

Diana Kocsis

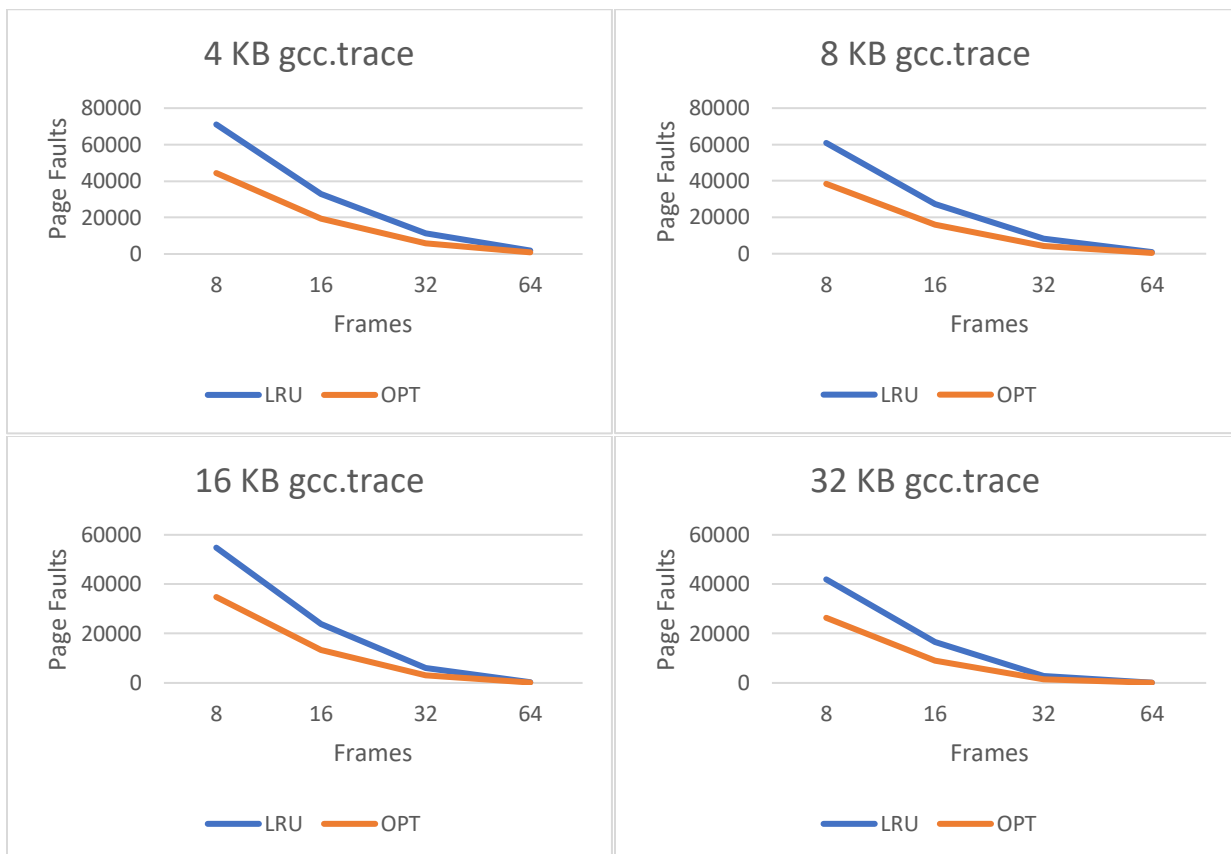
Sherif Khattab

Introduction to Operating Systems

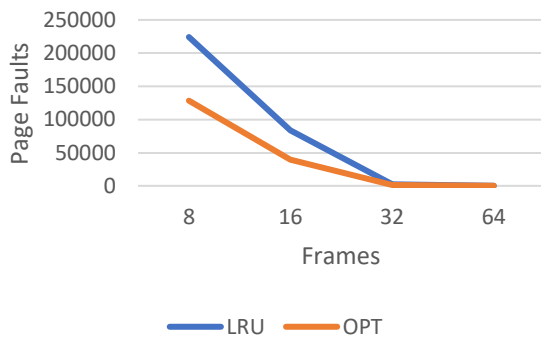
20 July 2020

Project 3

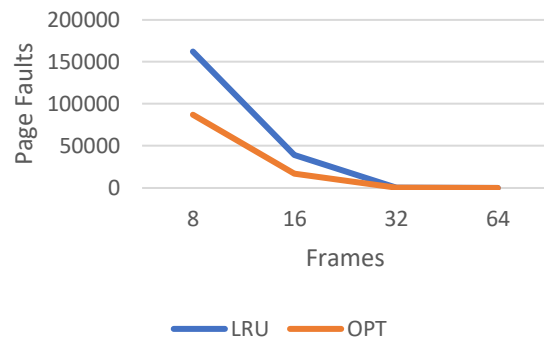
Part 1: The following graphs plot the number of page faults versus the number of frames while running the LRU page replacement algorithm and the OPT page replacement algorithm. Each graph displays the results when the algorithm is run with a specific page size and trace. By analyzing the graphs, it is found that the number of frames and the number of page faults have an inverse relationship. As the number of frames increases, the amount of page faults decreases until some point, and then the curve starts to flatten. Page size also has an effect on the number of page faults. The larger the page size, the less page faults until a certain point. The difference between the LRU algorithm and the OPT algorithm is that the LRU algorithm causes more page faults, as its line lies just above the OPT line.



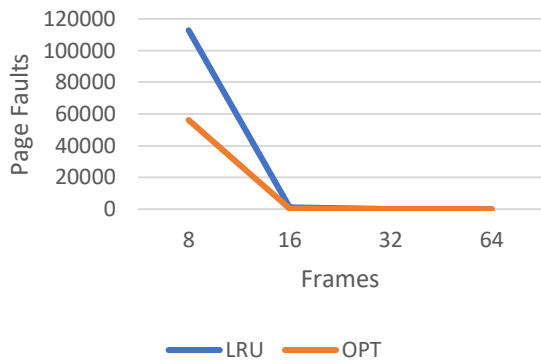
4 KB go.trace



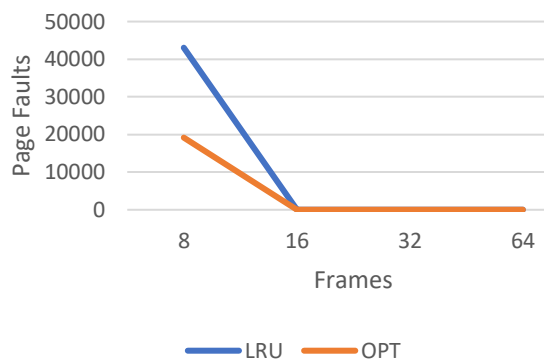
8 KB go.trace



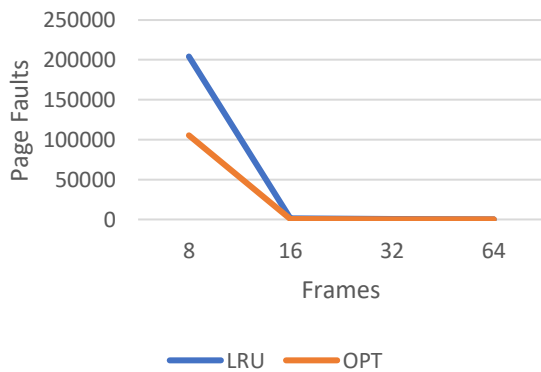
16 KB go.trace



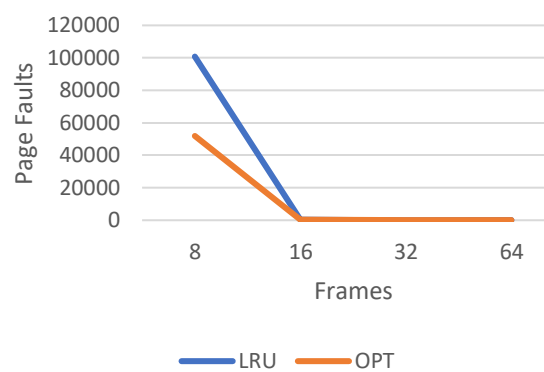
32 KB go.trace

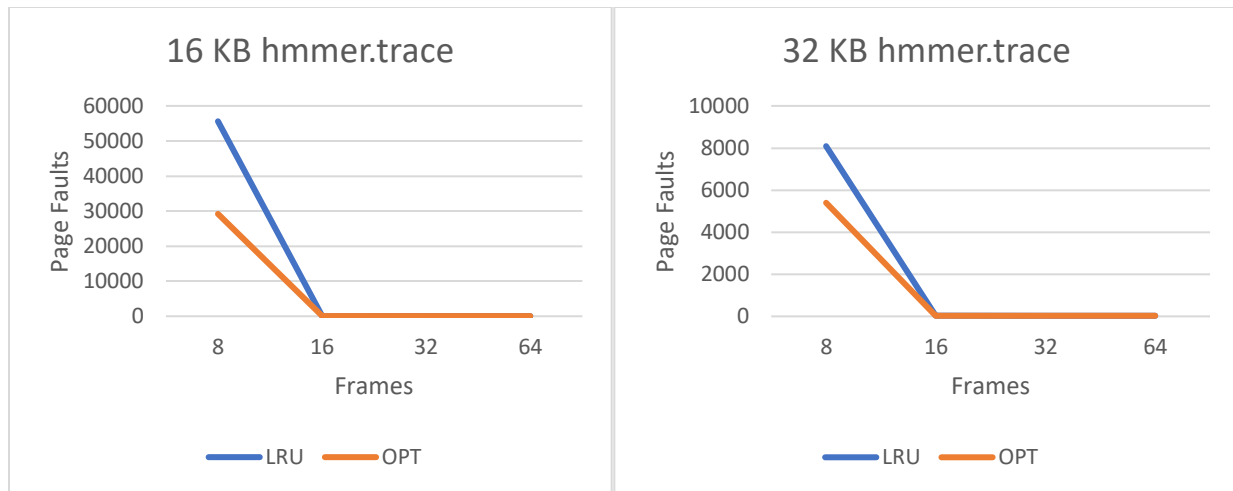


4 KB hmmer.trace



8 KB hmmer.trace





Part 2: The optimal algorithm was implemented by tracing through the program twice: once to generate the reference trace and once more to apply the optimal algorithm. When pre-traversing the file in the beginning, each page address with its access type (store or load) and access number (the order at which it is being accessed) is hashed and placed into a separate chaining hash table. The runtime of putting the page addresses into a hash symbol table is $O(n)$, where n is the amount of page addresses in the trace file. Inserting the page address into the symbol table is constant time because each linked list in the hash table is a doubly linked list with a tail pointer, so we can easily insert at the tail.

After this, the file is traced again one line at a time with $O(n)$ runtime. For each page address in the file, it searches through the frames already in physical memory looking for the same page address. This is $O(n)$ runtime where n is the number of frames already in memory. If the page address is found, the algorithm updates the access count to represent when it was last accessed. In addition, if the access type is "s", it will update the page address's access type to "s" in memory. The "s" is representative of the dirty bit in a page table, and whenever it is removed from memory, it is a notifier that it must write to the disk.

On the other hand, if the page address is found in memory and memory is not full, it will simply insert the new page address with its access type and access count number into a frame in memory at $O(1)$ runtime, since physical memory is also a doubly linked list with a tail pointer.

However, if the page address was not found in memory and memory is currently full, it needs to decide which page address in memory it will replace. This is decided by which page address currently in memory will be accessed furthest away in the future. To do this, it will traverse memory again with $O(n)$ runtime where n is the number of frames currently in memory and hash each page address to find its location in the hash table. By doing this, it will grab the access count of the next time that page address is used. This works because each page address with its type and access number is deleted from the hash table after it is processed, and then it is placed into memory. This is beneficiary so we do not have to traverse a very long linked list at an index of the hash table. We then find the greatest access count number and delete that page address from memory. However, if a page address is not used at all in the future, it will return an access count of -1. This means that we should delete this page address from memory instead. If there are multiple page addresses that are never used in the future,

then we have a tie. To determine tie breakers, we must find the least recently used page address and delete it from memory. To do this, we place them in another linked list called the tieList. With these frames (which are already in memory), we traverse them again with $O(n)$ runtime looking for the lowest access count number. Whichever has the lowest number is the one that is least recently used and the one that will be replaced by the incoming page address.