Madison Solarana
CS 284 Section A
Homework #2 Report
Dr. Ercal – Fall 2011

Algorithms for Memory Management

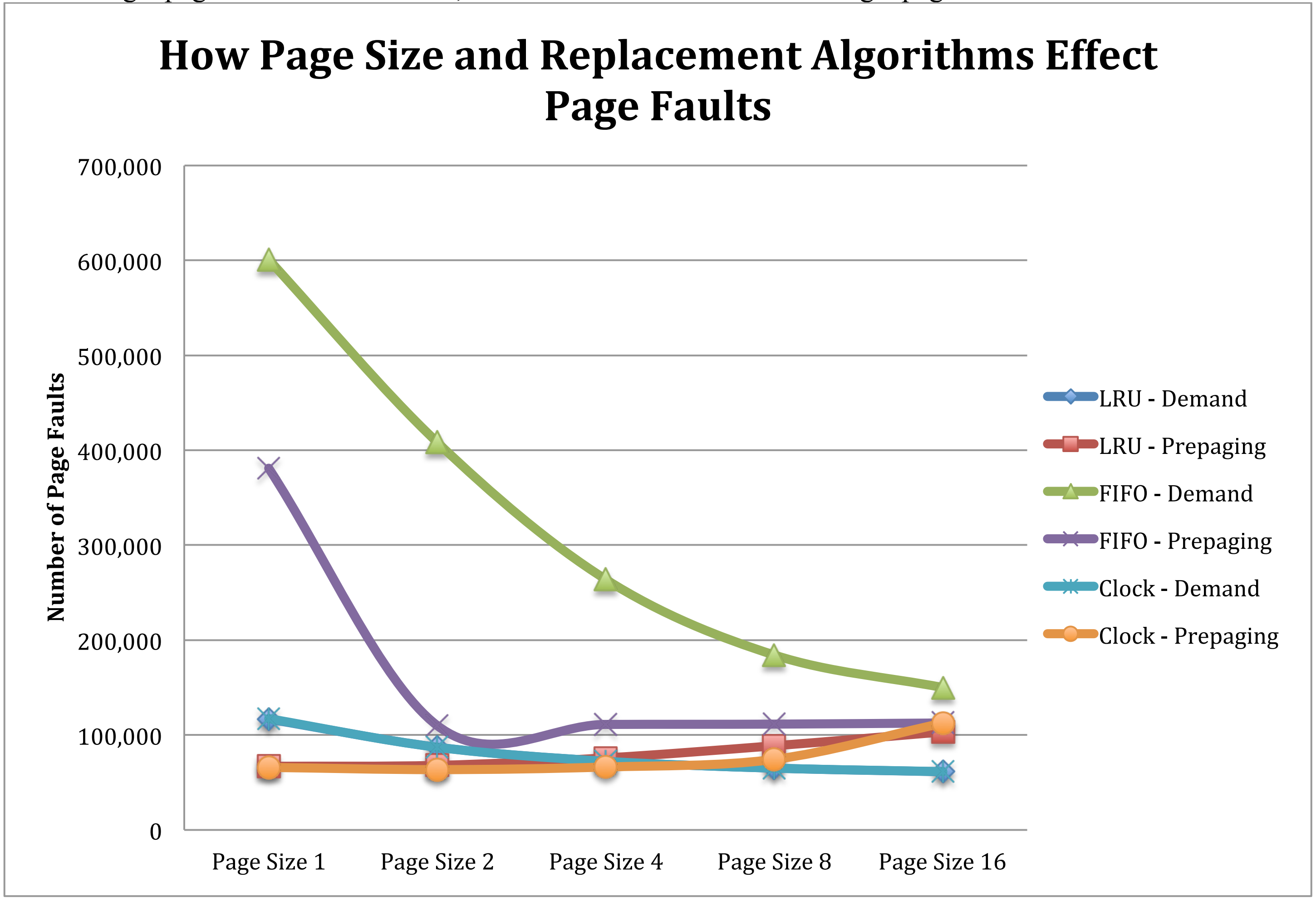
For this assignment, I was tasked to examine the effects of different memory management algorithms with different page sizes. For each algorithm, I set the different page sizes to 1, 2, 4, 8, & 16. This allowed me to see how efficiently the algorithms behaved when the page table was adjusted. I also implemented prepaging and demand paging for each algorithm to see if that helped cut down the number of page faults. The three different algorithms were last recently used, first in-first out, and clock based policy. Each algorithm has a different method of managing the pages in memory and how it retrieves pages from the page table (virtual memory).

The first algorithm that I implemented was last recently used (LRU). LRU looks at pages that have been used in a short time period and then makes the decision of whether or not to remove the page from memory and insert a new one in its place. LRU makes the assumption that pages that were used recently and/or heavily during the last clock cycle are going to be used again. Therefore, when it looks for pages to remove from memory, it checks to see how much time has passed since the page in question was used. If it finds one that has been used least, it is replaced with the page that was requested. If prepaging was enabled, it would removed the least used two pages and bring in the requested page and the next sequential page for that program if global page replacement is not active. This algorithm, while simple to implement, has a drawback. If the requested page is not found in memory, the algorithm must check all the pages that currently reside in memory. This is a costly operation due to the size of memory in most modern computers.

The second algorithm that I implemented was first in-first out (FIFO). FIFO is the simplest algorithm out of the three that I examined. FIFO operates like a queue, each successive page is added on to the next and when it comes time for a page swap, the operating system removes the first page from the front and adds the new one to the back. It has very little overhead, but performs really poorly since it doesn't care about page use, just the order of the queue. Prepaging seems to help the algorithm since it adds a small amount of intelligence, but FIFO still doesn't have any of the complexity of the other two algorithms that make them better.

The third and final algorithm that I implemented was clock based policy (Clock). The clock algorithm is like the FIFO algorithm, but smarter and more efficient. The clock algorithm keeps track of all the pages in memory by using a list. All the pages in the list have a single bit flag that can be either 0 or 1. The clock rotates around the list and decrements the flag bits. Once a zero is found, that page is replaced and its flag bit is set to one. Although the clock algorithm is smarter than FIFO, it still can take a lot of time just like LRU. It has to take at least two clock cycles (relative to a specific page) to replace a page since the flag bit has to be decremented and then reexamined the next cycle.

The results for my simulation can be found on the next page, though I will begin talking about them here. At first I didn't know what to expect in regards to LRU vs. Clock, but I did know beforehand that FIFO would be the worst algorithm since it is not intelligent. FIFO performed well as soon as the page size increased, but prepaging seemed to help a lot even with the smaller page sizes as it allowed for the possibility of more relevant pages to be added in. As for LRU, it performed decently with demand paging, but worked a lot better with demand paging when the page size was increased. When it was prepaged, it worked well when the page size was smaller. Compared to Clock, LRU did almost the same. For both algorithms, the number of page faults were so close together that on the graph, the Clock demand line covers up the LRU demand line perfectly. Also, the Clock prepaging line almost covers up the LRU prepaging line, but there was a small difference when the page size was eight. Overall, Clock seemed to be the best performer, with LRU very close to the same results. I also noticed that prepaging seems to work better when the page sizes are small and that demand paging works a lot better when the page sizes are large. This makes sense because when page sizes are smaller, there are not a lot of memory addresses on the page, so when prepaging brings in the next page, this increases the chance of avoiding a page fault. This also explains why prepaging does not work so well in large page size situations because there are so many memory addresses on the page that prepaging does very little to increase the chance of avoiding a page fault. In some cases, this increases the chance of having a page fault.



It is important to note that trying to compute which algorithm is best without a standard trace would be very difficult. Since this simulation used a fixed trace, it ensured that each algorithm had the same amount of data to work on and memory was consistent, making the results fair. In the future, it would be interesting to see how some of the variations of LRU, FIFO, and Clock work since computer scientists have been working on making them better. For example, there is a slightly different twist to LRU called NRU (not recently used). This algorithm works by dividing pages into four different classes and then makes decisions every clock cycle based on what class the page it is looking at is in. There is also a Clock variant that is based off of LRU that maintains a list of recently swapped pages and uses the list to change its behavior and frequency of swapping pages. There even is a FIFO variant called Second-Chance that operates like FIFO, but instead of immediately swapping out the first page in the queue, it checks a reference bit. If the bit is set, it is cleared and that page is moved to the back of the queue and then the algorithm begins again by checking the newly promoted front page in the queue.