Ryan Lochrane

Gabriel Ybarra

09DEC20

CS433

Files Submitted on server:

DLL.h, DLL.cpp, LRUReplacement.h, LRUReplacement.cpp, pageTable.h, pageTable.cpp, main.cpp, Makefile, prog5, large_refs.txt, small_refs.txt

Programming Assignment 5

Ryan Lochrance was responsible for test 1 which takes the input file "smallrefs.txt", opens it, and begins reading in each logical address while calculating the page number and storing it into the variable pageNumberT1. Next, test 1 checks the valid bit of the pageEntry object located in the page table to see if there is a page fault(valid bit is 0) or if there is no page fault (valid bit = 1). The if statement on line 125 checks the condition of the pageFaultT1 flag and if it is true that there was a page fault, we increment the variable numberOfPageFaults by 1 then call on the function addNewEntry to insert it in the page table. After insertion we increment the variable frameNumAssign by 1 so when we insert the next item the frame number will have the correct value. Test 1 then uses some print statements to print out the statistics of the memory reference such as the logical address, page number, frame number, and if there is a page fault. Following these print statements we print out the overall stats which include number of references, number of page faults, and the number of replacements.

Finally we close the file. When test 1 ran with the file "smallrefs.txt" we noticed that it did not trigger any page replacements meaning that none of our algorithms were used. The reason for this is because we had enough physical memory to accommodate the list of logical addresses so there were enough frames to assign to each reference. We assumed since this is a simulation we could number the frames numerically as opposed to real world operations where frames most likely would be randomly assigned.

For the First In First Out algorithm we created the variable start to store the current starting time. Then we create the variable victimFrame and initialize it to zero. We set it to zero because for FIFO we assume that the first element will be the one that is chosen first. We then create a page table as well as a framelist to store the correct amount of frames. Our algorithm then uses a for loop to loop over the memory references in order to calculate the page numbers.

Inside the for loop we evaluate the valid bit to see if a page fault will occur. If a page fault occurs then we increment the variable numberOfPageFaults by 1, but we also check to see if there are frames available. If there are frames available then we simply assign a frame number to the page entry by using the addNewEntry() followed by the incrementation of the variable frameNumAssignT2 to update the next frame to be assigned.

In the case where a replacement is needed, we increment the variable numberOfReplacements by 1, followed by obtaining the frame to be replaced. Once we have chosen a frame provided to us by the variable victimFrame, we assign the page number into the array frameListFIFO. We then call the replace function where we replace the frame that has the old page number to store the value of the new page number. Finally we update the variable victimFrame by incrementing it then performing modulo numFrames to make it circular.

When we ran the FIFO algorithm for test 2 we used multiple configurations of page sizes along with physical memory sizes as well. We noticed that as we increased the physical memory size the number of page faults trended downward. When it comes to the number of page replacements it was noticed that in both page sizes 256 and 512, when the physical memory size was 64MB there were zero page replacements. All subsequent page sizes did not result in a zero page replacement for FIFO. As the page sizes increased the rate of the decrease was less and less for page replacements. Finally we noticed as the page sizes and physical memory sizes were increasing, the total execution time was trending downward until the configuration of page size 2048 physical mem size 4MB, the rate of decrease began to slow although still trending down until we hit page size 8192 physical memory 4MB we began to see an increase in execution time. The overall trend was downward, but the little spikes here and there we attributed to Belady's anomaly.

Gabriel Ybarra was responsible for implementing both the Random Replacement and the LRU Replacement algorithms. Both have the same structure as the FIFO replacement algorithm. First we use the Page Table object to create the page table that has num_pages entries in it. Each page entry has the valid bit and the frame number associated with it. We additionally have the second data structure frameListRand which for each frame, stores where it is located in the page table. So to do random replacement, first each reference is loaded one by one, the page number is calculated and the valid bit is checked for that page, if there is a page fault then there are two cases, one is where there are still free frames left to assign. These frames are assigned numerically and assigned to the page entry in the page table with the addNewEntry function. Of course, the frames location in the page table is also stored. If all the frames have previously been assigned then one of the pages has to be replaced. The pages that are eligible to be replaced are those that have the frames assigned to them and therefore have valid bits of one. So because of this we call the rand function to get a random frame number and use that one to do the replacement with the replace function.

As for the LRU implementation, the structure is the same as the Random and FIFO for how the frames are assigned, but the difference is that LRU has a doubly linked list that keeps track of which page is the least recently used one. In this case the least recently used is always kept at the front pointer of the list. This is accomplished by moving pages to the back of the list any time they are referenced. To implement the features of the doubly linked list I used the DLL.h and DLL.cpp files which are files that are modified versions of the list files I used to complete Programming assignment 1. To reiterate, these files were created by me in my cs311 class for an assignment in that class. Many of the functions from these two files are not used, but the main purpose of the file is to have the DLL utilities. In addition to the DLL files, we used the LRUReplacement files to create the DLL and to handle the page references. In the newReference function, we hand the page references by either adding them to the back of the list if they are new, or if they are in the list already, then they have to be moved up to the top. To get the victim page, the function is used and the page at the front node is returned and it is moved to the back, signifying its use. Finally to move the pages to the back of the list, we used the move node to back function, this function moves the pointers depending on the 4 cases, if the item is the only one in the list or if it is already in the rear, the list does not have to do anything, so it just simply returns. If the page is in the front, then the corresponding pointers are moved. If the page is in the middle the middle pointers are moved. Overall we choose to implement the LRU this way because it is much faster in terms of execution time because of the way the algorithm works as compared to the counter implementation.

As for the results, we can see that in the rand page replacement, it has very similar performance metrics to the FIFO in terms of page faults. As the physical memory sizes increase then the # page faults go down. In addition to this as you increase the size of the pages you also have less page fault because you have overall less pages. Looking at the Random replacement in terms of execution time also shows a more exaggerated case of Belady's anomaly, where depending on how the random pages are selected the execution time can spike up in certain locations. Overall though these execution times do trend down but it is not as good as compared to the LRU. For LRU, we noticed that our output does not really match the values of the provided executable. We were not able to determine what was exactly causing this because we know that the stack is working correctly through testing in another program, so we are not sure as to what is causing the lesser amount of page faults. Overall though the program does compile and does not crash on runtime. In addition to this despite the different numbers as compared to the example, the data we collected does show how the LRU behaves. Looking at the page faults and the execution time we can see that the decline rates are more linear and there are few if any cases of Belady’s anomaly. So in that sense LRU is the best replacement to use because it is very reliable and has good management of the page faults. This is why operating systems like linux use this method and this experiment proves how LRU is the best replacement to implement that we learned about in the lecture. The data and the graphs for the FIFO, RND, and LRU will be included on the last page.

We learned a few lessons when completing this project. We learned that the counting approach to the LRU algorithm is not the most optimal to use because you have to use a liner search to find the minimum value for count. When we tried to implement the stack approach, we ran into issues about how we structured our data. A big issue was finding out how to take a page number in the middle of the linked list and insert it into the rear of the list. We learned that if we use a third data structure consisting of an array with pointers that point to each element in the linked list, we can move the correct node to the rear where the most recently used node should be placed.