SAN DIEGO STATE UNIVERSITY

Computer Engineering 571
Professor Aksanli
Programming Assignment #4

Ethan Nagelvoort, RedID: 821234668
Ivan Cruz, RedID: 823794173

How we're representing page tables:
To represent our page tables, we create a class called PageTable. This class has an array (of size 128) of another class called PageEntry. The PageTable class has an int that represents the page table number, either 1, 2, 3, or 4. This class has the proper getter and setter functions to access and change that array and int. There is also a function in this class that checks to see if an incoming page request is in the page table. In the beginning of the program, we get the number of page tables by looping through the text file and adding a page table to a list of page tables every time a new number in the first column of the text file appears. In every PageEntry class, we have a int to tell what page table its in, an int for the dirty bit, an int for the reference bit, an int to tell if its been used (for PER), an int for the VPN, a timer made through the built in Stopwatch class(for LRU), and a int to tell how many times its been accessed(for our own algorithm). There are necessary getter and setter functions for each of these variables.
How we're making virtual to physical address translation:
First, it is important to note that we also have a MainMem class that holds an array of 32 PageEntry classes and a reference counter int (for PER). This class has the proper getter and setter functions for those two values. It also has a function to enter a page, a function to check if the page is there, resetR() to reset reference bits, and checkPC() which is used in our own algorithm. When a new page request comes in, the program parses the text file to get the address, page table number, and operation. The address is shifted by 9 to get the VPN. First, if the main memory is not full, we check if the entry in the page table is null or not. If it is null, then that means that page is not in main memory as well and so a page fault happens. If it is not null, then that means that it is already in the main memory and so a translation happens. Once the main memory is full, then we use the check function that is found in the MainMem class to check to see if that entry is in the MainMem. If it is there, then that means that that page entry is already in its corresponding page table and in Main Memory and so a translation happens.
How we're determining when there is a need to replace a page:
First, we check if our main memory is full by using a counter. This counter is incremented every time there is a page fault and so a new entry enters the main memory. Once this counter reaches 32, then that means that the main memory is full. Then, when a new page entry request comes in, our code goes to an else statement that was attached to an if statement that detected if the previously mentioned counter was not over 32. In this else statement, we use check() (same function that was previously described) to loop through the main memory to see if that page entry is there. If it is not there, then a page fault occurs and then there is a need to replace the page. Then, depending on the algorithm chosen, the code will perform that algorithm to replace the page. It is important to note that in the program, whenever a page fault is detected there is a page fault counter that is incremented and a disk reference counter that is incremented.
How we're determining the replaced page:
Once a page fault occurs, the main memory is full, and so there is a need to replace a page, we perform one of the chosen page replacement algorithms; RAND, FIFO, LRU, PER, or our own algorithm. Our own algorithm will be described later on. In the RAND algorithm, we declare a random int that will have the value between 0 to 31. Then the program uses that int as an index to get to that page entry in the main memory's array of page entries. Then that page entry in the main memory is replaced with the incoming page entry. In the FIFO algorithm, the program uses a for loop to shift the main memory's page entry array up by one, thus making the page entry at index 0 equal the page entry at index 1. Then the page entry at index 31 is replaced by the incoming page entry. This is because the page entry at index 0 will always be the first one in and so should be the next one to be replaced. In the LRU algorithm, we look at the timers that are in every page entry in the array of page entries in the main memory. The entry with the

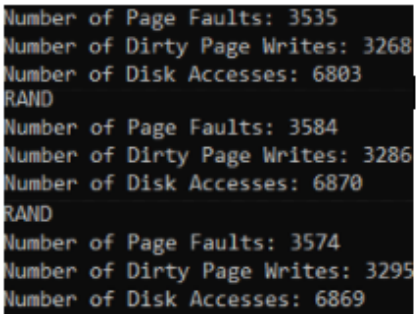
timer that holds the most time will be the page that is replaced. It is important to note that every time there is a translation, the page entry in main memory that matches the requested page entry will have its timer restart. If two timers are equal to each other, then the one with a dirty bit is replaced. If both are dirty or neither are dirty, then the one with the lower VPN is replaced. This is done through a for loop and an if-else statement. In the PER algorithm, we first set an int ind to equal 55, ind will represent the index of the page entry in main memory that needs to be replaced. First, we loop through the main memory to determine what entry has been unused through their used int. The used int turns to 1 whenever this page entry is accessed during a translation. Hence, ind will equal the index of the page entry that is unused and has the lowest VPN get replaced. If no entry is unused, then ind will equal the page entry with the lowest VPN that has its refr int (reference bit) and dirty int (dirty bit) both equal to 0. Refr is set to 1 when its corresponding page entry is accessed during a translation. Dirty is set to 1 if the page entry originally came with the W operation or there was a translation with its corresponding page entry and the requesting page entry had a W. Then, if there is no page entry that is like the one previously described, ind will equal the page entry with the lowest VPN that has its refr int equal to 0 and dirty int equal to 1. If that page entry does not exist in main memory, then ind will equal the page entry with the lowest VPN that has its refr int equal to 1 and dirty int equal to 0. Lastly, if that does not exist as well, ind will equal the page entry with the VPN that has its refr int and dirty int both equal to 1. Then we have the page entry at main memory's ind place it in its page entry array and replace that with the requested page entry. It is also important to note that every time there is a translation, the program increments a reference counter found in the main memory class since there will be a reference during translation. Once that counter gets over 200, all reference bits in the main memory and page tables get set back to 0. The counter gets set back to 0 as well. It is also important to mention that for all algorithms, we check that if the page being replaced is dirty, we increment a dirty counter and the disk reference counter.

Explanation of our replacement algorithm (additional credit):

We first declare two ints, pcount and flag. Flag is initialized to 0 and pcount is initialized to 1 + the current row number of the line in the text file being read. Then the program goes into a while loop that is dependent on flag equaling 0 and pcount being lower than the number of lines in the text file. Then we parse the text file at pcount line in the text file and get the page table number and VPN of that line. Then the program loops through main memory to find a page entry with matching page table number and VPN. If there is a match, then the pc int of the page entry in main memory is incremented. After the loop, pcount is incremented and there is an if statement to check if pcount did not go above 500+the current line being read in the text file. If it is, then flag is set to 1. Then after the while loop, we find the index of the page entry with the highest pc in main memory. We use that index to replace that page with the incoming new page entry. We also check if that page entry is dirty and increment our dirty counter and disk reference counter. Then all pc's in every page entry in the main memory is set back to 0. Overall, this algorithm requires the user to be able to tell what the future requests are to limit the number of page faults. They need to either know the next 500 requests or the last requests if there are less than 500 requests left. The page entry that is going to be accessed the least is the one that gets replaced. We came up with the number 500 by testing different numbers. We found that 500 is sufficient and so knowing the next 500 requests grants this algorithm the ability to have the best performance with the given text files.

Results:

RAND (3 test runs):



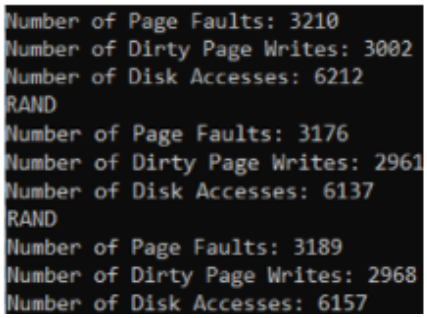
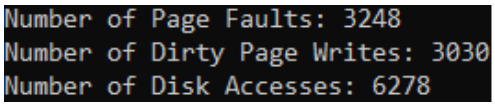


Figure 1) RAND using Data1.txt (Left) & Data2.txt (Right)

FIFO:



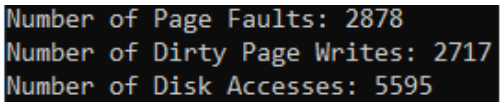
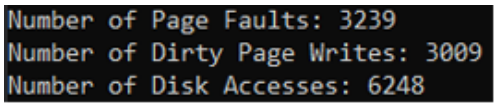


Figure 2) FIFO using Data1.txt (Left) & Data2.txt (Right)

LRU:



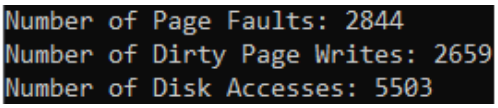
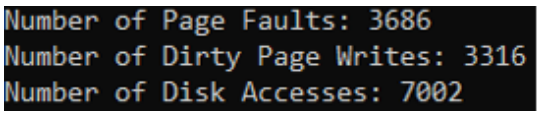


Figure 3) LRU using Data1.txt (Left) & Data2.txt (Right)

PER:



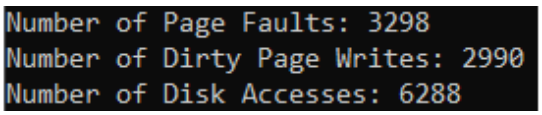
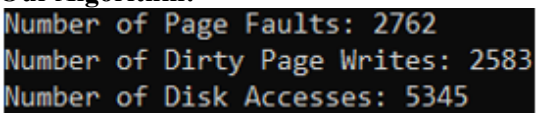


Figure 4) PER using Data1.txt (Left) & Data2.txt (Right)

Our Algorithm:



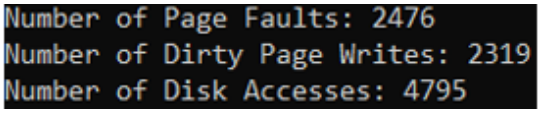


Figure 5) Our Own Algorithm using Data1.txt (Left) & Data2.txt (Right)

LRU has the best results since the page that is least recently used is not likely to be accessed in the near future. FIFO gives the next best results and then Rand gives the third best. Rand gives the third best since the page to be replaced is just random. The page to be replaced in this algorithm could result in a lot of page faults or less, it is unpredictable. Then PER gives the worst number of page faults.

Variable Comparisons with all Algorithms (RAND uses the average of the three tests we did above):

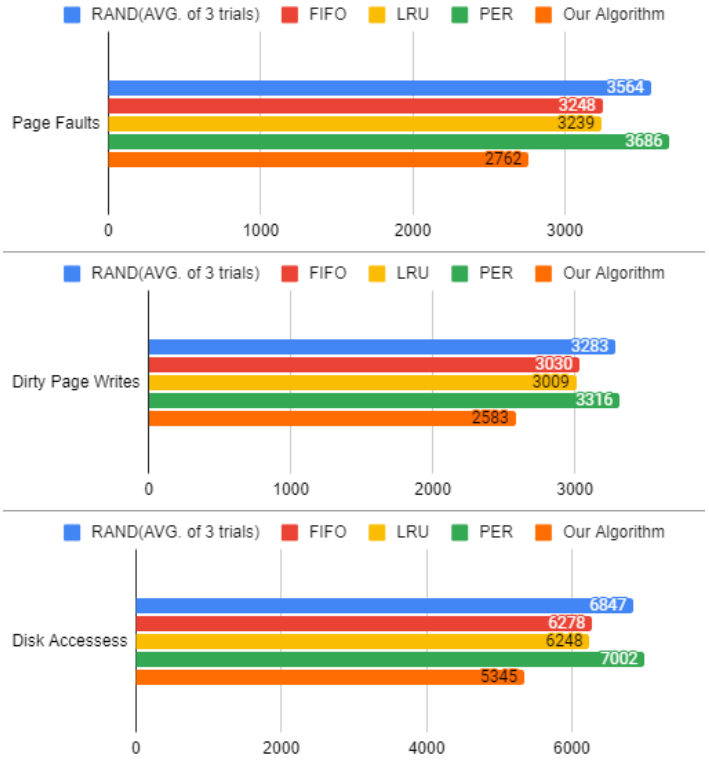


Figure 6) Bar graph comparison of Three Variables with respect to Replacement Algorithms using Data1

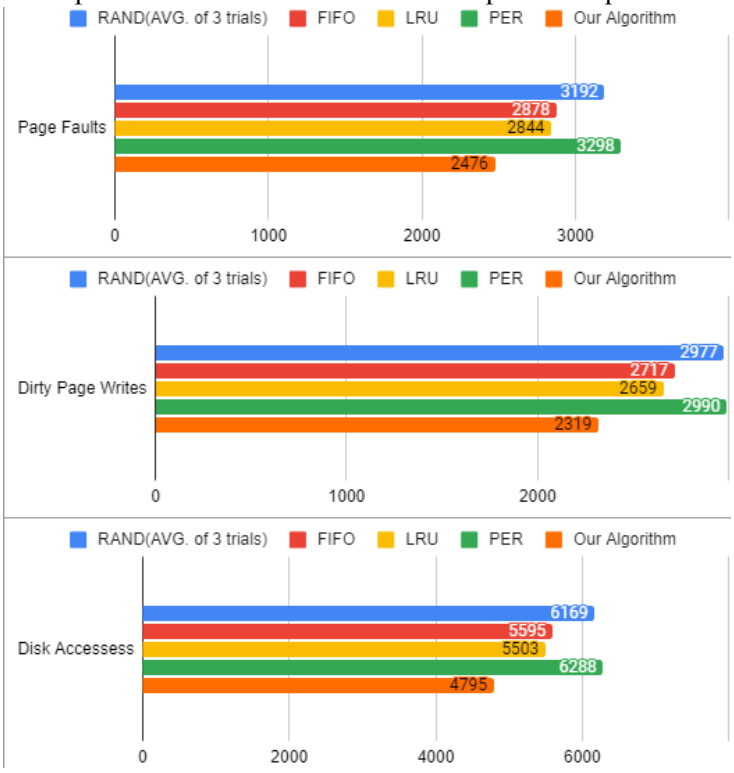


Figure 7) Bar graph comparison of Three Variables with respect to Replacement Algorithms using Data2