Assignment 03 Programming specs
Multi-level Paging with Page Replacement

Code Structure

The implementation of page table, page replacement, and the main flow of the program should be separated into their respective code files. You are certainly encouraged to use more code modules 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** (or page table node) structure and the **mandatory interfaces** below. A sample data structure is summarized below and advice on how it might be used is given on the assignment page (canvas), refer to ***pagetable.pdf***.

Page table structures

- **PageTable** – A descriptor containing the attributes of a N level page table and a pointer to the root level (Level 0) object.
    - PageTable stores the multi-level paging information (see pagetable.pdf) which is used for each Level or tree 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 to the next level objects in each non-leaf internal Level, or the number of entries in the array of the PFN mapping in the leaf node level, 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 in c++) 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 encapsulates the PFN the VPN is mapped to.
    - Level (or Node) contains an array of pointers to the next level or virtual page to physical frame mappings.
        - For non-leaf or interior level nodes: an array of Level* (or PageNode*) pointers to the next level, essentially a **double pointer 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 the selection of using which one is dependent on where the level is at: the Level* array is used by the non-leaf interior levels, the mapping array is used by the 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 (or next) levels / nodes, you could use either 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. You need a way to indicate whether the mapped frame is valid, you can either use a boolean flag for that or use a special value like -1 to indicate it is an invalid frame. A Map instance is initialized with a false valid flag or a frame value with -1 and assigning the frame with any number 0 or positive would indicate it is a valid mapped frame. Make sure you put proper code 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.

- Map * **searchMappedPfn**(PageTable *pageTable, unsigned int virtualAddress)
    - You may use a different signature, but **virtualAddress argument MUST be there**, and the function name and idea should be the same.
    - Given a page table and a virtual address, search the page table and return the mapped physical frame of the VPN of the virtual address, 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. If **searchMappedPfn** was a method of the C++ class PageTable or class Level, the function signature could change in an expected way: Map * PageTable:: **searchMappedPfn** (unsigned int virtualAddress) or Map * Level:: **searchMappedPfn** (unsigned int virtualAddress). This advice should be applied to other page table functions as appropriate.

- void **insertMapForVpn2Pfn** (PageTable *pagetable, unsigned int virtualAddress, int frame)
    - You may use a different signature, but **virtualAddress argument MUST be there**, and the function name and idea should be the same.
    - Insert the VPN along a page table tree path with the VPN to PFN mapping, **or** update an existing VPN to PFN mapping:
        - when the virtual page (VPN corresponding to the passed-in virtual address) has not yet been allocated with a frame (i.e., searchMappedPfn returns NULL or invalid for the vpn).
        - 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.
        - You may replace void with int or bool and return an error code if unable to allocate memory for the page table.
        - **Important**: When inserting a page (VPN from the virtual address), you do not always add nodes at every level. Part of the VPN may already exist at some or all the levels from previous insertions.

- unsigned int **extractVPNFromVirtualAddress**(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 page 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 **extractVPNFromVirtualAddress**(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.

Page Replacement – a modified NFU (Not Frequently Used) with aging

As part of the demand paging simulation, simulate a modified NFU (with aging) page replacement algorithm for finding a victim frame to evict and allocating the freed frame to the unallocated 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.

Refer to the **TEXTBOOK 3.4.7** "Simulating LRU in Software" for general details of the NFU with aging algorithm. But **make sure you read the details below for the modified** version you need to implement.

Data Structures (proposed, not mandatory)

For **page replacement**, you may use any collection class (type) from the C++ Standard Template Library.

Use a collection to remember all loaded page information and track their recent accesses, each entry contains:

- **Virtual page number**, and its allocated physical frame number.
    - You may not need to store physical frame number (pfn) if you use the index of the collection as the pfn.
- For each loaded page, a **16-bit** bit string is used for remembering how recently and frequently the page has been accessed.
    - Bit string should start with a value of **1 << 15 (2^{15})** for the page when it is first loaded. This ensures the newly loaded page has a relatively large bitstring to start with since it was just loaded.
    - You need to use **a separate collection for tracking the page accesses during the current NFU interval** for bitstring update (see algorithm below).
- **The last access time** to the page

    - You could use **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.

Algorithm

When allocation reaches the limit of available frames, a page fault would trigger the page replacement to run (part of the page fault handling). **Note**: 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).

The NFU with aging page replacement inspects the loaded (allocated) pages and **picks the one with the smallest bitstring as the victim frame**. If there is a tie in bitstring (i.e., more than one frame has the smallest bitstring), **pick the one with the smallest last access time among the tied ones**.

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

- Update the information for **tracking the page accesses** during the current NFU period, i.e., remember which pages have been accessed during the current NFU period since the last bit string update.

- Check and do the **bit string update if needed**. **When** it needs to perform bitstring update: if the number of memory accesses since the last **bit string update** has reached the number of memory accesses for bit string update (the NFU bit string update interval).
  This is necessary because each memory access could trigger a page replacement, we need to have the bit string updated **first** (before the potential page replacement) if the NFU update interval has also been reached with the current memory access.

  How the bitstring update is done:
  - Traverse the loaded page collection and **for each loaded** page:
    - shift the current bitstring to the right by 1,
    - then for **each page accessed during the NFU period**, **prepend** the shifted bitstring by a **ONE** bit.
      - e.g., suppose the page was accessed during NFU period, its current bitstring is 0b0100000000000000 will become 0b1010000000000000.
    - **REMEMBER to reset the page access stats** during the last NFU period.

- Perform **page replacement,** if necessary, i.e., if the page being accessed is not loaded and all available frames are used. See above for the NFU with aging algorithm.
  Once the victim frame is found:
  - update the victim entry in the loaded pages collection to the new vpn being accessed.
  - update the page table for the new mapping and invalidate the old mapping (see insertMapForVpn2Pfn above).
  - note for logging with vpn2pfn_pr flag, you also need to pass victim frame’s vpn and its current bit string to the logging function.

- Update the **last access time** of the corresponding page entry (or newly loaded page entry).