This homework covers OSTEP chapters 14-16. It is due in class Friday, February 27. Hand in a hardcopy of your answers in class.

While you may discuss ideas and strategies for problems with other students, you should always make the first attempt on a problem yourself and **you must write up your own solutions in your own words**. *You may not collaboratively write solutions or copy a solution that one person in the group writes up. You also may not look for, copy, or use solutions from other sources, including from generative AI like ChatGPT, even if you make changes. There's no such thing as using someone else's solution for a problem "as an example" for writing your own.*

Preliminaries

- Copy the directory `hw4` (and its contents) from from `/classes/cs331` to your `~/cs331/workspace` directory. (You won't be writing any code for this part, but you will need to compile and run the provided programs so they might as well go into your `workspace` directory.)
- Copy the directories `vm-mechanism` and `vm-segmentation` (and their contents) from `/classes/cs331` to your `~/cs331` directory. (For these parts you won't be writing any code, so the copied directories don't need to go into your `workspace` directory.)
- The `vm-mechanism` and `vm-segmentation` directories each contain a python program (`relocation.py` and `segmentation.py`, respectively) and a `README` with instructions for using the program. You can also find the text of the `README`s in the appendix sections A and B at the end of this document.

Exercises

Chapter 14 homework problems (at the very end of chapter 14) —

- There is no need to write code for any of these problems — find the programs referred to in the `hw4` directory that you copied. A `Makefile` has also been provided. You should look at each program and make sure you understand the code, however.
- Also explain your answers — what's going on? How does what happened or the output produced relate to the problem? (Don't just report the output or result.)
- Tools: `valgrind` and `gdb` are not available on `csmath-terminal.hws.edu` — connect to the VDI or boot Linux in Demarest 002.

1. For #1, compile `null.c` using the command `make null`.

2. For #2, just run `gdb` as directed and answer the question posed — `make null` already compiled the program with the `-g` flag. (You might have noticed that in the command line echoed by `make`.) Press `ctrl-d` or type `exit` to exit `gdb` when you are done.

3. Do #3.

4. Do #4–7: the programs are `malloc.c`, `bounds.c`, `free.c`, and `badfree.c`, respectively. Compile them individually with `make malloc`, `make bounds`, etc.

Chapter 15 homework problems (at the very end of chapter 15) —

5. Do #1 for practice — make sure you understand the concepts (the role of the base and limit (bounds) registers in address translation) and how to run and use `relocation.py`, but you don't need to hand in your solutions. (Remember that you can check your work by running `relocation.py` with the `-c` option.)

6. Do #2–4. Note that the `-l` flag mentioned involves a lowercase letter l, not the number 1. For #4, repeat #2 and #3 with the following three combinations: address space 2k (and default physical memory), physical memory size 32k (and default address space), and address space 4k with physical memory size 64k.

Chapter 16 homework problems (at the very end of chapter 16) —

7. Do #1 for practice — make sure you understand the concepts (address translation with segmentation, and the specific layout of the address space used by the simulator) and how to run and use `segmentation.py`, but you don't need to hand in your solutions. (Remember that you can check your work by running `segmentation.py` with the `-c` option.)

8. Do #2. Also identify the lowest and highest addresses in the whole address space (in addition to the four addresses in the original question) and include them in your answer about the `-A` flag setting,

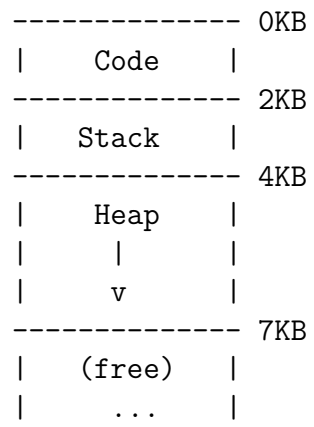
9. Do #3.

10. Do #4–5, giving the command line for the behavior described. Also explain your answers — identify which parameters are important for the outcome (as asked in #4) and explain why.

A relocation.py

This program allows you to see how address translations are performed in a system with base and bounds registers. As before, there are two steps to running the program to test out your understanding of base and bounds. First, run without the -c flag to generate a set of translations and see if you can correctly perform the address translations yourself. Then, when done, run with the -c flag to check your answers.

In this homework, we will assume a slightly different address space than our canonical one with a heap and stack at opposite ends of the space. Rather, we will assume that the address space has a code section, then a fixed-sized (small) stack, and a heap that grows downward right after, looking something like you see in the Figure below. In this configuration, there is only one direction of growth, towards higher regions of the address space.



In the figure, the bounds register would be set to 7˜KB, as that represents the end of the address space. References to any address within the bounds would be considered legal; references above this value are out of bounds and thus the hardware would raise an exception.

To run with the default flags, type relocation.py at the command line. The result should be something like this:

```
prompt> ./relocation.py
...
Base-and-Bounds register information:

  Base   : 0x00003082 (decimal 12418)
  Limit  : 472

Virtual Address Trace
  VA  0: 0x01ae (decimal:430) -> PA or violation?
  VA  1: 0x0109 (decimal:265) -> PA or violation?
  VA  2: 0x020b (decimal:523) -> PA or violation?
```

```
  VA  3: 0x019e (decimal:414) -> PA or violation?
  VA  4: 0x0322 (decimal:802) -> PA or 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.

As you can see, the homework simply generates randomized virtual addresses. For each, you should determine whether it is in bounds, and if so, determine to which physical address it translates. Running with -c (the "compute this for me" flag) gives us the results of these translations, i.e., whether they are valid or not, and if valid, the resulting physical addresses. For convenience, all numbers are given both in hex and decimal.

```
prompt> ./relocation.py -c
...
Virtual Address Trace
  VA  0: 0x01ae (decimal:430) -> VALID: 0x00003230 (dec:12848)
  VA  1: 0x0109 (decimal:265) -> VALID: 0x0000318b (dec:12683)
  VA  2: 0x020b (decimal:523) -> SEGMENTATION VIOLATION
  VA  3: 0x019e (decimal:414) -> VALID: 0x00003220 (dec:12832)
  VA  4: 0x0322 (decimal:802) -> SEGMENTATION VIOLATION
```

With a base address of 12418 (decimal), address 430 is within bounds (i.e., it is less than the limit register of 472) and thus translates to 430 added to 12418 or 12848. A few of the addresses shown above are out of bounds (523, 802), as they are in excess of the bounds. Pretty simple, no? Indeed, that is one of the beauties of base and bounds: it's so darn simple!

There are a few flags you can use to control what's going on better:

```
prompt> ./relocation.py -h
Usage: relocation.py [options]

Options:
  -h, --help            show this help message and exit
  -s SEED, --seed=SEED  the random seed
  -a ASIZE, --asize=ASIZE address space size (e.g., 16, 64k, 32m)
  -p PSIZE, --physmem=PSIZE physical memory size (e.g., 16, 64k)
  -n NUM, --addresses=NUM # of virtual addresses to generate
  -b BASE, --b=BASE     value of base register
  -l LIMIT, --l=LIMIT   value of limit register
  -c, --compute         compute answers for me
```

In particular, you can control the virtual address-space size (-a), the size of physical memory (-p), the number of virtual addresses to generate (-n), and the values of the base and bounds registers for this process (-b and -l, respectively).

B segmentation.py

This program allows you to see how address translations are performed in a system with segmentation. The segmentation that this system uses is pretty simple: an address space has just *two* segments; further, the top bit of the virtual address generated by the process determines which segment the address is in: 0 for segment 0 (where, say, code and the heap would reside) and 1 for segment 1 (where the stack lives). Segment 0 grows in a positive direction (towards higher addresses), whereas segment 1 grows in the negative direction.

Visually, the address space looks like this:

```
 --------------- virtual address 0
 |    seg0     |
 |             |
 |             |
 |-------------|
 |             |
 |             |
 |             |
 |             |
 |             |
 |(unallocated)|
 |             |
 |             |
 |             |
 |-------------|
 |             |
 |    seg1     |
 |-------------| virtual address max (size of address space)
```

With segmentation, as you might recall, there is a base/limit pair of registers per segment. Thus, in this problem, there are two base/limit pairs. The segment-0 base tells which physical address the *top* of segment 0 has been placed in physical memory and the limit tells how big the segment is; the segment-1 base tells where the *bottom* of segment 1 has been placed in physical memory and the corresponding limit also tells us how big the segment is (or how far it grows in the negative direction).

As before, there are two steps to running the program to test out your understanding of segmentation. First, run without the "-c" flag to generate a set of translations and see if you can correctly perform the address translations yourself. Then, when done, run with the "-c" flag to check your answers.

For example, to run with the default flags, type:

```
prompt> ./segmentation.py
```

or

```
prompt> python ./segmentation.py
```

You should see this:

```
ARG seed 0
ARG address space size 1k
ARG phys mem size 16k

Segment register information:

  Segment 0 base  (grows positive) : 0x00001aea (decimal 6890)
  Segment 0 limit                  : 472

  Segment 1 base  (grows negative) : 0x00001254 (decimal 4692)
  Segment 1 limit                  : 450

Virtual Address Trace
  VA  0: 0x0000020b (decimal:  523) --> PA or segmentation violation?
  VA  1: 0x0000019e (decimal:  414) --> PA or segmentation violation?
  VA  2: 0x00000322 (decimal:  802) --> PA or segmentation violation?
  VA  3: 0x00000136 (decimal:  310) --> PA or segmentation violation?
  VA  4: 0x000001e8 (decimal:  488) --> 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 address space with
two segments: the top bit of the virtual address can thus be used to check
whether the virtual address is in segment 0 (topbit=0) or segment 1
(topbit=1). Note that the base/limit pairs given to you grow in different
directions, depending on the segment, i.e., segment 0 grows in the positive
direction, whereas segment 1 in the negative.
```

Then, after you have computed the translations in the virtual address trace, run the program again with the "-c" flag. You will see the following (not including the redundant information):

```
Virtual Address Trace
  VA  0: 0x0000020b (decimal:  523) --> SEGMENTATION VIOLATION (SEG1)
  VA  1: 0x0000019e (decimal:  414) --> VALID in SEG0: 0x00001c88 (decimal: 7304)
  VA  2: 0x00000322 (decimal:  802) --> VALID in SEG1: 0x00001176 (decimal: 4470)
  VA  3: 0x00000136 (decimal:  310) --> VALID in SEG0: 0x00001c20 (decimal: 7200)
  VA  4: 0x000001e8 (decimal:  488) --> SEGMENTATION VIOLATION (SEG0)
```

As you can see, with -c, the program translates the addresses for you, and hence you can check if you understand how a system using segmentation translates addresses.

Of course, there are some parameters you can use to give yourself different problems. One particularly important parameter is the -s or -seed parameter, which lets you generate different problems by passing in a different random seed. Of course, make sure to use the same random seed when you are generating a problem and then solving it.

There are also some parameters you can use to play with different-sized address spaces and physical memories. For example, to experiment with segmentation in a tiny system, you might type:

```
prompt> ./segmentation.py -s 100 -a 16 -p 32
ARG seed 0
ARG address space size 16
ARG phys mem size 32

Segment register information:

  Segment 0 base  (grows positive) : 0x00000018 (decimal 24)
  Segment 0 limit                  : 4

  Segment 1 base  (grows negative) : 0x00000012 (decimal 18)
  Segment 1 limit                  : 5

Virtual Address Trace
  VA  0: 0x0000000c (decimal:   12) --> PA or segmentation violation?
  VA  1: 0x00000008 (decimal:    8) --> PA or segmentation violation?
  VA  2: 0x00000001 (decimal:    1) --> PA or segmentation violation?
  VA  3: 0x00000007 (decimal:    7) --> PA or segmentation violation?
  VA  4: 0x00000000 (decimal:    0) --> PA or segmentation violation?
```

which tells the program to generate virtual addresses for a 16-byte address space placed somewhere in a 32-byte physical memory. As you can see, the resulting virtual addresses are tiny (12, 8, 1, 7, and 0). As you can also see, the program picks tiny base register and limit values, as appropriate. Run with -c to see the answers.

This example should also show you exactly what each base pair means. For example, segment 0's base is set to a physical address of 24 (decimal) and is of size 4 bytes. Thus, *virtual* addresses 0, 1, 2, and 3 are in segment 0 and valid, and map to physical addresses 24, 25, 26, and 27, respectively.

Slightly more tricky is the negative-direction-growing segment 1. In the tiny example above, segment 1's base register is set to physical address 18, with a size of 5 bytes. That means that the *last* five bytes of the virtual address space, in this case 11, 12, 13, 14, and 15, are valid virtual addresses, and that they map to physical addresses 13, 14, 15, 16, and 17, respectively.

If that doesn't make sense, read it again – you will have to make sense of how this works in order to do any of these problems.

Note you can specify bigger values by tacking a "k", "m", or even "g" onto the values you pass in with the -a or -p flags, as in "kilobytes", "megabytes", and "gigabytes". Thus, if you wanted to do some translations with a 1-MB address space set in a 32-MB physical memory, you might type:

```
prompt> ./segmentation.py -a 1m -p 32m
```

If you want to get even more specific, you can set the base register and limit register values yourself, with the –b0, –l0, –b1, and –l1 registers. Try them and see.

Finally, you can always run

```
prompt> ./segmentation.py -h
```

to get a complete list of flags and options.

Enjoy!