EECS 338 Homework 9

General requirements:

- See posted due date.
- Create a typed report for discussion questions and program output.
- Upload a single, compressed file (e.g. zip) to Canvas that contains all required files (programs + report).
- All work should be your own, as explained in the Academic Integrity policy from the syllabus.
- Include either a single makefile that compiles all programs or a separate makefile for each.

1. Assume variables have logical addresses with 16-bit page numbers and 16-bit offset using the memory configuration below. (Note that each hexidecimal is 4 bits long.)

Logical Address Format

0xppppdddd

pppp: page number
dddd: page offset

Page Table

Page	Frame	V/I
0	2	v
1	0	i
2	3	v

Physical Memory

Frame	Physical Address (starting)	Size (hex)	Size (dec)
0	0xc000	0x1000	4,096
1	0xd000	0x1000	4,096
2	0xe000	0x1000	4,096
3	0xf000	0x1000	4,096

   a. Suppose a 4-byte variable "x" has the logical address 0x0002000e. What is the physical address? Explain.

   b. Suppose a process has access only to the pages 0, 1, and 2, as shown in the page table. (Remember from section 9.2 of the textbook that the valid/invalid flag only refers to whether a given page is loaded in memory.) Give an example of a logical address that would generate a segmentation fault. Explain.

   c. Suppose the page table is complete. Give an example of a logical address that would generate a page fault. Explain. (Remember from section 9.2 of the textbook that the valid/invalid flag refers to whether a given page is loaded in memory.)

2. The file "page_fault.c" is provided and was adapted from work by Michael Recachinas. As demonstrated in lecture, it allows the user to influence the number of minor page faults (also called page *reclaims* on Mac OS) using a 2-D array where each row requires a full page of memory. This depends on the getpagesize() function and the loops which access all of the elements in the array. The program receives one command line argument which is the desired number of rows in the 2-D array. For reporting purposes, it also prints the system page size and size of a single int. Below is an example of executing it:

```
$ ./page_fault 10
Page size: 4096
Int size: 4
```

   To measure the number of page faults, /usr/bin/time can be used. In the following example, the number of array rows is 10, and there were 176 minor page faults (shown in **bold**):

```
$ /usr/bin/time ./page_fault 10
Page size: 4096
Int size: 4
Command exited with non-zero status 1
0.00user 0.01system 0:00.02elapsed 71%CPU (0avgtext+0avgdata 580maxresident)k
0inputs+0outputs (0major+176minor)pagefaults 0swaps
```

   Do the following experiments. **Show sample output in your report.**

a. Run the program several times, increasing the number of rows by the same amount (e.g. 100) each time. Describe what you observe for changes in the quantity of minor page faults. Explain the results (NOTE: any reasonable explanation will receive full credit).

b. Remove the outer for loop (the loop that uses "i") from page_fault.c, and repeat the experiment in (a). Describe what you observe for changes in the quantity of minor page faults. Explain the results (NOTE: any reasonable explanation will receive full credit).

c. Remove the inner for loop (the loop that uses "j") from page_fault.c, and repeat the experiment in (a). **Be sure to keep the outer loop with "i"!** Describe what you observe for changes in the quantity of minor page faults. Explain the results (NOTE: any reasonable explanation will receive full credit).

3. *WARNING: For this problem, use the Case HPCC via an interactive session or a standard Linux system. The EECS servers do not produce proper results. The program "pagemap.c" will not work on Mac OS and Windows Subsystem because they do not create a /proc/pid/pagemap file*. The program "pagemap.c" is provided and prints frame numbers for a range of logical addresses entered by the user. You should be able to compile the program without any special options. It is used as follows:

./pagemap *pid start stop*

where *pid* is the ID of a process to be analyzed, *start* is a starting address in hex format (with 0x prefix), and *stop* is a stopping address that is not included in the analysis. Below is an example where the PID is 19548:

```
$ ./pagemap 19548 0x7fff5e6e2000 0x7fff5e750000
```

As demonstrated in lecture the output includes the frame number, sometimes referred to as the "page frame number" or "pfn". A value "0" is displayed for frame numbers that are not mapped (described as not present). Do the following:

a. Run the program from HW #2, problem # 3, using the "pmap" command. You may use your previous solution or the posted solution. Remember that it should print the addresses of two variables. **Copy the output from "pmap" in your report.**

b. Run "pagemap.c" to observe the page numbers for the memory segment reported by pmap in (a) that contains the variables whose addresses were printed. *NOTE: The program needs to be running, and you need the PID!* You may find that most of the frame numbers are not present (pfn is zero). **In your report, show the output lines for all non-zero frame numbers.** Omit the lines that have zero frame numbers in the output.

c. Answer the following questions:
   - What is/are the frame number(s) that correspond to the variables whose addresses you printed?
   - Are the frames adjacent? In other words, are the frame numbers in order (e.g. 0x1ae, 0x1af, 0x1b0)?

COMMENTS: Optional information can be found here:
- File source: https://github.com/dwks/pagemap
- https://blog.jeffli.me/blog/2014/11/08/pagemap-interface-of-linux-explained/
- https://www.kernel.org/doc/Documentation/vm/pagemap.txt