This homework covers OSTEP chapters 18 and 20. It is due in class Monday, March 9. 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 directories `vm-paging` and `vm-smalltables` (and their contents) from `/classes/cs331` to your `~/cs331` directory. (You won't be writing any code, so the copied directories don't need to go into your `workspace` directory.)
- The `vm-paging` and `vm-smalltables` directories each contain a python program (`paging-linear-translate.py` and `paging-multilevel-translate.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 18 homework problems (at the very end of chapter 18) —

1. Look at the `README` to determine what the `-a`, `-P`, and `-u` flags do, then run the simulator with the 11 sets of parameters given in #1–3. What do the `-a`, `-P`, and `-u` flags do, and what happens to the page table as they increase? Why? (Explain what is going on.)
2. Do some address translations for practice: run the simulator with the following parameters and compute the physical address for each of the virtual addresses given (if valid). Make sure you understand how address translations work with a linear page table, but you don't need to hand in your solutions. (Remember that you can use `-c` to check your answers.)
    - `-P 1k -a 16k -p 32k -v -u 50`

3. Do #4: run the simulator with the parameters specified *except* use seed 8 (`-s 8`) instead of the one specified for all three runs.

   - For each set of parameters, what is the configuration (address space size, page size, physical memory size) and how big is the resulting page table?
   - Address the "sometimes quite crazy" comment in the question text — *are* some of these quite crazy? Or are they realistic? Explain.
   - Do some of the address translations for practice. Again make sure you understand how address translations work with a linear page table, but you don't need to hand in your solutions for this part. (Instead use `-c` to check your answers.)

4. Do #5: try out just the scenario suggested (where the address space is bigger than the physical memory). What happens, and why? Explain.

Chapter 20 homework problems (at the very end of chapter 20) —

5. Do #1: answer the questions posed, and explain your answers.

6. Do #2 for practice — make sure you understand the concepts (how a two-level page table works and how to use one to do address translations) but you don't need to hand in your solutions. (Use `-c` to check your answers.)

7. Do #3: answer the questions posed, and explain your answers. Why do you think the behavior will be as you say?

A paging-linear-translate.py

In this homework, you will use a simple program, which is known as paging-linear-translate.py, to see if you understand how simple virtual-to-physical address translation works with linear page tables. To run the program, remember to either type just the name of the program (./paging-linear-translate.py) or possibly this (python paging-linear-translate.py). When you run it with the -h (help) flag, you see:

```
prompt> ./paging-linear-translate.py -h
Usage: paging-linear-translate.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, ...)
-p PSIZE, --physmem=PSIZE
                        physical memory size (e.g., 16, 64k, ...)
-P PAGESIZE, --pagesize=PAGESIZE
                        page size (e.g., 4k, 8k, ...)
-n NUM, --addresses=NUM number of virtual addresses to generate
-u USED, --used=USED    percent of address space that is used
-v                      verbose mode
-c                      compute answers for me
```

First, run the program without any arguments:

```
prompt> ./paging-linear-translate.py
ARG seed 0
ARG address space size 16k
ARG phys mem size 64k
ARG page size 4k
ARG verbose False

The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
   0x8000000c
   0x00000000
```

```
   0x00000000
   0x80000006

Virtual Address Trace
  VA  0: 0x00003229 (decimal:    12841) --> PA or invalid?
  VA  1: 0x00001369 (decimal:     4969) --> PA or invalid?
  VA  2: 0x00001e80 (decimal:     7808) --> PA or invalid?
  VA  3: 0x00002556 (decimal:     9558) --> PA or invalid?
  VA  4: 0x00003a1e (decimal:    14878) --> PA or invalid?
```

For each virtual address, write down the physical address it translates to OR write down that it is an out-of-bounds address (e.g., a segmentation fault).

As you can see, what the program provides for you is a page table for a particular process (remember, in a real system with linear page tables, there is one page table per process; here we just focus on one process, its address space, and thus a single page table). The page table tells you, for each virtual page number (VPN) of the address space, that the virtual page is mapped to a particular physical frame number (PFN) and thus valid, or not valid.

The format of the page-table entry is simple: the left-most (high-order) bit is the valid bit; the remaining bits, if valid is 1, is the PFN.

In the example above, the page table maps VPN 0 to PFN 0xc (decimal 12), VPN 3 to PFN 0x6 (decimal 6), and leaves the other two virtual pages, 1 and 2, as not valid.

Because the page table is a linear array, what is printed above is a replica of what you would see in memory if you looked at the bits yourself. However, it is sometimes easier to use this simulator if you run with the verbose flag (-v); this flag also prints out the VPN (index) into the page table. From the example above, run with the -v flag:

```
Page Table (from entry 0 down to the max size)
  [       0]   0x8000000c
  [       1]   0x00000000
  [       2]   0x00000000
  [       3]   0x80000006
```

Your job, then, is to use this page table to translate the virtual addresses given to you in the trace to physical addresses. Let's look at the first one: VA 0x3229. To translate this virtual address into a physical address, we first have to break it up into its constituent components: a virtual page number and an offset. We do this by noting down the size of the address space and the page size. In this example, the address space is set to 16KB (a very small address space) and the page size is 4KB. Thus, we know that there are 14 bits in the virtual address, and that the offset is 12 bits, leaving 2 bits for the VPN. Thus, with our address 0x3229, which is binary 11 0010 0010 1001, we know the top two bits specify the VPN. Thus, 0x3229 is on virtual page 3 with an offset of 0x229.

We next look in the page table to see if VPN 3 is valid and mapped to some physical frame or invalid, and we see that it is indeed valid (the high bit is 1) and mapped to physical page 6. Thus, we can form our final physical address by taking the physical page 6 and adding it onto the offset, as follows: 0x6000 (the physical page, shifted into the proper spot) OR 0x0229 (the offset), yielding the final physical address: 0x6229. Thus, we can see that virtual address 0x3229 translates to physical address 0x6229 in this example.

To see the rest of the solutions (after you have computed them yourself!), just run with the -c flag (as always):

```
...
VA  0: 00003229 (decimal: 12841) --> 00006229 (25129) [VPN 3]
VA  1: 00001369 (decimal:  4969) --> Invalid (VPN 1 not valid)
VA  2: 00001e80 (decimal:  7808) --> Invalid (VPN 1 not valid)
VA  3: 00002556 (decimal:  9558) --> Invalid (VPN 2 not valid)
VA  4: 00003a1e (decimal: 14878) --> 00006a1e (27166) [VPN 3]
```

Of course, you can change many of these parameters to make more interesting problems. Run the program with the -h flag to see what options there are:

- The -s flag changes the random seed and thus generates different page table values as well as different virtual addresses to translate.
- The -a flag changes the size of the address space.
- The -p flag changes the size of physical memory.
- The -P flag changes the size of a page.
- The -n flag can be used to generate more addresses to translate (instead of the default 5).
- The -u flag changes the fraction of mappings that are valid, from 0% (-u 0) up to 100% (-u 100). The default is 50, which means that roughly 1/2 of the pages in the virtual address space will be valid.
- The -v flag prints out the VPN numbers to make your life easier.

B paging-multilevel-translate.py

This fun little homework tests if you understand how a multi-level page table works. And yes, there is some debate over the use of the term fun in the previous sentence. The program is called: `paging-multilevel-translate.py`

Some basic assumptions:

- The page size is an unrealistically-small 32 bytes
- The virtual address space for the process in question (assume there is only one) is 1024 pages, or 32 KB
- physical memory consists of 128 pages

Thus, a virtual address needs 15 bits (5 for the offset, 10 for the VPN). A physical address requires 12 bits (5 offset, 7 for the PFN).

The system assumes a multi-level page table. Thus, the upper five bits of a virtual address are used to index into a page directory; the page directory entry (PDE), if valid, points to a page of the page table. Each page table page holds 32 page-table entries (PTEs). Each PTE, if valid, holds the desired translation (physical frame number, or PFN) of the virtual page in question.

The format of a PTE is thus:

```
VALID | PFN6 ... PFN0
```

and is thus 8 bits or 1 byte.

The format of a PDE is essentially identical:

```
VALID | PT6 ... PT0
```

You are given two pieces of information to begin with.

First, you are given the value of the page directory base register (PDBR), which tells you which page the page directory is located upon.

Second, you are given a complete dump of each page of memory. A page dump looks like this:

```
page 0: 08 00 01 15 11 1d 1d 1c 01 17 15 14 16 1b 13 0b ...
page 1: 19 05 1e 13 02 16 1e 0c 15 09 06 16 00 19 10 03 ...
page 2: 1d 07 11 1b 12 05 07 1e 09 1a 18 17 16 18 1a 01 ...
...
```

which shows the 32 bytes found on pages 0, 1, 2, and so forth. The first byte (0th byte) on page 0 has the value 0x08, the second is 0x00, the third 0x01, and so forth.

You are then given a list of virtual addresses to translate.

Use the PDBR to find the relevant page table entries for this virtual page. Then find if it is valid. If so, use the translation to form a final physical address. Using this address, you can find the VALUE that the memory reference is looking for.

Of course, the virtual address may not be valid and thus generate a fault.

Some useful options:

```
-s SEED, --seed=SEED       the random seed
-n NUM, --addresses=NUM    number of virtual addresses to generate
-c, --solve                compute answers for me
```

Change the seed to get different problems, as always.

Change the number of virtual addresses generated to do more translations for a given memory dump.

Use -c (or –solve) to show the solutions.

Good luck with this monstrosity!