Louis Rozencwajg-Hays, Nicholas Gattuso, Kexian Wu

I pledge my honor that I have abided by the Stevens Honor System.

Analysis

Before we ran the tests, we expected LRU to be the algorithm that would result in the least swaps, followed by Clock, and then FIFO. LRU is the most efficient because only the most rarely used pages are swapped, leading to less faults. FIFO is the least efficient because a frequently used page could be swapped simply because it was entered early. Nevertheless, FIFO and Clock would likely have similar performance, because they have very similar implementations. We also predicted that the demand and pre-paging policies would produce very similar results. Additionally, we theorized that with a bigger page size, we would see more page faults. Because of proportional allocation, a larger page size will result in less pages per process. Therefore, having less space will result in more swapping.

After we ran the tests, we found that while our deductions about the efficiencies of the algorithms were correct, our other conclusions were wrong. For demand paging, as the page size increases, the number of faults decreases until size 8. Afterwards, for FIFO and Clock, it begins to increase again. With pre-paging, LRU and Clock slowly decline until page size 8, but then suddenly spike when the page size is 16. FIFO, however, declines until 2, then slowly rises until 8, before spiking like the others. Additionally, demand starts at a high page fault rate, while pre-paging starts at a moderately low rate. Nevertheless, all algorithms start at almost the exact same value (demand and pre-paging have different starting values).

If a random memory access trace had been supplied, it is possible that the results would change by a considerable margin. This is because by having random input, the program may be lucky or unlucky with regards to the order in which the pages are inserted, leading to noticeably more or less page faults.

Expected:

(For both demand and pre)

-LRU is most efficient, then Clock, then FIFO

-LRU is most efficient because only the most rarely used pages are swapped

-FIFO is least efficient because a frequently used page can be swapped simply because it was entered early

-FIFO and Clock should be very similar because they have a similar implementation

(Demand vs Pre)

-Demand should be more efficient with more space b/c there is more room to swap pages

-Pre should be more efficient with less space because due to proportional allocation, a larger page size will result in less pages per process. Because pre-paging requires a working set of pages, not having enough space for a working set will cause thrashing. Thrashing is the act of busy swapping pages due to a lack of space, in an attempt to conform to the pre-paging policy.

FIFO: O(n)

LRU: O(n)

Clock: O(n)

Louis Rozencwajg-Hays, Nicholas Gattuso, Kexian Wu

I pledge my honor that I have abided by the Stevens Honor System.

Analysis

What we expected:

Before we ran the tests, we expected LRU to be the algorithm that would result in the least swaps, followed by Clock, and then FIFO. LRU would be the most efficient because only the most rarely used pages are swapped out, leading to less page faults. FIFO would be the least efficient. Belady's Anomaly states that when using a FIFO page replacement policy, increasing the number of frames available can actually increase the number of page faults that occur. So, we believed when we have more pages, recently requested pages could remain at the bottom of the FIFO queue longer, which results in increased number of page faults. Nevertheless, FIFO and Clock would likely have similar performance, because they have very similar implementations. We also predicted that the demand and pre-paging policies would produce very similar results. Additionally, we theorized that with a bigger page size, we would see more page faults. Because of proportional allocation, a larger page size will result in having fewer frames per process. Therefore, having fewer frames will result in greater number of page faults.

After we ran the tests, we found that while our deductions about the efficiencies of the algorithms were correct, our other conclusions were wrong. For demand paging, as the page size increases, the number of faults decreases until size 8. Afterwards, for FIFO and Clock, it begins to increase again. With pre-paging, LRU and Clock slowly decline until page size 8, but then suddenly spike when the page size is 16. FIFO, however, declines until 2, then slowly rises until 8, before spiking like the others. Additionally, demand starts at a high page fault rate, while pre-paging starts at a moderately low rate. Nevertheless, all algorithms start at almost the exact same value (demand and pre-paging have different starting values).

What we observed:

We observed that LRU is most efficient, then Clock, then FIFO, same as what we expected. However, FIFO and Clock clearly increased page replacements when from page size 8 to page size 16 in both cases. Also, with pre-paging, we observed that LRU starts increasing for page size greater than 8. We believe the reason behind is that there are not enough frames available to satisfy its minimum working set. Basically, not having enough space for a working set will cause thrashing. Since the number of frames is equal to the size of the memory divided by the page size, increasing the page size will proportionately decrease the number of frames. In our program, we will hold 512 memory locations. If the page size is 16, we only hold 3 pages for each process, which results in busy swapping pages due to a lack of space.

Another mismatch is that demand starts at a high page fault rate, while pre-paging starts at a moderately low rate. We think this might be the case because prepaging is to predict the pages that will be needed in the near future, and page them in before they are actually requested, which will increase the chance that the memory location we try to access is held in the main memory.

Time Complexity:

LRU was the easiest one to implement. We updated last accessed time each time we accessed a page. When there is a need to swap pages, we swapped the oldest page with a new page. We simply did a linear search to find the page with the oldest timestamp. So, time complexity for this approach is O(n).

FIFO and Clock had similar complexity since they both involved making the same data structure (Linked List). Clock also had the extra overhead of keeping track of the reference bit for each page, the hands of the clock moving around in the linked list. For FIFO, when there is a need to swap pages, we will pop off the page that was loaded first (the first element of this linked list) and push a page to the end. For CLOCK, when there is a need to swap pages, if the page at the beginning of the linked list did not have its reference bit set, then that page will be popped off, a new page will be added to the end of the list, and the pointer would point to new head node. The time complexity for FIFO and Clock are both O(n) in our implementation.

Randomness:

If a random memory access trace had been supplied, it is possible that the results would change by a considerable margin. This is because by having random input, the program may be lucky or unlucky with regards to the order in which the pages are inserted, leading to noticeably more or less page faults. Overall, we believe the performance of the three algorithm will be worse. For FIFO, the performance will not be much worse.

…………………………….