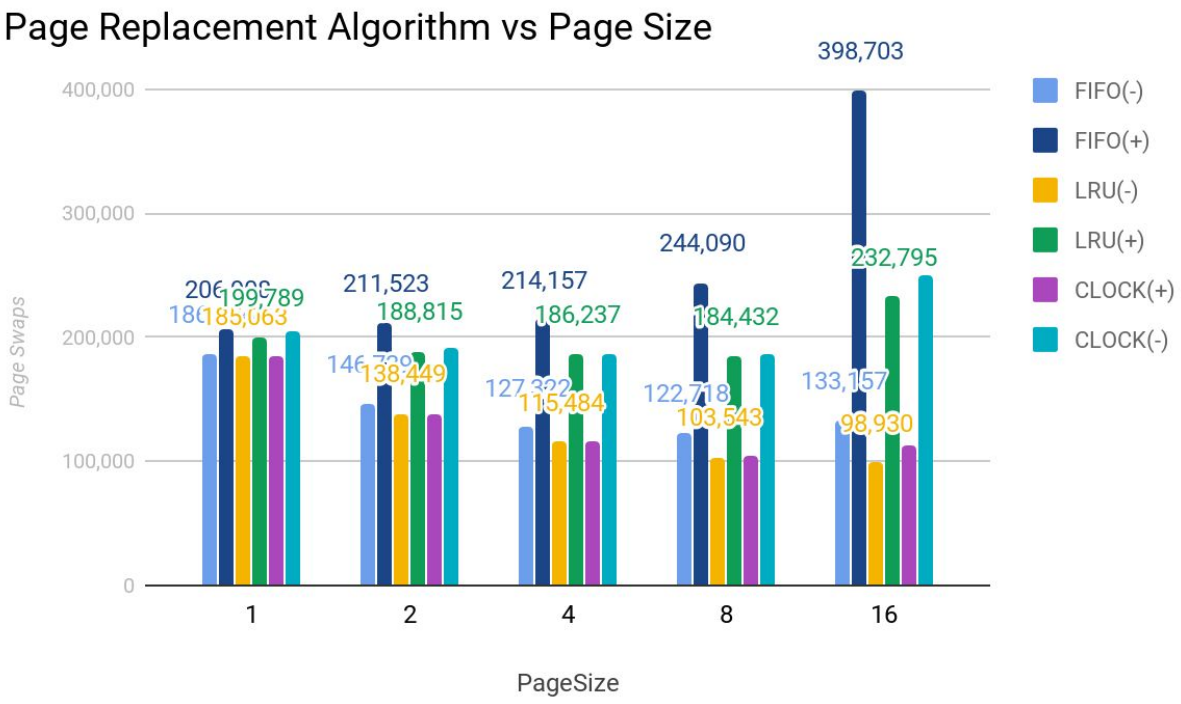
Donald Tang
Tianwei Liu

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

Data

	FIFO (-)	FIFO (+)	LRU (-)	LRU (+)	CLOCK (-)	CLOCK (+)
1	186,384	206,908	185,063	199,789	184,892	204,278
2	146,739	211,523	138,449	188,815	138,334	190,743
4	127,322	214,157	115,484	186,237	115,446	187,139
8	122,718	244,090	103,543	184,432	104,454	186,620
16	133,157	398,703	98,930	232,795	112,123	250,859



What We Expected VS. What We Actually See

We had expected that prepaging would result in less page swaps because the intention of prepaging is to decrease the amount of page faults. However, looking at the data, we can see that prepaging actually results in more page swaps.

We also expected that running the algorithms with bigger page sizes would result in less swaps because bigger page sizes mean that there are more memory addresses mapped to each page, so the probability of the next memory location requested already being in memory would be higher. Based on the data, it is somewhat true, as the number of page swaps decreased as we increased the page size from 1 to 8, but then the number of page swaps increased once we used a page size of 16, probably because there are fewer pages able to fit in main memory, so too big of a page size would result in less memory locations being able to fit in main memory. Thus, from the data, it seems apparent that there are less page swaps when a page size of 8 is used, a page size that is not small or big, but in between.

Relative Complexity of Programming Each Algorithm

FIFO

Implementing FIFO was relatively simple, since the first pages that go into main memory are the ones that get replaced first. Thus, if the main memory is full, we can simply remove the first frame in the main memory and push the new page to the back of the main memory. This is easily done since we are using deque as main memory, popping the front and pushing to the back to perform a page swap.

LRU

Implementing LRU was quite tricky, as the idea employs the idea of using time to track which page is the least recently used, and then swap that page out if there is no free frames in main memory. However, in our assignment, main memory is initialized by giving each process “N” equal amount of frames and then pushing N pages from each process’ page table into main memory one at a time, meaning that after initialization, the least recently used pages will be at the beginning of main memory, whereas the most recently used pages are at the end of main memory. This approach is similar to use a priority heap, however since we are using a deque the most recently used will be in the back and the least recently used will be in the front of the deque. Then, we can pop the front (remove the first frame of main memory) since it will always be the least recently used if we need to replace a page. If a page is already in memory, we can find the page in memory and push it to the back of main memory so that we maintain the order of main memory from least recently used to most recently used.

Clock

Implementing clock was not as tricky as LRU, as it just required checking each page’s reference bit for a value of 0, and if not, checking the next frame in memory, in a clock like structure. The trickiest part was keeping track of the index of the clock hand in the frame table so we reset the clock hand back to the beginning of the frame table if we reach the end. Also, we had to keep

start of where the clock hand started so that we know when we have made a full lap around the frame table searching for a reference bit of 0.

How The Data Might Have Changed If a Completely Different Memory Access Trace Was Provided

The data could have easily changed if a completely different memory access trace was provided because the number of page swaps depend on if the memory locations being requested are already in memory or not. Thus, a memory access trace that requests memory locations that are clustered together will result in less page swaps since the memory locations will most likely already be in memory already.