Assignment 03
Part II Programming (110 points)

Specification: Multi-level Paging with Page Replacement

Code structure

You are **required** to **separate** the logic of page table operations, page replacement operations, and the main flow of the functionality to separate code files, although you are certainly encouraged to use more if you see appropriate.

You are given considerable latitude as to how you choose to design your data structures and interfaces. However, your implementation **must** fulfill the **implementation requirements** specified in the **Level** structure and **mandatory interfaces** below. A sample data structure is summarized below and advice on how it might be used is given on the Canvas assignment page, refer to *pagetable.pdf*.

Page table structures

- **PageTable** – Top level descriptor describing attributes of the N level page table and containing a pointer to the level 0 (root node) page tree/table structure.
  - PageTable stores the multi-level paging information (see pagetable.pdf) which is used for every page Level (or node) object: number of levels, the bit mask and shift information for extracting the part of VPN pertaining to each level, the number of entries in each non-leaf internal Level array to the next level objects, or the array to the PFN mapping for the leaf level / node, etc.
  - Since the tree operations start from root node, it would be convenient to have the PageTable have a reference / pointer to the root node (Level) of the page tree.

- **Level** (or PageNode) – An entry for an arbitrary level, this is the structure (or class) which represents one of the levels in the page tree/table.
  - Level is essentially the structure for the multi-level page tree node. Multi-level paging is about splitting and storing the VPN information into a tree data structure along the tree paths. Starting from the root node, each tree path (the root node to a leaf node) stores a VPN, and the leaf node captures the PFN the VPN is mapped to.
  - Level (or Node) contains an array of pointers to the next level or page mappings.
    - For non-leaf or interior level nodes: an array of Level* (or PageNode*) pointers to the next level, it is essentially a double Level**
    - For leaf level nodes: an array of mappings, each mapping maps a VPN to a PFN physical frame
    - Note you can have both a Level* array (for non-leaf interior nodes) and a mapping array (for leaf nodes) included in the Level structure and use the one dependent on where the level is at: i.e., interior or leaf level.
  - Implementation **requirements** (violation would incur **50% penalty** of **autograding**):
    - You must implement this Level / PageNode structure as a tree.
    - You must **NOT** use a hash map for storing children levels / nodes, you could use either Level** or Level*[] or std::vector<Level*> or similar **array** representation.

- **Map** – A structure containing information about the mapping of a page to a frame, used in leaf nodes of the tree. It is ok to just use a number for the frame, you may use a special value (like -1) to indicate whether the frame is a valid frame, make sure you put proper comments.

Page table mandatory interfaces

Implementation **requirements** (violation would incur **50% penalty** of **autograding**): You **must** implement **similar** functions for multi-level paging as proposed below. Your exact function signatures may vary, but the functionality should be the same.

All other interfaces may be developed as you see fit.

- unsigned int **getVPNFromVirtualAddress**(unsigned int virtualAddress, unsigned int mask, unsigned int shift)
    - Given a virtual address, apply the given bit mask and shift right by the given number of bits. Returns the virtual page number. This function can be used to extract the **VPN of any level** or the **full VPN** by supplying the appropriate parameters.
    - Example: With a 32-bit system, suppose the level 1 pages occupied bits 22 through 27, and we wish to extract the level 1 page number of address 0x3c654321.
        - Mask is 0b00001111110000000000, shift is 22. The invocation would be **getVPNFromVirtualAddress**(0x3c654321, 0x0FC00000, 22) which should return 0x31 (decimal 49).
        - First take the bitwise '**and**' operation between 0x3c654321 and 0x0FC00000, which is 0x0C400000, then shift right by 22 bits. The last five hexadecimal zeros take up 20 bits, and the bits higher than this are 1100 0110 (C6). We shift by two more bits to have the 22 bits, leaving us with 11 0001, or 0x31.
        - Check out the given **bitmasking-demo.c** for an example of bit masking and shifting for extracting bits in a hexadecimal number.
    - **Note**: to get the full Virtual Page Number (VPN) from all page levels, you would construct the bit mask for all bits preceding the offset bits, take the bitwise **and** of the virtual address and the mask, then shift right for the number of offset bits.

      Full VPN (all levels combined) is needed for page replacement, see below.
- void **insertVpn2PfnMapping** (PageTable *pagetable, unsigned int vpn, int frame)
    - Used to add new entries to or update the page table for the VPN to PFN mapping:
        - when the virtual page (VPN corresponding to the passed-in virtual address) has not yet been allocated with a frame (findVpn2PfnMapping returns NULL or invalid).
        - when trying to update mapping of the VPN to a particular PFN.
        - frame (or PFN) is the frame index that is mapped to the VPN.
            - **important:** you could use a **-1** value for the **frame** argument for updating the mapping to be invalid. In page replacement, you would need to update the replaced (evicted) VPN to PFN mapping to be invalid.
        - if you wish, you may replace void with int or bool and return an error code if unable to allocate memory for the page table.
        - **HINT**: If you are inserting a page, you do not always add nodes at every level. Part of the VPN may already exist at some or all the levels.
- Map * **findVpn2PfnMapping**(PageTable *pageTable, unsigned int vpn)
    - Given a page table and a VPN, return the mapping of the VPN to physical frame from the page table, return type could be a custom Map structure or just a frame number. If the virtual page is not found or the mapping is not with valid flag, return NULL or an invalid frame number. Note that if you use a different signature than the one proposed, the function name and idea should be the same. If findVpn2PfnMapping was a method of the C++ class PageTable, the function signature could change in an expected way: Map *

PageTable:: findVpn2PfnMapping (unsigned int vpn). This advice should be applied to other page table functions as appropriate.

Page Replacement – a modified working set clock (WSClock) algorithm

As part of the MMU simulation, you are required to implement a simulation of a modified working set clock (WSClock) page replacement algorithm for finding a victim frame to evict and allocating the freed frame to the VPN being accessed. Page replacement happens when there is a page fault (i.e., page hasn't been loaded and mapped to a frame), and all available physical frames are used.

Data Structures

Like what's described in the TEXTBOOK for the WSClock algorithm, a circular list (or a ring like structure) is used for tracking the most recent access to the allocated page frames.

The circular list starts empty. When the first page is loaded, it is added to the list. As more pages are added, they go into the list to form a ring. The list should **keep the order** of pages added **according to the frame number**, i.e., frame 0, frame 1, frame 2, etc. After the list size reaches the number of available frames, a page fault would trigger the page replacement to run (part of the page fault handling). **Note**:

- The WSClock circular list has the size of allocated frames.
- The number of available frames is specified by the -f command line optional argument or the default value of it (see user interface in a3.pdf).
- Each entry in the list contains:
    - mapped page (VPN) information,
    - the last access time to the page,
    - and dirty flag indicating whether the page has been written since it was loaded.
        - read and write access history is from the file specified in the second mandatory argument of the program. The file contains a series of zeros and ones, 0 means it is a read access, 1 means it is a write access.

With **every memory address access**, it needs to:

- update the last access time of the corresponding page entry or set the newly added page entry in the WSClock circular list.
    - Use a virtual time for tracking each memory / page access time. An **address count** would be a good choice serving as a virtual time, it starts from zero before the access of the first address and is incremented by 1 after reading each address from the trace file.
- update the dirty flag if necessary, dirty flag starts with false:
    - a read access should NOT change it.
    - dirty flag should only be set to true by a write access.
- perform page replacement (below) if necessary

Algorithm

Starting from the current clock hand position, the **WSClock** page replacement algorithm finds the first page that has an age of last access greater than a threshold and the page is clean (not dirty). The page frame is simply claimed, and the new page put there. On the other hand, if the page is dirty (has been written), it cannot be claimed immediately before the page is written to persistence storage. To avoid a process switch, the write to persistent storage is scheduled, the clock hand is advanced, and the algorithm continues with the next page.

You need to **implement a modified WSClock version as specified below**:

- The **first** run of the page replacement should start the clock walk from the **frame 0** position, advance to frame 1, then frame 2 etc. Subsequent runs of page replacement should **start** the **clock hand from the position right after** the last victim frame.
- After the clock walk reaches the last allocated frame position, with the wsclock circular list, it would go to frame 0 position again to continue the search.
- The page at the clock hand index position is inspected first, and advance one position at a time, until it finds a page frame:
    - whose last access time is older than an age threshold (ageOfLastAccessConsideredRecent):
        - the age threshold is from the -a command line optional argument or the default value of it. Again, assume autograder testing will not test an age threshold bigger than the number of available frames (see user interface in a3.pdf).
        - you could use
          last access virtual time < current time – ageOfLastAccessConsideredRecent
          to tell whether a page's last access time is over the age threshold.
    - and whose dirty flag is false:
        - every time a page frame with a true dirty flag is inspected, clear the dirty flag, this is to emulate the scheduling for write to disk (see above), and advance to the next page frame in the circular list.
- Once the victim frame is found:
    - update the victim entry in the wsclock circular list with the vpn being accessed.
    - update the page table for the new mapping and invalidate the old mapping (see insertVpn2PfnMapping above).
    - note for logging with vpn2pfn_pr flag, you also need to pass vpn being replaced (or evicted) to the logging function.

Important: You may use any C++ standard template library class for implementing the page replacement.