Brett Heinkel
CS 3800
Section 1B
April 6, 2017

Assignment #2 Report

Hypothesis and Complexity vs. Performance

The first in, first out (FIFO) algorithm is considered to be one of the more simpler page replacement algorithms, and the intended behavior can be inferred from the name: the operating system keeps track of all the pages in memory in a queue, with the most recent arrival at the back, and the oldest arrival in front. This creates issues when the page that was replaced needs to be added again soon, which happens often with the FIFO implementation. While FIFO is cheap and intuitive, it performs poorly in practical application and will likely create the most page faults.

The clock algorithm keeps a circular list of pages in memory, with the "hand" (iterator) pointing to the last examined page frame in the list. When a page fault occurs and no empty frames exist, then the use bit is inspected at the hand's location. If usebit is 0, the new page is put in place of the page the "hand" points to. Otherwise, the use bit is cleared, then the clock hand is incremented and the process is repeated until a page is replaced. Although the complexity of the algorithm may lend to an increase in time for processing the number of page faults, the method which is used in replacing pages in memory should lead to a lower number of page replacements when compared to the FIFO algorithm.

The least recently used (LRU) algorithm works on the idea that pages that have been most heavily used in the past few instructions are most likely to be used heavily in the next few instructions too. This factor of the LRU algorithm is its greatest advantage, along with being similar enough to the FIFO implementation that it is not incredibly complex to implement for page replacement. LRU is expected to have the least number of page faults.

Demand vs. Prepaging

While demand paging is advantageous in its simplicity, prepaging offers the benefit of a largely decreased initial page fault count at the cose of greater complexity. By bringing in more pages than necessary, specifically the next subsequent page, the number of page faults at process start time is greatly decreased. This acceleration in page utilization also results in a distinct increase in page faults towards the end of execution. This difference is illustrated in Figure 1, which visualizes page size versus page faults for each replacement algorithm, both with and without prepaging.

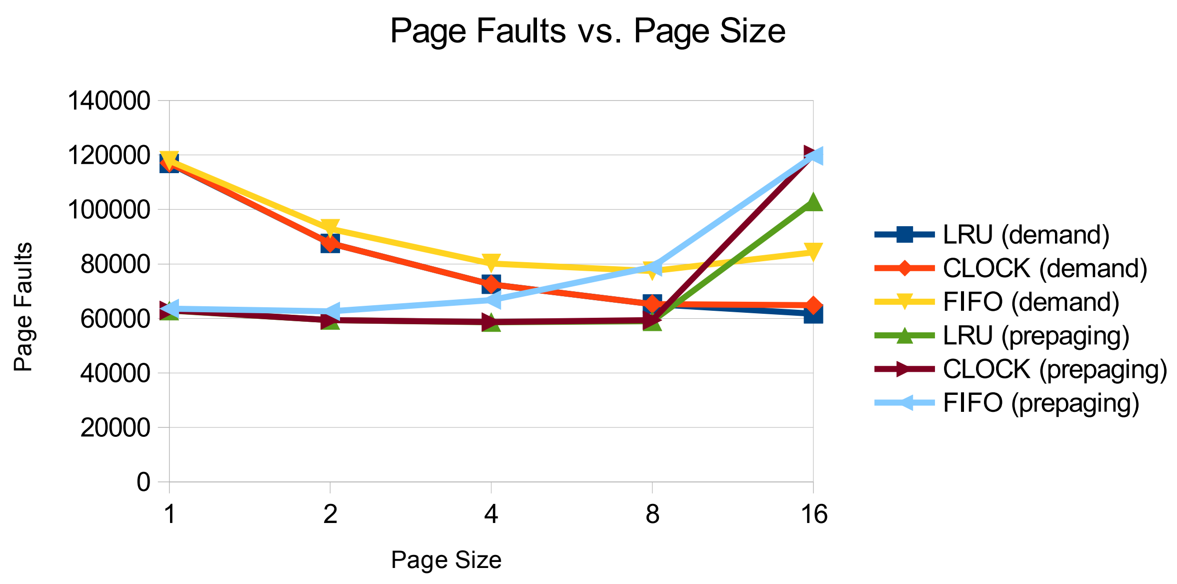

Figure 1

Page size	LRU (demand)	CLOCK (demand)	FIFO (demand)	LRU (prepaging)	CLOCK (prepaging)	FIFO (prepaging)
1	116886	117042	117879	62893	63004	63656
2	87573	87622	92894	59410	59401	63618
4	72551	72539	80154	58619	58697	66751
8	65273	65320	77382	58958	59353	78876
16	61705	64824	84232	102892	120151	119663

Results

The results were consistent with my hypotheses.

Random Trace vs. Supplied Trace

If the provided program trace had contained random memory accesses as opposed to the sort of predictable, systematic data that we were supplied with, the number of page faults would most likely have decreased, particularly with prepaging. This is because there are several lines in the programtrace file where a page is called for a specific program and then the same program makes a request for the page immediately after. In this case, prepaging allows the next available page to be loaded into memory with the page requested, and since the next request is often the next page in long term memory, the number of page faults is decreased.