1. Virtual memory overview
    a. Fixed and variable-size partitioning is inefficient
        i. Fixed size partitioning leads to bad fits for small processes
        ii. Variable size partitioning leads to memory fragmentation
        iii. Both require the entire process' memory space to be in RAM at the same time
    b. Instead, let's divide RAM into small, fixed size chunks called *frames*
        i. OS must keep track of unallocated frames in a list
    c. Processes' address space can be divided into *pages* that are the same size as RAM frames
        i. Processes can be assigned multiple pages if it needs more memory
        ii. This way, only the last page in a process isn't necessarily full
        iii. Pages always sized in powers of 2, to make addressing easy
        iv. Imagine a picture frame
            1. The picture frame is a RAM frame
            2. The picture that goes inside it is the virtual page
            3. We can change the picture inside, but the picture must be the same size as the frame
    d. Given process can have one or more of its pages in RAM at a time
        i. OS maintains a page table for each process
        ii. CPU does conversion from logical to physical address based on page table
    e. Virtual memory reduces the memory fragmentation of partitioning
        i. This comes at the cost of having to keep track of many page tables
        ii. Page tables must be stored in RAM

2. Virtual memory and ties to caching
    a. Virtual memory only keeps active pages of each process in memory
        i. Very similar to caching
            1. Caching stores values in cache and RAM
            2. Virtual memory stores value in RAM and on disk
        ii. All that virtual memory does is mapping
            1. Maps a virtual address to a physical address
    b. When a process references data that is in a page not in RAM, have a *page fault*
        i. Same idea as a cache miss
    c. When a page fault occurs, must overwrite existing page in RAM with the new page
        i. If the old page had been modified, must be written back to RAM first
        ii. Same idea as a line eviction and the dirty bit

3. Ways to fetch pages
    a. Demand paging – bring a page into memory only when requested, not in advance
        i. If pages are too small, end up discarding a page just before we need it again
        ii. *Thrashing* – keep swapping same pages that we just discarded instead of working on program
    b. Alternative based on locality (both spatial and temporal)
        i. Program tends to cluster its memory references on small set of pages
        ii. That set of pages is known as a program's *working set*
        iii. Working set moves slowly over time
            1. Usually keep accessing same set of pages until we move to another part of the program
    c. Idea: could prefetch pages before they're needed
        i. Make a correct prediction? No page fault, page was already in RAM
        ii. Incorrect prediction? Wasted the time and power to prefetch the wrong page
            1. Must deal with the page fault and get the requested page
            2. Still have to pay the penalty of the page fault

4. The page table and address translation
    a. Assume our processor generates 12-bit addresses
    b. However, we only have 2 KB of memory
        i. 2^{12} = 4 KB
        ii. Not enough space for multiple programs to reside in RAM at once
        iii. Might not even be enough room for *one* program
    c. Break main memory into multiple, smaller fixed-size chunks/frames
        i. Frames are physical, pages are virtual!
    d. Remap virtual addresses generated by the processor
        i. Translate these into the physical addresses that we need to access RAM
        ii. Page table tells us exactly how to make these translations
        iii. Parameters of memory above tell us how to lay out physical and virtual addresses
        iv. Virtual memory allows us to have a process with an address space larger than all of RAM!