Gordon Lu

Professor Daniel Mosse

CS 1550

Due: 06 April 2020

CS 1550 Project 3 Report:

In this report, I will highlight the results of each of the Page Replacement algorithms of sample trace files and compare performance of each algorithm. I will also determine if there are any instances of Belady's Anomaly within the sample traces using Second Chance. Additionally, I will describe the implementation I have taken to simulate the Optimal Page Replacement Algorithm, as well as its runtime. Project 3 required us to simulate the following algorithms:

1) **Optimal Page Replacement:** In this algorithm, we assumed perfect knowledge of the future, and evicted the page that would be used in the furthest in the future. In a real-world setting, however this algorithm is not achievable in many systems. However, for the purposes of this project, it will be used as the benchmark to gauge performance across our algorithms.

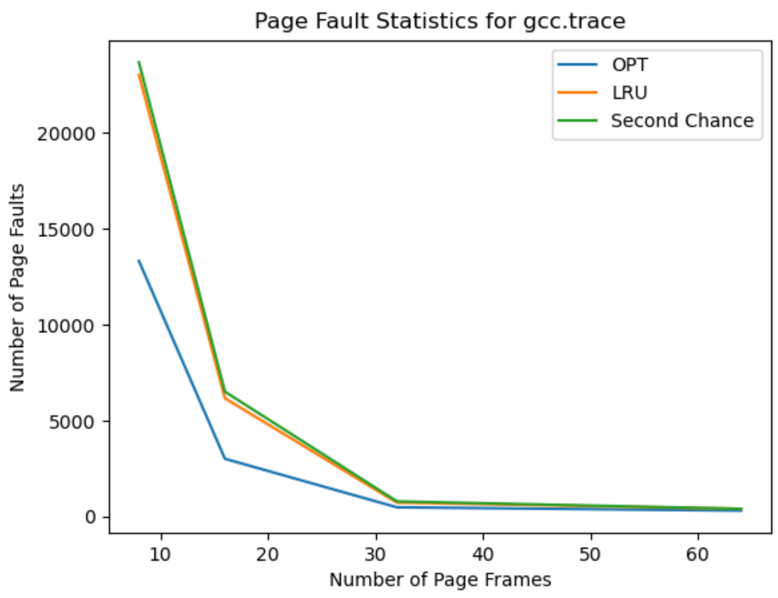
2) **Least-Recently Used (LRU):** In this algorithm, we assume that pages used recently will be used again soon and enforce a LIFO-esque policy to evict pages. The drawbacks of this algorithm lie in the notion of aging. Though we can enforce a LIFO-esque policy using a global counter, it does not enforce the idea of time, and clock cycles.

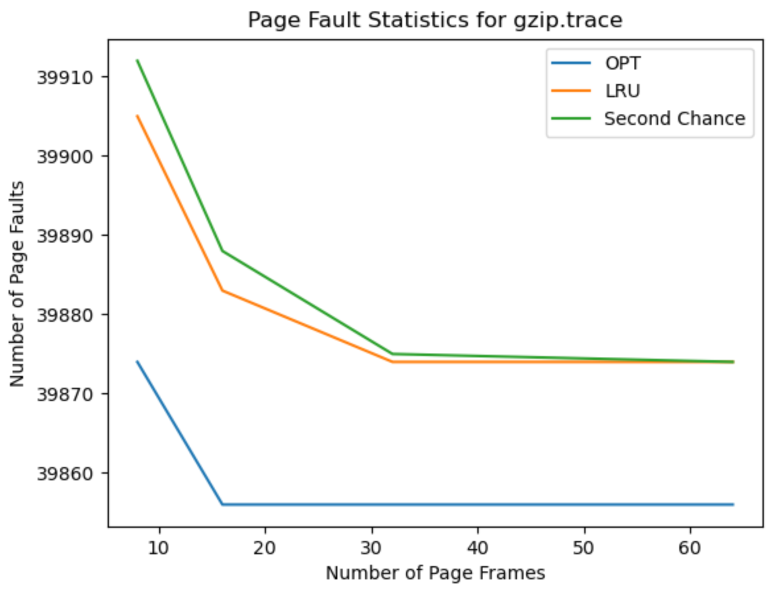
3) **Second-Chance:** In this algorithm, we simulate an algorithm akin to Round Robin. In other words, we enforce a FIFO-esque policy. In evicting a page, we consider each page's reference bit. Depending on the reference, we move the page to the tail of the list. This algorithm can also be simulated using the **Clock Algorithm**. In the Clock Algorithm, we maintain a clock pointer, that will point to the next page to replace, and will evict pages in a similar way to Second Chance, in the sense that we use a page's reference bit as the basis to evict a page. We continually cycle around our clock until we find an appropriate page to evict.

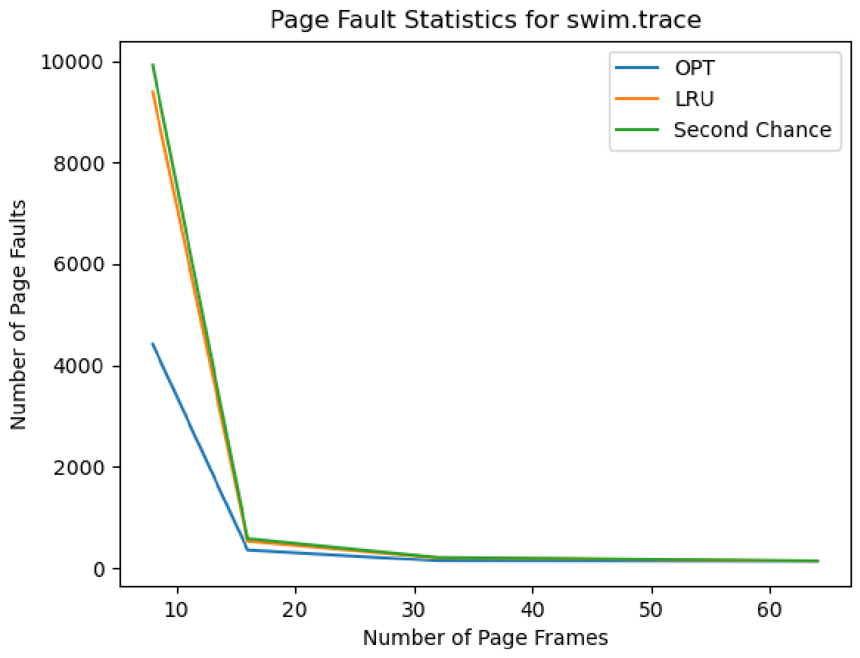
Part I:

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.

In this part, I will describe the page fault statistics for each of the page replacement algorithms for 8, 16, 32, and 64 frames for the given trace files. From the below figures, it is evident that the optimal page replacement algorithm performs best. This should come as no surprise, as in simulating the optimal page replacement algorithm, we assume perfect knowledge of the future, and evict the page that will be used furthest from the future. Therefore, with the optimal page replacement algorithm, the "best" frame will always be chosen to evict. Both second chance and LRU algorithms seem to fail to approximate the results of the optimal page replacement algorithm in gzip.trace. In the LRU algorithm, the evicted page is determined by means of memory references. The drawback of LRU is that there is no enforcement of age. Even though a page may have a high counter value, this does not imply that the page will be heavily used later. With a small number of frames, we can see based on the graphs that LRU fails to approximate the optimal algorithm well. This can be attributed to what was stated above. A smaller number of frames could be problematic. Pages that are referenced heavily could potentially not be evicted, while frames that have the highest counter value could be evicted instead. We should not be surprised that LRU's performance converges to the optimal algorithm as the number of frames increases. As the number of frames increases, we will evict far less pages, and therefore, we are less susceptible to evicting the "wrong" page. The contradiction to this arises with gzip.trace, from the below figure, we can see that both LRU and Second Chance both fail to approximate the optimal page replacement algorithm. LRU failing in gzip.trace implies that there are likely a lot of repeated memory accesses, and the method to simulate LRU without the notion of aging results in LRU evicting a page based on a counter, rather than being based on a counter. Second Chance has the potential fail as it has a high dependency on a page's referenced bit. If the entire page table is consistently full of referenced pages, second chance will consistently scan through the entire page table, and potentially find the "wrong" page to evict. Additionally, with the dependency on the referenced bit, and the instruction type, we have the potential to evict the "wrong" page. Approximation-wise, both second chance and LRU are produce results that are relatively close to each other. We should not be surprised, in simulating these algorithms, we are taken a more Monte Carlo-esque approach. Pattern-wise, both LRU and Second Chance, with the exception of gzip.trace seem to initially have a horrible approximation to the optimal page replacement algorithm, however as the number of frames increases, the approximation improves, and seems to almost converge to the optimal page algorithm. This likely implies that both swim.trace and gcc.trace involve memory addresses that are new, and therefore, bear much less risk than with similar memory addresses. Overall, it seems that LRU typically performs better than Second Chance, and thus, approximates the optimal page replacement algorithm better than second chance. Therefore, LRU, though difficult to implement in real systems is the most appropriate out of the given page replacement algorithms for use in an actual operating system.



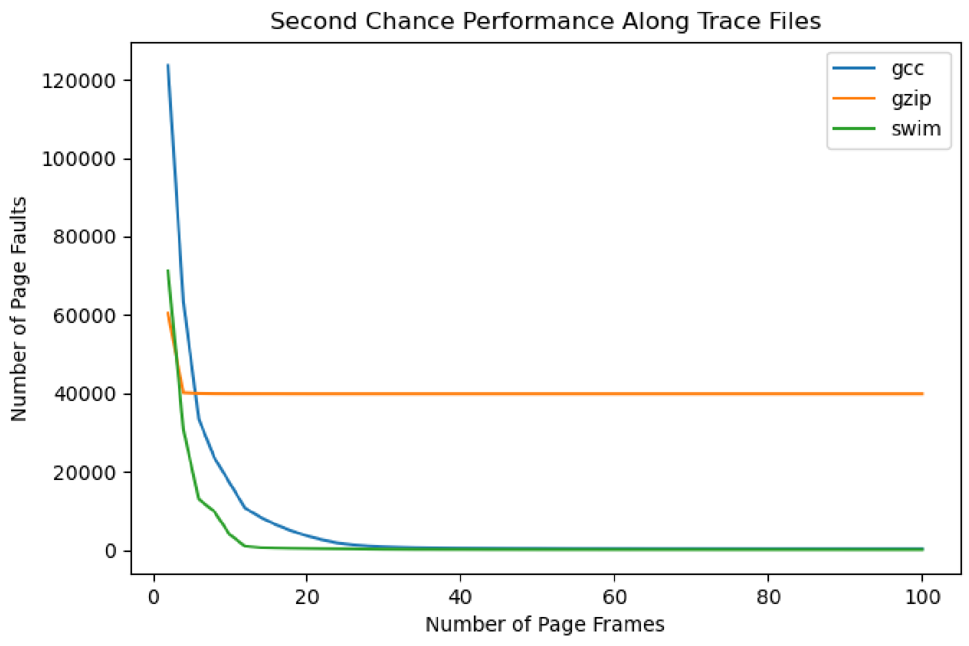




Part II:

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.

In this part, I will describe whether there are any instances of Belady's anomaly in simulating the Second Chance algorithm from 2 to 100 frames. Based on the figure below, there does not appear to be any instances of Belady's anomaly. As we can see, the number of frames does not degrade our algorithm's performance. Particularly, it seems to decrease, as expected. I would however note that Second Chance is still a sub-optimal algorithm, as it operates under a FIFO, Monto-Carlo-esque algorithm. Although second chance performs better than FIFO on its own, the performance doesn't yield much more performance benefits as the number of frames increase. It has more of a reliance on each page's reference bit, this reference bit doesn't necessarily indicate a candidate page to evict. Generally, in a real OS setting, we would want a page replacement, similar to LRU, but that also enforces aging. Thus, ideally, Aging or WSClock would be more optimal in a real system.



Page III: Discuss the implementation and runtime of the OPT algorithm.

I have decided to implement the Optimal Page Replacement algorithm using two TreeMaps, one to represent the future page table, which maps the keys as the addresses read from the trace file upon the first scan through the file, to a Linked List of Integers to represent the lines in the trace file that the address is found on. The second TreeMap is used to represent the actual page table, where I map the keys as the address, and the values associated with entry as Page objects. On the first scan through the tracefile, I put the address, and I add the current index (counter) to the linked list as the value. This way, it will be less costly to search for the future page when a page needs to be evicted. After I finish reading from the file, I scan through the file again, but this time to simulate the optimal page replacement algorithm. I utilize an integer array to keep track of the frames that are in physical memory. I determine whether or not pages can be added to the page table without eviction by comparing the valid bit of the current page that I'm looking to add into the page table, and then comparing a current count of how many frames have been evicted to the number of frames to simulate the optimal page replacement requested by the user. I add to the integer array the address read in, and update the valid bit, as the valid bit would imply that the current page table entry that we want to add is not in the page table yet. If the current page that I want to add to the page table is not in the page table, and I determine when I do need to evict a page. I do so by peeking at the number of frames that have not been evicted, and seeing if it reaches the number of frames that the user has requested. Once I have decided that eviction is required, I determine which page to evict by traversing the future page table, and looking for a page within the page table with the same page number as the current page number. Once I've identified which linked list that corresponds to, I simply peek, and decide on the first linked list element as the page to evict. Once I have identified the page that I want to evict, I swap out the page that I want to evict with the current page that I want to put in. Once that's all said and done, I continue this process until there are no more lines in the trace file.

Performance wise, we will not expect quadratic runtime. Rather, in the worst case, we can expect to search every key inside of the TreeMaps, and since the individual TreeMaps do not have a dependency on each other, we should expect $\theta(\log(n))$ runtime for fetch, and insert methods. TreeMaps are backed by a red-black tree, meaning that any operation is guaranteed to take logarithmic runtime. However, in addition, in the worst case, for the future page table, in evicting a page, within traversing a TreeMap, the possibility exists that we have to traverse an extremely dense linked list within the TreeMap, resulting in runtime akin linearithmic runtime with the additional cost of searching through the pageTable itself, which should be an independent term. The logarithmic portion of the runtime comes from fetching and inserting into the TreeMap, while the linear portion of the runtime comes from inserting and getting elements from the TreeMap's Linked List, while the independent term comes from searching the pageTable, which should not depend on the denseness of the future page table.

To summarize, the runtime of my implementation of the Optimal Page Replacement Algorithm should be: $\boldsymbol{\theta\big(nlog(n)\big) + \ \theta\big(log(n)\big) \sim \theta\big(nlog(n)\big)}$ which, using tilde approximations, should approximate to linearithmic runtime.