Ben Merrick
Professor Thain
Operating Systems
April 24th, 2020

Project05 Report

Section01: Experiment Description

The purpose of this experiment was to analyze three core statistics related to page faults across three separate page replacement policies and four separate programs. The three main statistics were the total number page faults, disk reads, and disk writes for a given replacement policy and program. The replacement policies were 1) random which replaced a random page, 2) fifo which always replaced the first placed page in the table, and then 3) custom algorithm which replaced the least recently written to. The four programs were alpha, beta, delta, and gamma which are described in detail in Section04. A sample test would be as follows for a test of 100 pages, 10 frames, the random replacement policy, and the alpha program: `./virtmem 100 10 rand alpha.` This exact line was repeated for 10-100 frames in increments of 10 frames for random, then fifo, then custom. After all of those policies were completed, I would move onto beta, gamme, and delta. Below are my results of the experiment as performed on student04.cse.nd.edu.

Section02: Analysis of Custom Page Replacement Policy

For my custom page replacement policy, I attempted to implement a version of the least recently used (lru) policy. Except, it ended up following a least recently written to pattern. First, I assign an extra bit to the frame struct called “lru” to represent the frame’s status. This bit is either a 1 or a 0. As for the function itself, the philosophy is very straightforward. Essentially, the function loops through all of the frames using a counter. It checks to see if we have found the first instance of the least recently written to frame as indicated by a 0 and then we return that frame. If the current frame is not equal to 0, then we set the lru bit equal to 0 and increase the counter. Normally, this loop would expire once we reach the maximum number of frames. However, an if statement at the bottom resets the loop and starts over. But because we set each frame to 0 after it is checked, we simply return the first frame if we are unable to find the least recently written to. The final piece of the policy is where we actually set the lru bit. This is done in the page fault handler where we check to see if we only need to write the bit. Here, the lru bit is set to 1. Since we are adding the write bit to the current frame, that means it is no longer the least recently written to frame. Overall, the goal of the policy is an attempt at determining the least recently written to frame and using that frame as the replacement frame.

Section03: Graphs and Data

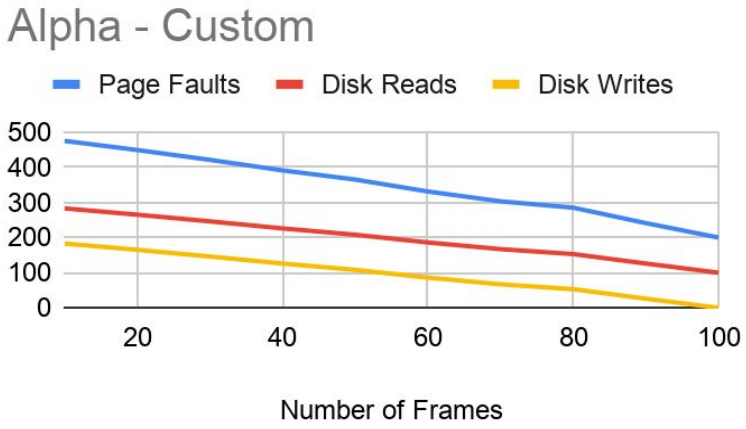

Figure1: Alpha Custom Statistics

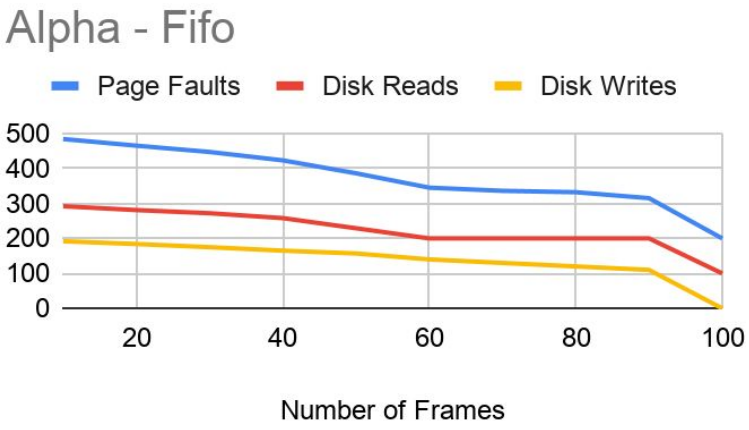

Figure2: Alpha Fifo Statistics

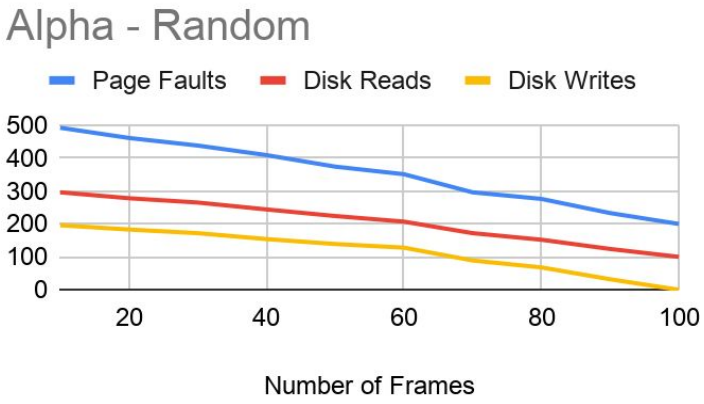

Figure3: Alpha Random Statistics

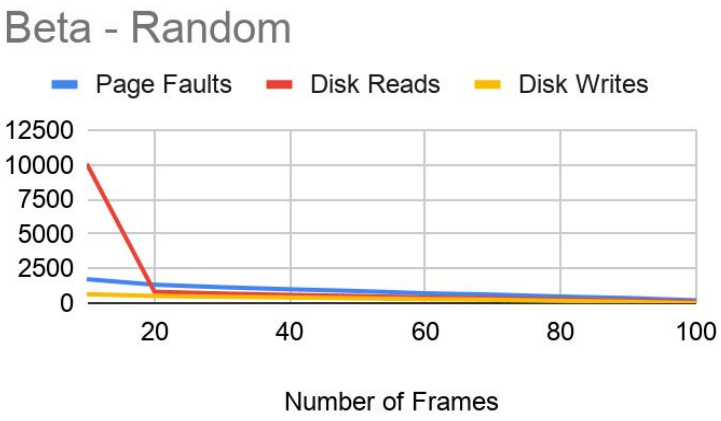

Figure4: Beta Random Statistics

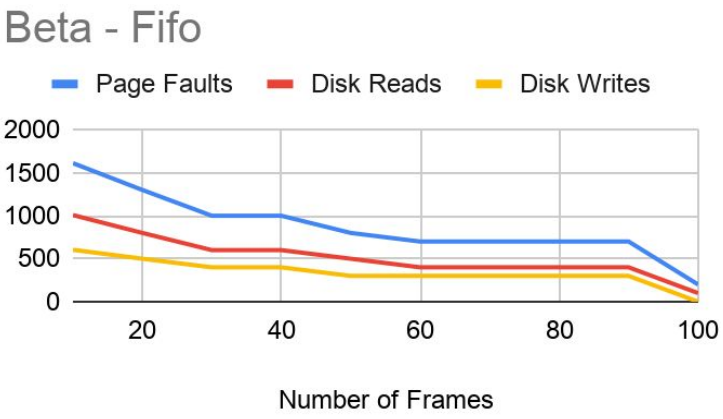

Figure5: Beta Fifo Statistics

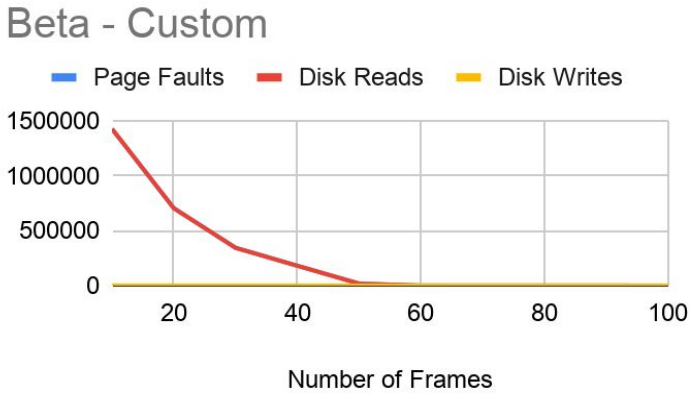

Figure6: Beta Custom Statistics

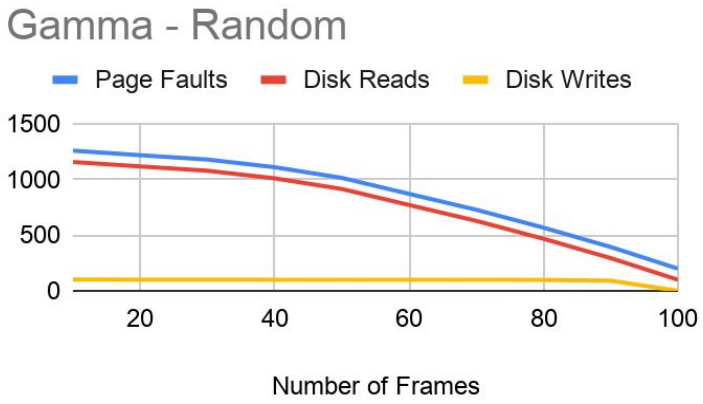

Figure7: Gamma Random Statistics

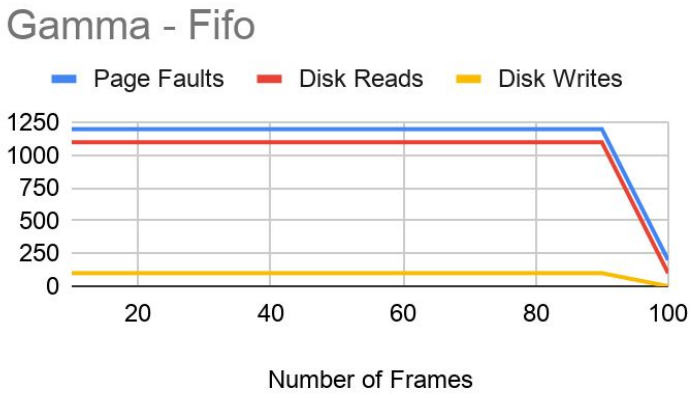

Figure8: Gamma Fifo Statistics

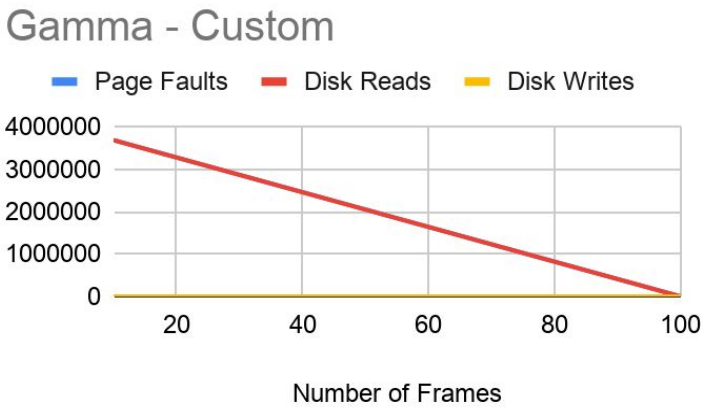

Figure9: Gamme Custom Statistics

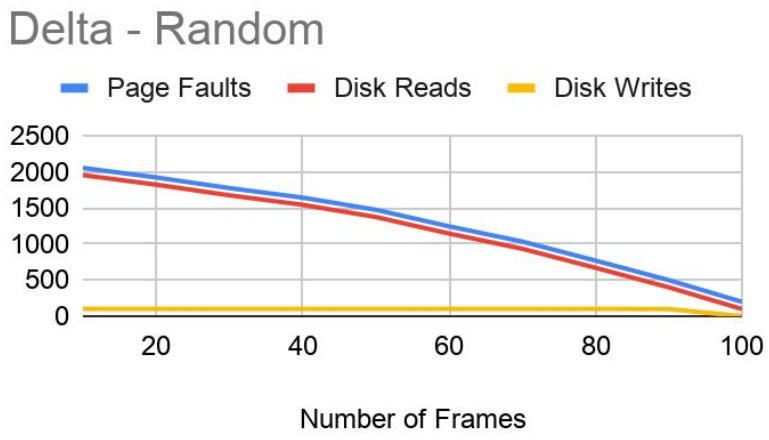

Figure10: Delta Random Statistics

Figure11: Delta Fifo Statistics

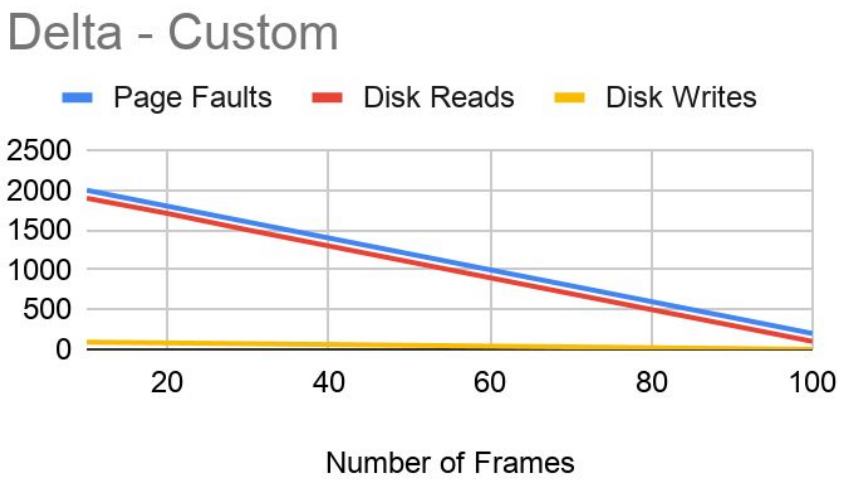

Figure12: Delta Custom Statistics

Section04: Concluding Analysis

Alpha:

To begin, the function initializes the internal random buffer with a series of 38290 random 48 bit values and initializes an array of data to all 0s. Next, the function begins it's main loop which loops 100 times and contains another nested for loop which also loops 100 times. Within this nested for loop, we use a combination of random numbers and modulus to determine the spot at which we assign another random number. Not all spots are assigned values. After reviewing the final array, it is clear to see that the function assigns random numbers in clusters of 25. There will be a series of 0s interrupted by a series of 25 data values which result in "empty space" in the array.

With alpha, my custom function performed the best out of the three. Initially, all of the functions are seen to act about the same as we are simply assigning the data to 0. However, when we dive into the random aspect creating random clusters of 25, custom begins to take over. Fifo levels off as it can not deal with the random data points in the array. The data is being read, and written to in these clusters making it easier for the least recently written to policy to access these points.

Beta:

The beta function begins by initializing the internal random buffer with a series of 4856 random 48 bit numbers. It then loops through each entry in the array and assigns it a random value leaving no spot empty. Then, we call qsort() on the data to sort the array from lowest to highest. This can take a varying amount of time since the random numbers can play into the sorting, but it statistically should workout to be the same. After the sorting is complete and the data is in order from lowest, the data is summed and returned.

With beta, my random and fifo functions performed about the same while custom performed terrible for a frame count including and below 50 frames. Rand initially struggled with a lot of disk reads at a low number of frames because there was so much data to go through and we needed to constantly read it into the disk. Custom eventually performed well around 50 and above frames since prior to, the lower number of pages led to more disk reads, and less disk writes. Fifo, improved consistently over time proving to be the best for overall performance.

Gamma:

The gamma function begins by iterating up to half the size of the length of data. In this loop, it assigns one array equivalent to data a value and another array equivalent to the first half of data a value. Regardless, the array equivalent to the original data array only has half of the values assigned since the loop iterates up to length/2. Then, a set of nested for loops adds the values of both arrays at the same index together, and repeats 10 times

With gamma, random performed the best overall. The number of disk writes was so low since we were dealing with about half of the overall data array. Fifo maintained a constant performance regardless of the number of frames up until an even number of pages and frames. This is most likely due to the fact that frame ordering does not matter for this specific program. Once again, custom performed terribly overall for all frame counts. Since the overall disk writes for this function is so low, we struggled to find the least recently written to frame since we were never writing anything.

Delta:

The delta function begins by iterating through the length of the data array and assigning the current spot a value related to index and a constant. Then, we reach an outer for loop containing two inner for loops, not nested. The first inner for loop adds up all the values in the array from first to last and the next inner for loop continues to sum from last to first. This is done 10 times by the outer for loop.

With delta, custom performed the best in a decreasing linear fashion. Once again, the overall disk writes is so low for this entire delta program. However, as compared to last time, since we are dealing with the data in a fashion that resembles walking up and down a ladder, we are finding less page faults and disk reads due to the structure of the code. Random decreases in an almost constant linear fashion and fifo levels off at about 40 frames until 100 frames. This is probably due to the fact that the frames towards the middle of the data array are closer to each other and can be accessed easier. The order doesn't matter as much anymore.