Week 5 - Problems

The problems for this week will be a bit different from the ones in previous weeks and will serve to demonstrate some of the concepts we've discussed this week.

For this week, we're not providing an image, as we recommend doing these exercises directly on your host, or ideally on a Linux machine/virtual machine.

Question 1

Often, you will notice some systems will start using more memory than is available in physical main memory. In these situations, the extra memory will go over to something called "swap memory", which is a designated space on disk where pages that aren't used often are sent. This is so that main memory can be freed up for allocating more pages. On Windows, it's a file, while on Linux it's *usually* a dedicated partition on your disk. This question will build up some intuition for how it works.

1. First, we need to understand what information is available when deciding whether to evict a page. This is the intel IA64 programmer manual. The document is massive, so skip to Vol. 3A 4-10. Pages 10 through 13 here are the relevant ones for our discussion. Take a look at the data stored in page table entries (PTE) in figure 4-4 and table 4-6. This tells you what each segment of memory of the page table entry is used for. Which ones would be relevant when deciding whether to evict, or kick out a page to swap?
2. Given the information from the last question, imagine you were designing the code in the kernel that would choose which pages to kick out based on the data in the page table entry. How would you design this algorithm?
3. Look into the clock algorithm. How does it compare to your design? Is it better or worse, and in what axes does it perform differently?
4. Imagine runtime and memory wasn't important. How would you design a page eviction algorithm that would perform perfectly?

Question 2

In your language of choice implement two programs. Have the first fills a vector of fixed size with random integers between 0 and 100, calculates the sum, and prints it. For the second program, do the same, but use a dynamically allocated linked list.

We recommend using C++, which provides a linked list as well as a vector datatype. However, you can also create your own linked lists in C, using structs and `malloc` (read `man malloc` and search online to learn more about this).

1. Collect statistics and graph out what happens to the runtime of both programs as the size of the inputs gets larger (feel free to use something like excel or google sheets for this). Which one is slower? Is the difference significant? Does the order of which one's faster flip at some point?

   HINT: You can use the `perf` command to time this (don't use `time`, it's not accurate enough). For example, you can use `perf stat -r 100 ./my_program` to get information on runtime for a program (this will run your program 100 times to get you an average and standard deviation on the data). You might need to look up instructions online for how to install it on your system. Look at the output of `perf stat --help` for how to use it, and for other useful flags.

2. Add the `-d` flag to the `perf stat` invocation. It should give you information on the number of L1 cache misses (one of the CPU caches). How do the two programs compare on this front? (Note: you might need large lists, around and upwards of 100K elements, to see the effects).
3. At which size do your lists stop fitting on a page? Do you see anything in the data that might show you roughly where that threshold is?