Read Me

Design Choices

The main structure that operations are performed on is the proxy object, the page_table structure defined in prsim.h. It contains a number indicating how many free frames are left, a list of nodes that contain references to the PTEs that are in memory, a hash table containing the PTEs themselves, and function pointers to the appropriate replacement policies.

A PTE is stored within a somewhat generic node defined in prsim.h. The key is the page number and the data field is used to store the modified and valid bits representing whether or not a page has been written to and needs to be written back to disk and whether or not the page is in memory, respectively. These bits are retrieved via a bitmask that is created with the create_bitmask function. Note that the function creates the mask from left to right (i.e. the leftmost bit (bit 31) is the "first" bit).

The hash table used to store the PTEs is based on a simple hashing function (i.e. page number % hash table size). The size of the hash table can be configured via changing the pt_size constant located in prsim.c. Initially, pt_size is set to 0x40000000. So the space requirements for the hash table is 0x40000000 * sizeof(*linked_list) byte / 1 MB = ~4 MB. This can grow depending on how many collisions, since the hash table uses separate chaining to resolve conflicts.

The replacement algorithms are implemented as separate functions located in policies.h/policies.c. Each implementation has two functions associated with it: **_add_page_mem_policy and **_replacement_policy, where ** represents FIFO, Random, or LRU. The add_page_mem_policy is used when there are free physical frames left and the particular page can be loaded into that frame. If there are no free frames available, then the replacement_policy function is called instead. The only reason why I differentiated between these two cases is because the method one chooses to populate the inmem_pages list is important. For example, in the case of

LRU, the head of the list represents the most recently used page, so therefore the pages must be inserted to the front of the list. In the case of FIFO, the PTE references are inserted at the end of the list; therefore the front of the list represents the next page to be evicted. In the case of random, it doesn't really matter the order in which the PTE references are inserted into the list. The **_replacement_policy functions work in a similar fashion based on the order of the inmem_pages list.

The start_simulator and pt_load_page functions are where the vast majority of the logic is located. The start_simulator page initially creates the page_table structure and sets the appropriate replacement policy based on the command line arguments. This function is also where the memory reference strings are read in. The memory references are read in, 128 references at a time, using the 'read' command.[1] Once those 128 references are read in, the simulator begins processing them via the pt_load_page function.

The pt_load_page function first queries the hash table contained in the page_table. If it cannot find the PTE, an entry is created in the hash table for the PTE. Therefore, the page_table is lazily loaded. After the PTE is retrieved (or created), the simulator checks whether or not the memory reference is a write instructions. If it is, the page is marked as dirty. Then, the function checks if the page is valid; if it is not, a page fault occurs and the loading of the page is dealt appropriately using either the **_add_page_mem_policy or **_replacement_policy functions depending on whether or not there are any free frames left. There is also an extra step for when the LRU replacement policy is chosen. The inmem_pages list is updated when a page has been referenced and is in memory. The page is brought to the front of the list to indicate that it is the most recently used page.

---

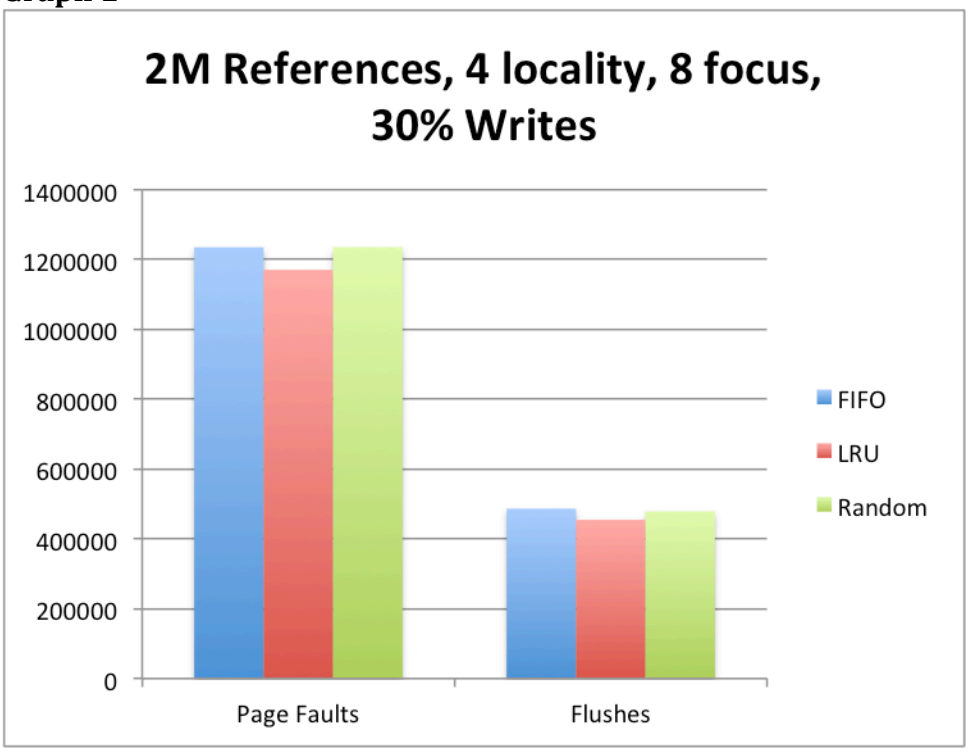
[1] I know the requirements state that the memory references had to be "read in one by one", but that seems highly impractical, since the amount of time spent in System mode ('read') is ridiculous, since you are calling read $\Theta(|\text{memrefs}|)$. The amount of overhead is unacceptable. When the number of memrefs < 128, then "technically" the simulator is waiting for all the references to be read in. But how do we know how many references are being passed to the simulator??

Limitations:

My implementation of prsim breaks when the max memory reference address approaches 2^{22}. I'm assuming the problem lies in the hash table and the hashing function. I wish I had time to find a solution to this problem, as I hate leaving this unresolved. If I had more time, my solution to this problem would be to use a 2-level hash table and 2 use hash functions: 1 to locate which hash table, 1 to find which entry contains the PTE in the second hash table, using two different parts of the address for each hash function. The hash tables would not use linked lists to store the buckets, just a simple array containing pointers to the data. Using this implementation one could easily get any PTE in at most O(2) time.

Performance Benchmarks

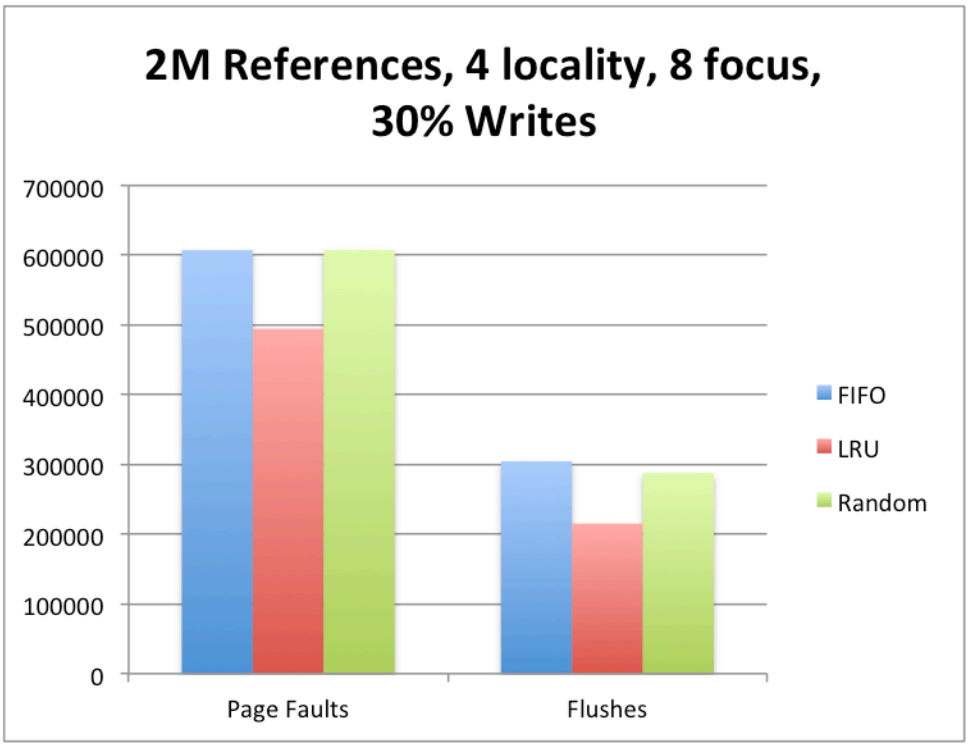
Graph 1



Note: Page size = 32 bytes, Memory Size = 64 bytes, 2 Physical frames

When there are only 2 physical frames, the page fault rate is very high for all the replacement algorithms. LRU is slightly better because it recognizes when certain pages are used more often and keeps them in memory, although its benefits are minimized since the number of frames is so low.

Graph 2

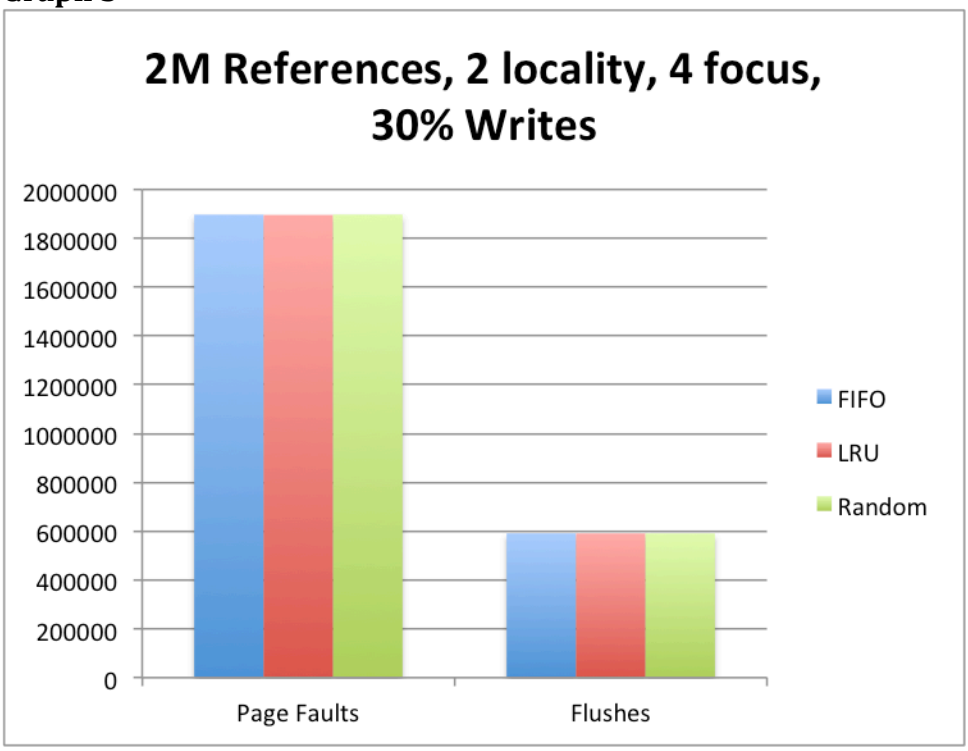


Note: Page Size = 32 bytes, Memory Size = 256 bytes, 8 physical frames; Same memory reference string as Graph 1.

The performance the FIFO and Random algorithm have similar performances. The benefits of the LRU algorithm can be seen a little more clearly now, since the number of frames has increases. Since the focus is high in the memory reference string, the LRU algorithm is more likely to realize which pages are used the most and are kept in memory the longest, thus decreasing the number of page faults, and by extension, page flushes.

The one downside of the LRU algorithm is its running time. Every time a page is referenced that is in memory, the linked list containing the pages in memory needs to be altered, bringing the most recently used page to the front of the list. These operations on a linked list can be costly if the same few pages are constantly be referenced, since the list's nodes are constantly being shuffled around.

Graph 3



Note: Page Size = 32 bytes, Memory Size = 256 bytes

When locality and focus are low, all three replacement policies perform in a similar fashion: poorly, as one can see. Since FIFO and LRU rely heavily on the principle of locality, the horrible performance is not unexpected.