Purpose: This assignment provides experience with paging and virtual memory. It focuses only on the page replacement portion of paging. It is a single program, no critical sections, no semaphores.

It you remember your assembler, you had to "load" a value into a register before using it. In paging, you have to "load" a page into memory before using it. I'll call these pages `mem[0]`, `mem[1]` ...

The problem with paging is you can't just do:
`mem[2] = mem[0] + mem[1];`
because the pages that you need might not be in memory. So what you have to do instead is:
`vmem(2) = vmem(0) + vmem(1);`
where `vmem(1)` either uses `mem[1]` or it fetches from disk the page containing `mem[1]`, then uses it. Actually, in C you need two functions because one of these reads memory and the other writes, so our memory access becomes:
`wr_mem(2, rd_mem(0) + rd_mem(1));`

This means we need to upgrade what we did in the previous assignment. We will need to do page fetches and page writes.

Your virtual memory size will be 8 bytes. (You will need an 8 byte file.) Your program will have a "real" memory size of 5 pages, with one byte (a very small integer) per page. That is, your program will need 8 bytes of memory, but you only have 5; so at least 3 of your pages will need to be saved in a disk file. This disk file is your paging area. (Note this is often inaccurately called the swap area.) As your program runs, you will need to load new pages from the page file, but you won't have enough memory, so you will have to replace existing pages.

You will still need your page table (which is now length 8 instead of length 3). If the page is in memory, you need a page table entry indicating where in memory the page is (just like the previous assignment). If the page is on disk, you need to remember that also. To keep it simple, each of our pages will have a "reserved" place to go if the page is on disk, that is, page 0 will always be found in byte 0 of the page file, page 1 will be found in byte 1, etc. This is a simplification so that your program does not have to remember where on disk a page is, because it can always figure it out from the page number. Your paging algorithm will be the simple *second chance* algorithm (or LRU approximation algorithm) described both in lecture and in the book. For this assignment you will *not* use dirty bits, that is, the algorithm will pick a page for replacement without considering if the page has been modified and it will not try to avoid writing unmodified pages.

You should start with the `paging.c` program which is available in the instructor's 326 directory. You will implement both virtual memory functions. `rd_mem` will get a value from memory and `wr_mem` for storing a value in memory. Both are very similar, but the difference is that to change memory you need to be passed two values, the memory location to be changed and the new value to be placed in that location; while to fetch a value from memory you need to be passed the memory location and you need to return the value from the memory location.

To do paging you will need: 1) main memory; 2) a page table; 3) one bit per page to remember if the page is in memory (1 means in memory, 0 means on disk); 4) one reference bit per frame (for the second chance algorithm); 5) one integer per frame that keeps track of which page is in that frame. I give you (1) and (2), they use the same identifiers as in the previous assignment, you need to declare the otehres. For (5) you can use the following convention, if the integer is -1, the frame is unused. All frames should start unused. You will need to initialize some of these globals. Do that at the point indicated in the main program; the OS does this at boot, this part of the main program represents the OS boot/initialization process.

For every memory access, your program will report to the screen, a) If the page was in memory: which frame was the page in. b) If the page was loaded from disk: which frame was it loaded into. c) If a frame was paged out: what frame was emptied and what page was paged out from that frame.