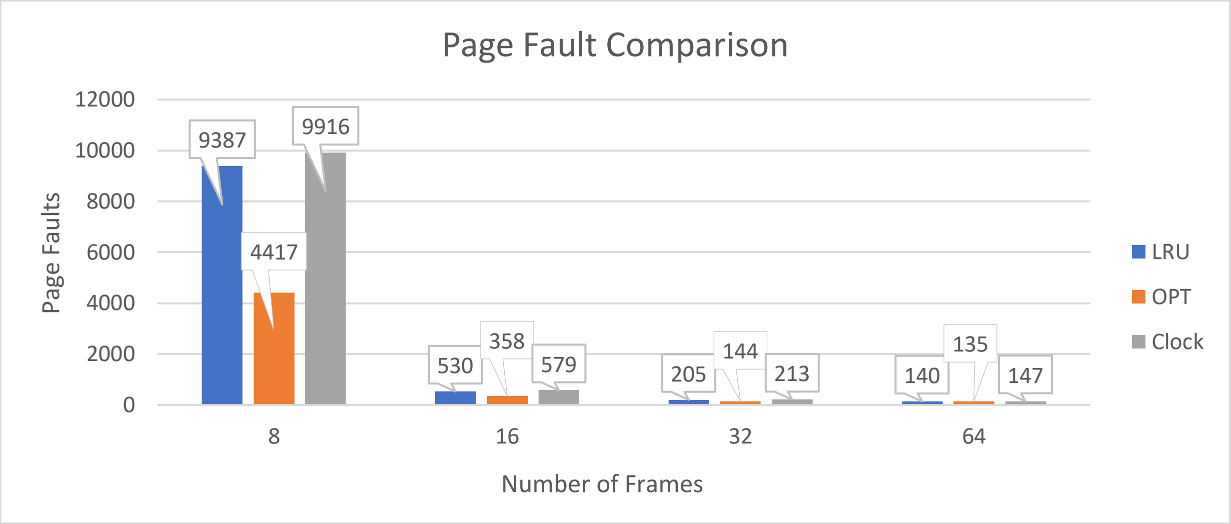
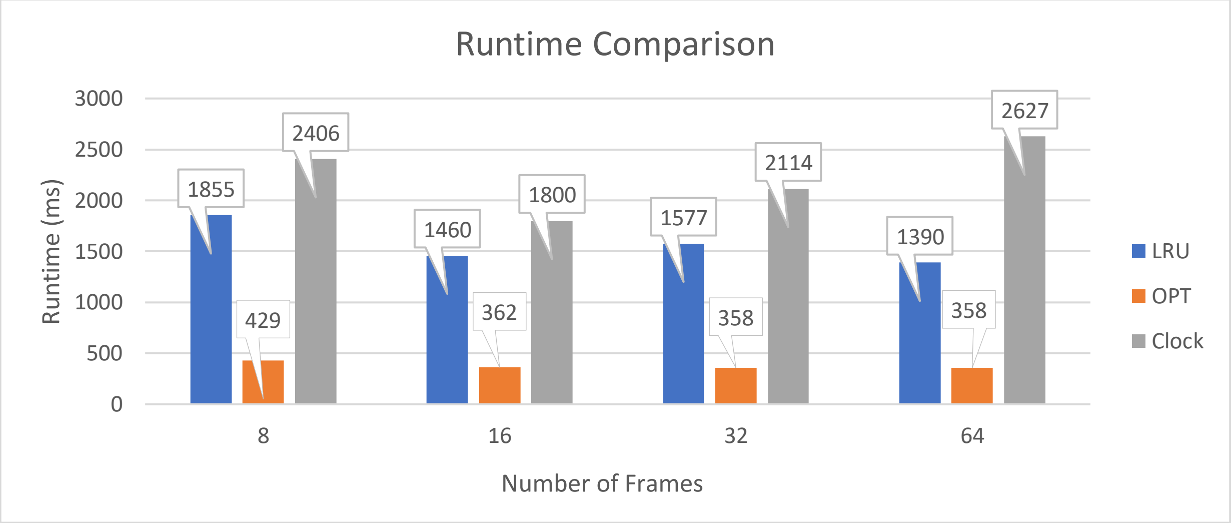
Steven Montalbano
CS 1550
Project 3 Report

Part 1: For each of your three algorithm implementations, describe in a document the resulting page fault statistics for **8, 16, 32, and 64 frames**. Use this information to determine which algorithm you think might be most appropriate for use in an actual operating system. Use OPT as the baseline for your comparisons.

The following graphs display my simulations results comparing the amount of page faults and runtime against themselves on varying amount of frames in physical memory. The tests were ran using the swim.trace file provided.





From these results, I believe that Least Recently Used is the most appropriate page replacement algorithm to be used in an actual operating system. Using OPT as a baseline, LRU was the best performing algorithm. LRU had less page faults and a smaller runtime compared to the Clock implementation of the Second Chance algorithm. This pattern was observed across all four tests using 8, 16, 32, and 64 frames of physical memory.

Part 2: For Second Chance, with the three traces and varying the total number of frames from 2 to 100, determine if there are any instances of Belady's anomaly. Discuss in your writeup.

To search for instances of Belady's anomaly, I wrote a script that ran the three files (gcc.trace, swim.trace, gzip.trace) in a for loop that used 2-100 as the frames argument. I then output the page faults to a text file where I was able to bring the results into Excel. A simple macro was written that scanned down the results and highlighted all instances of page faults reported being greater than that of the previous test which would have 1 less frame. I found 23 instances of Belady's anomaly when the frames were 40, 50, 60, 70, 73, 75-78, 82, 84-88, 90-94, and 97-100. I also noticed that the first occurrence of Belady's anomaly didn't happen until 40 frames.

Part 3: Discuss the implementation and runtime of the OPT algorithm

For my optimal page replacement algorithm, the main algorithm logic runs in a while loop, similarly to the other algorithms. However, OPT is different from the others in that it calls a preprocess() method before entering the while loop that reads the trace file into memory and creates and initializes the future hash map of type <Integer, LinkedList<Integer>>. The purpose for this is because in order for OPT to select the page table entry (PTE) whose next access is the farthest in the future, the algorithm must have a perfect knowledge of each future memory access from the OS. The LinkedList in the future hash map contains integers representing the line of the trace file where that page address (the map key) is accessed. During the first readthrough of the file, it also creates MemoryAccess objects for each instruction read so that the algorithm does not have to read the file again since I/O is one of the slowest operations performed.

The actual algorithm reads the MemoryAccesses from an ArrayList on each loop iteration. It reads the instruction's address and mode (load or store) and then gets that PTE from the pageTable hash map. The algorithm then checks if that PTE is currently in RAM, if so it updates the last accessed time for the PTE, which is used when selecting a victim page. If the PTE is not in RAM, it checks if there are any vacancies which would allow it to be inserted without an eviction. If an eviction is needed, the algorithm calls a helper method locateVictim().

Inside of locateVictim(), an iterator is created to easily access the contents of RAM. The method loops through RAM and gets the address of the current PTE and looks that address up in the future hash map. I use the LinkedList getFirst() method to return the next occurrence of that PTE, since LinkedLists follow insertion order, the list was ordered ascendingly as the preprocess() method went down the file line by line. Once an instruction is read, it removes the first entry from the future list. In the event that a PTE's future list is empty, it means that it can be removed immediately. However, a case can occur when multiple PTE's have an empty future. In this case, the algorithm resorts back to LRU. When an empty future is found, that PTE gets added to a Priority Queue acting as a MinHeap sorted by the last access field of the PTE class. So if one or more PTE’s have an empty future the victim selection is as easy as polling the highest priority element of the queue. If no PTE’s have an empty future, a for loop is used to look at the head of the LinkedList of every PTE in RAM and determine which one has the greatest next access line number, meaning this PTE is the furthest first access in the future.

The OPT algorithm performed the fastest out of all three that were implemented in the project. Being that the trace files are a constant size, the overhead of the preProcess() method is a constant factor. The following methods are the ones that I’ve used for my data structures:

Method	PQ: offer()	PQ: poll()	HashMap: Put()	HashMap: Get()	ArrayList: Get()	LinkedList: getFirst()	LinkedList: remove()
Complexity	O(logn)	O(logn)	O(1)	O(1)	O(n)	O(1)	O(1)

In most cases, my methods are constant or logarithmic time in the case where multiple PTE’s have an empty future and the priority queue is needed. The locateVictim() method also runs in O(n) time, where n is the number of RAM frames. Another bottleneck is reading from the MemoryAccesses ArrayList since that operation runs in O(n) time, where n would be the number of lines in the trace file. Since these methods and data structures I’ve chosen run very efficiently, it’s clear why the performance of OPT was a fraction of the others runtime in all cases.