr:1096 sz:4]

./malloc.py -c -n 1000 -P 100
All of space are full, there is no space in the free list anymore. The heap runs out of memory, returning -1,

./malloc.py -c -n 1000 -P 0
Nothing allocated.

7. What kind of specific requests can you make to generate a highlyfragmented free space? Use the -A flag to create fragmented free lists, and see how different policies and options change the organization of the free list.

./malloc.py -c -A +3,-3,+5,-5,+8,-8,+3,-3,+1,-1
./malloc.py -c -A +3,-3,+5,-5,+8,-8,+3,-3,+1,-1 -C

./malloc.py -c -A +3,-3,+5,-5,+8,-8,+3,-3,+1,-1 -l SIZESORT-
./malloc.py -c -A +3,-3,+5,-5,+8,-8,+3,-3,+1,-1 -l SIZESORT- -C
./malloc.py -c -A +3,-3,+5,-5,+8,-8,+3,-3,+1,-1 -p FIRST -l SIZESORT+
./malloc.py -c -A +3,-3,+5,-5,+8,-8,+3,-3,+1,-1 -p FIRST -l SIZESORT+ -C
./malloc.py -c -A +3,-3,+5,-5,+8,-8,+3,-3,+1,-1-p FIRST -l SIZESORT-
./malloc.py -c -A +3,-3,+5,-5,+8,-8,+3,-3,+1,-1 -p FIRST -l SIZESORT- -C
./malloc.py -c -A +3,-3,+5,-5,+8,-8,+3,-3,+1,-1 p WORST -l SIZESORT+
./malloc.py -c -A +3,-3,+5,-5,+8,-8,+3,-3,+1,-1 p WORST -l SIZESORT+ -C
./malloc.py -c -A +3,-3,+5,-5,+8,-8,+3,-3,+1,-1 p WORST -l SIZESORT-
./malloc.py -c -A +3,-3,+5,-5,+8,-8,+3,-3,+1,-1 p WORST -l SIZESORT- -C
./malloc.py -c -A +3,-3,+5,-5,+8,-8,+3,-3,+1,-1 p BEST -l SIZESORT+
./malloc.py -c -A +3,-3,+5,-5,+8,-8,+3,-3,+1,-1 p BEST -l SIZESORT+ -C
./malloc.py -c -A +3,-3,+5,-5,+8,-8,+3,-3,+1,-1 p BEST -l SIZESORT-
./malloc.py -c -A +3,-3,+5,-5,+8,-8,+3,-3,+1,-1 p BEST -l SIZESORT- -C

The first sort has better performance with size sort. Because it always use the first fit chunk, the when the free chunks sort by order, the task always can find the most fit chunk in a faster speed.
When first sort uses SIZESORT-, it would have the same result as worst fit, but with better time efficiency.

For best fit and worst fit, there is not much difference with size sort. Because they still need traverse all of the free list.