Guthrie Alexander

CS 4414- Operating Systems

Project 3 Writeup

Due: April 14 2016

Design and implementation of project 3

**Design**

I started off making sure I understood the assignment correctly, and what the execution of the program had to accomplish. Basically, the program would need to read in a list of virtual addresses, using a TLB and page table that are constructed within the program, and then translate the virtual address to a physical address. It’s stated in the specification that for this project we don’t need to worry about an offset, and that as long as the page of memory is loaded that is sufficient.

Once I had a solid understanding of the overall flow of the program, I started coding in order to read in the numbers and get the page numbers, and went from there.

**Reading and Extracting Page Number**

As you can see in the project.c file, the first thing the main function does after declaring and initializing variables and arrays as well as allocating memory is to read in the list of numbers, and then store their corresponding page numbers in a separate array.

The first method used is **getSize()**, which just returns the number of numbers in the file (assuming the last line in the file is blank). I did this in order to allocate the correct amount of space for the **int \* input** in the main function. Then, I set input, which is just a blank array at this point, equal to the result from the **readIn()** function, which results in the input array holding the addresses. The next function call is **pageNum()**, which returns an array of ints, the same size as input (i.e. same number of elements in both arrays). The array returned from this function contains the page numbers for each of the logical addresses in input.

The page number is obtained by doing a rightshift 8 places for each of the logical addresses. This is because the page size is 256 bytes, meaning the offset will be 8 bits would be needed to represent each of those bytes (in the offset). This means the right 8 bits in the number would be the offset, and the rest would be the page number, hence why doing an 8-bit right shift would work to consistently yield the page number. The pageNums array holds the values for the page numbers for the input numbes from addresses.txt. Now that the program has the page numbers, it can proceed to the next steps.

**TLB**

The for loop that surrounds the rest of the main function iterates through the pageNums array, and checks and updates the TLB, page table, and physical memory as needed. First, the TLB is checked.

Before the main function, the TLB data structure is created (as a global variable). **Tlb[4]** is an array of **tlb\_struct** type, which just contains two ints: int page and int frame. Each element in the array **tlb[]** contains a page and a frame, and there are 4 elements, just as the specification requires. At the beginning of the main function, I set all of the pages and frames to -1 so it would be clear if there was nothing in the TLB (since there is a page 0 and frame 0).

After the variables are set at the beginning of the for loop at line 212, the while loop at line 220 checks the TLB. Until there is a hit or the tlb\_index reaches 4, the page number (**pn** variable) is checked. If it’s in the TLB, the **tlb\_hit** variable is set to 1, and the result is printed to console. If the page number is in the TLB, the only other thing to do is update the LRU list for the Page Table replacement policy. This happens in the call to the **moveToEnd()** function in the else statement at line 307.

The TLB will also need to be updated when physical memory is full, and an evicted page is listed in the TLB (as well as normally updated on a memory request). I’ll discuss how I implemented that in the LRU section.

**Page Table**

If there is a TLB miss (i.e. if tlb\_hit==0, line 230), then the page table must be consulted. The page table is implemented using a single struct **page\_table** that contains an array of pages. The index of the array is the page number, and the value stored is the frame where the page can be found in memory. The instance of the struct is declared as a global variable **pt**, and initialized in the main function. I initialized the entire array to -1 to make it clear when a page table entry was empty (-1) and when it was full (0-7).

The first thing the main function does if there was a TLB miss is run the **checkIfInPT()** function, which checks if the page number is in the page table. The function is simple, it checks pt.pages at index **pn**, or page number, and if the element is -1, then it will return 0. If it’s anything but -1, then there is an element in the array, and hence the page is already in the page table. If that result is 1, then we proceed to the else statement, which just updates the LRU linked list.

Finally, if the page was found in the page table, we need to update the TLB. This is done next, with the **tlb[next\_tlb\_index].page** and **tlb[next\_tlb\_index].frame** assignments. The next\_tlb\_index is just incremented (mod 4) every iteration, unless there was a page fault, in which case a few things happen.

**Page Faults and LRU**

If the **checkIfIntPT()** function returns 0, then there was a page fault (i.e. the page number is not in the page table currently). The first thing done is checking if the frame list is full; if it is full (i.e. the **isFull()** function returns 1), then a frame needs to be freed in order to accommodate the new page.

This is done with the **freeLRU()** function, which returns the page number of the page that was least recently used (the head linked list has a list of page numbers, with the most recently accessed number at the tail and the least recently used number at the head). This function also removes the head node from the linked list (with the **getFirst()** function). The freeLRU() function finds the frame number from the current page table, updates **free\_frames[]** to be 0, updates the **frame** variable so that that particular frame will be filled, and updates the page table so that at index **pn** the frame is -1, indicating it’s empty.

Back in the main method, we update the TLB if the evicted frame contains a page number held in the TLB. This is done with the help of a 2nd linked list, strictly used by the TLB. The **removeItem()** function locates the node in the linked list, and removes it (preserving the order of the rest of the nodes), so that the FIFO order is preserved.

Next, the **tmp\_buffer** array is filled with the corresponding page from the BACKING\_STORE.bin, which is returned from the **readBS()** function. The page table is updated with the new information, the **physical\_memory** array is filled with the corresponding data from the tmp\_buffer array, and the free\_frames[] array is updated (indicating that the frame has been filled).

Finally, just as we do when an item is found in the page table, the TLB is updated.

**Statistics and Counting**

The variable **total\_page\_faults** counts the total page faults aggregated across all the logical addresses. The **size** variable was obtained from the original reading of the addresses (i.e. it’s the number of memory references). The **total\_tlb\_hit** variable is a sum of tlb hits, which is incremented in the TLB hit section of the main function.