



Assignment 03
Part II Programming (120 points)
Due: Beginning of the class, Mar 22nd

Specification: Multi-level Page table with TLB

Page table mandatory interfaces and structures

- unsigned int virtualAddressToPageNum (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 access the page number of any level by supplying the appropriate parameters.
 - Example: Suppose the level two pages occupied bits 22 through 27, and we wish to extract the second level page number of address 0x3c654321.
virtualAddressToPageNum(0x3c654321, 0x0FC00000, 22) should return 0x31 (decimal 49). Remember, this is computed by taking the bitwise and of 0x3c654321 and 0x0FC00000, which is 0x0C400000. We 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.
 - **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.

VPN from all levels combined is needed for caching the Virtual Page Number (VPN) to Physical Frame Number (PFN) mapping in the TLB, see below.

- **PageTable** – Top level descriptor describing attributes of the N level page table and containing a pointer to the level 0 page tree/table structure.
 - PageTable captures your multi-level paging information (as put in the pagetableDS.pdf) which is used for every Level 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 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** – An entry for an arbitrary level, this is the structure (or class) which represents one of the sublevels in the page tree/table.
 - Level is essentially the structure for the page tree node. The multi-level paging is about splitting and storing the VPN information into a tree data structure. Starting from the root

node, all the way to the leaf node which captures the mapping from the VPNs to PFNs (in the Map[] array in the leaf node) along the VPN path from the root node to that leaf node.

- **Map** – A structure containing information about the mapping of a page to a frame, used in leaf nodes of the tree.

You are given considerable latitude as to how you choose to design these data structures. A sample data structure and advice on how it might be used is given on the Canvas assignment page. Refer to [*pageTableDS.pdf*](#).

- `Map * pageLookup(PageTable *pageTable, unsigned int virtualAddress)`
 - Given a page table and a virtual address, return the mapping of virtual address to physical frame (in Map*) from the page table. You must have an appropriate return value for when the virtual page is not found (e.g. NULL if this is the first time the virtual page has been seen). Note that if you use a different data structure than the one proposed, this may return a different type, but the function name and idea should be the same. Similarly, If pageLookup was a method of the C++ class PageTable, the function signature could change in an expected way: `Map * PageTable::pageLookup (unsigned int virtualAddress)`. This advice should be applied to other page table functions as appropriate.
- `void pageInsert(PageTable *pagetable, unsigned int virtualAddress, unsigned int frame)`
 - Used to add new entries to the page table when we have discovered that a page has not yet been allocated (pageLookup returns NULL). Frame is the frame index which corresponds to the page number of the virtual address. Use a frame number of 0 the first time this is called, and increment the frame index by 1 each time a new page→frame map is needed. 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. The Map structure may already exist at some or all of the levels.
- All other interfaces may be developed as you see fit.

Translation Lookaside Buffer (TLB)

As part of the MMU translation simulation, you are required to implement a simulation of TLB for caching the virtual page to physical frame mappings from the page table, each TLB entry caches one mapping of a Virtual Page Number to a Physical Frame Number (**VPN → PFN**, whichever the way you prefer to implement).

The TLB size (max number of mappings or entries) can be designated by an optional command line argument (see -c in the user interface specification below). The default size is 0 if the argument is not specified.

Your TLB cache will use an approximation of the least recently used (LRU) cache replacement policy. When TLB is full (number of mappings cached reaches the TLB size) and a TLB miss occurs, i.e., there

is an attempt to access an address of a virtual page whose mapping is NOT cached, you would need to go to the page table (or demand paging to allocate a new frame for mapping) to find the mapping, then you need to insert that mapping to the TLB, but since the TLB is already full at the moment, you need to evict an existing mapping from the TLB and then you can insert the new mapping to the TLB, the determination of which mapping to be evicted is what we call cache replacement policy. Here you would use an approximation of the least recently used policy which tracks a set of recently accessed **pages** to evict the oldest accessed page tracked there.

It is reasonable to track a small set of recently accessed pages for approximating the perfect LRU cache policy. In this assignment, you will track the **10 most recently accessed pages (with distinct VPNs)** and their access times for implementing the LRU cache replacement. Note a perfect LRU cache policy would track the access times of all cached pages in TLB which would add a lot more overhead and usually is not practical.

Important:

- An access to a memory address is simulated by reading in a virtual address from a trace file.
- With every virtual address access, insert or update the recently accessed pages for the virtual page (that contains the virtual address) being accessed and the access time. If the page being accessed is not in the recently accessed pages and if the number of recently accessed pages reaches 10, the oldest page (the page with the smallest last access time) tracked there should be removed, and the page being accessed should then be added to the recently accessed pages.
- It is very **important** to understand that you need to track the **10 most recently accessed PAGES (with distinct VPNs)**, NOT the last 10 accessed addresses. There could often be a series of access attempts to the virtual addresses of the **same** page.
 - As part of the implementation for cache replacement, whether it is a cache hit or miss, you always need to update the most recently accessed pages.
- After a cache miss, whether it is a page hit or page miss, the latest accessed page always needs to be inserted in the TLB. And when TLB is full, you would need to perform cache replacement.
- You may want to use a virtual time instead of the actual 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 / accessing each address from the trace file.

You are free to use the data structures you see most fitting for implementing the TLB and its cache replacement policy.