CSCI 5103 Programming Assignment 3

- Team member: Tianhong Zhang (zhan4868) and Mengzhen Li (li001618)

Experiments purposes and set up

1. Experiments purpose

   The goal of this experiment is to compare three page replacement algos with regard to the volume of disk writes and page faults that occur when running programs with various patterns of memory access.

   By contrasting the outcomes of the three page replacement algorithms applied to different memory access patterns, we can learn which one is more appropriate for a given use case.

2. Experimental setup

    - We ran our experiment on a Ubuntu machine with pandas 1.5.2 installed.
    - Check pandas version:

      ```
      > pip show pandas
      ```

    - If need to upgrade panda, please run the following command:

      ```
      > pip install --upgrade pandas
      ```

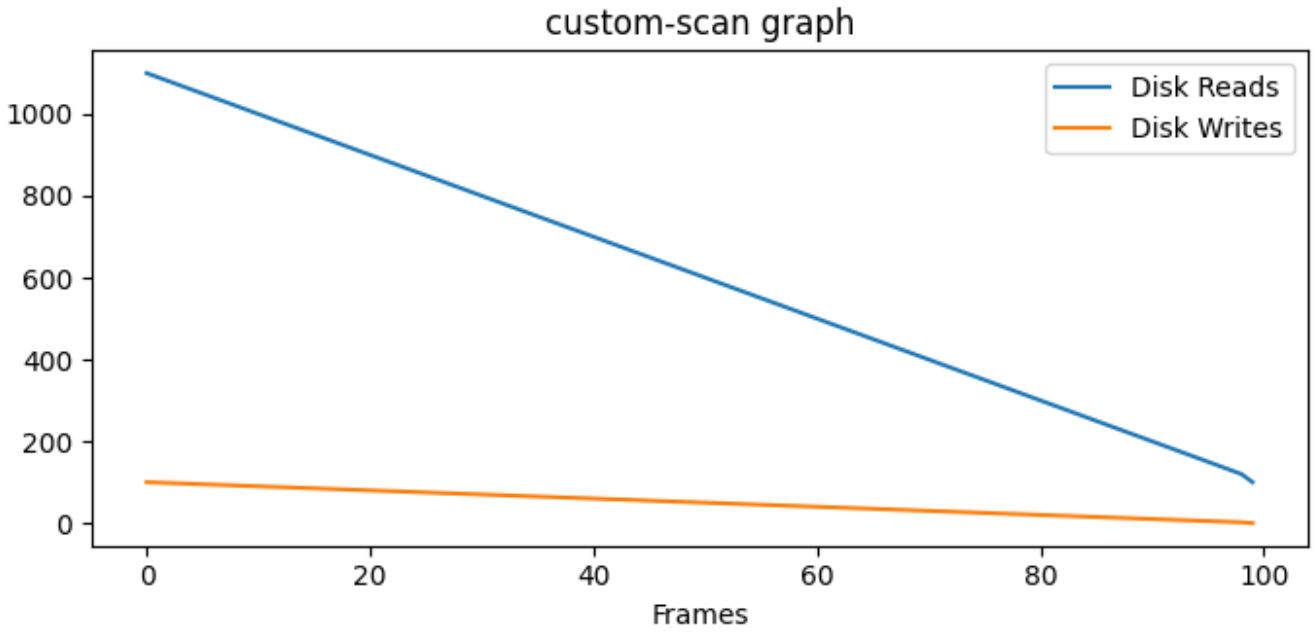
3. How to run the program?

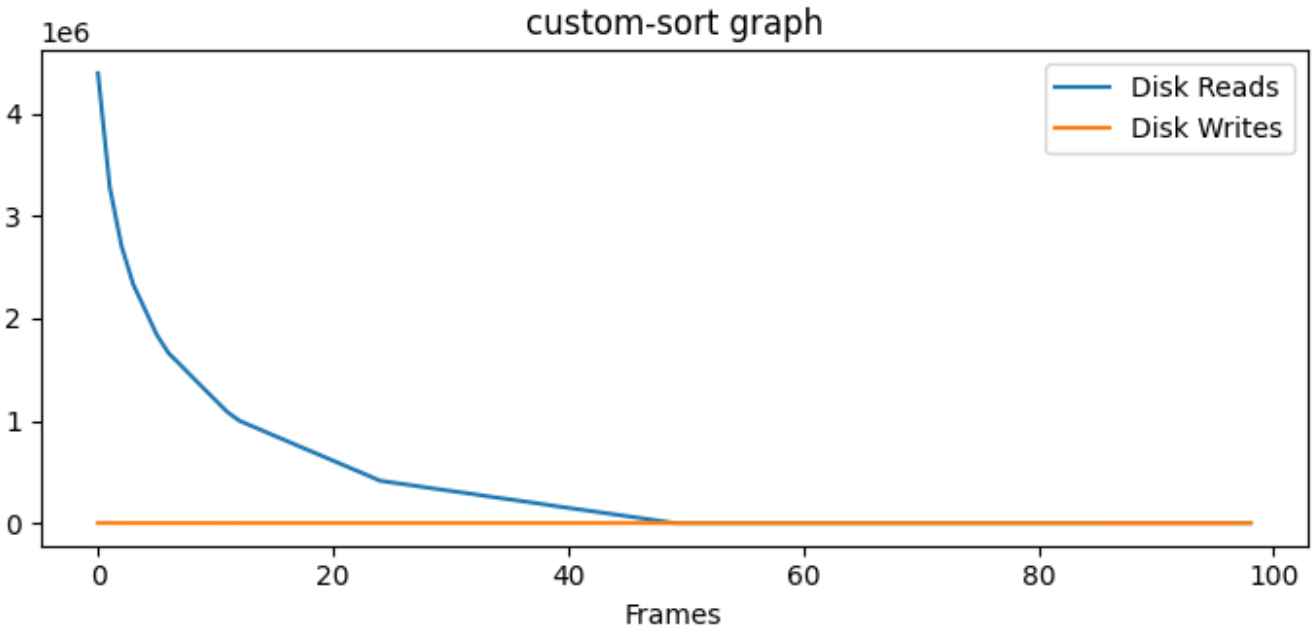
   To generate the graphs we presented in this report, please run the following commands in order:

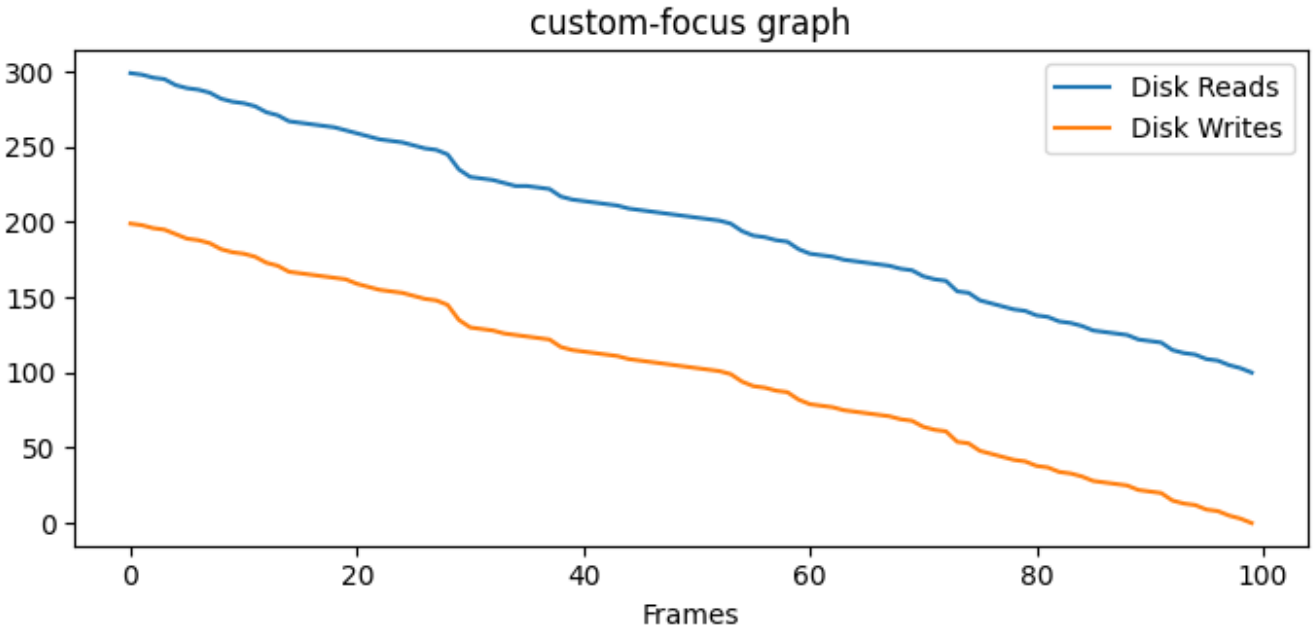
   ```
   > ./evaluation.sh
   > python3 generate_graphs.py
   ```

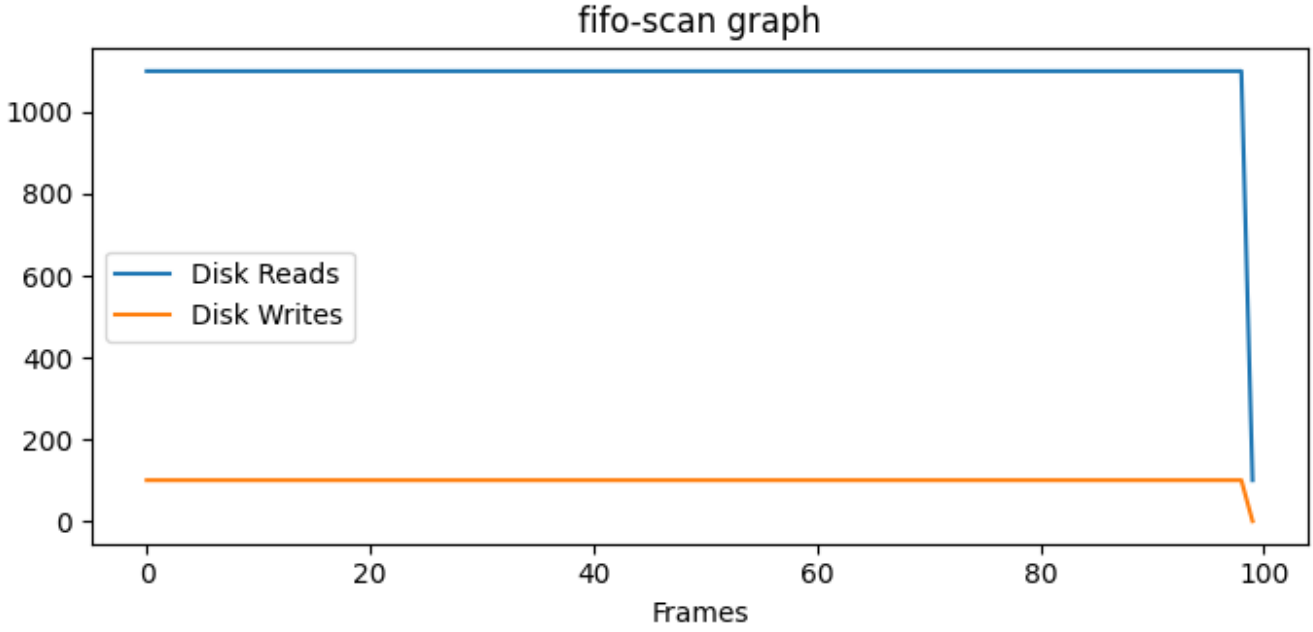
The `evaluation.sh` script runs each program and each page replacement algorithm using 100 pages and varying numbers of frames between 1 and 100, and stores the result into a csv file in the csv-files folder. The `generate_graphs.py` will generate 9 graphs for every algorithm and program combinations, and 1 big graph which combines the comparison graphs of each algorithm. All graphs generated by `generate_graphs.py` are stored in the project ./graphs directory.

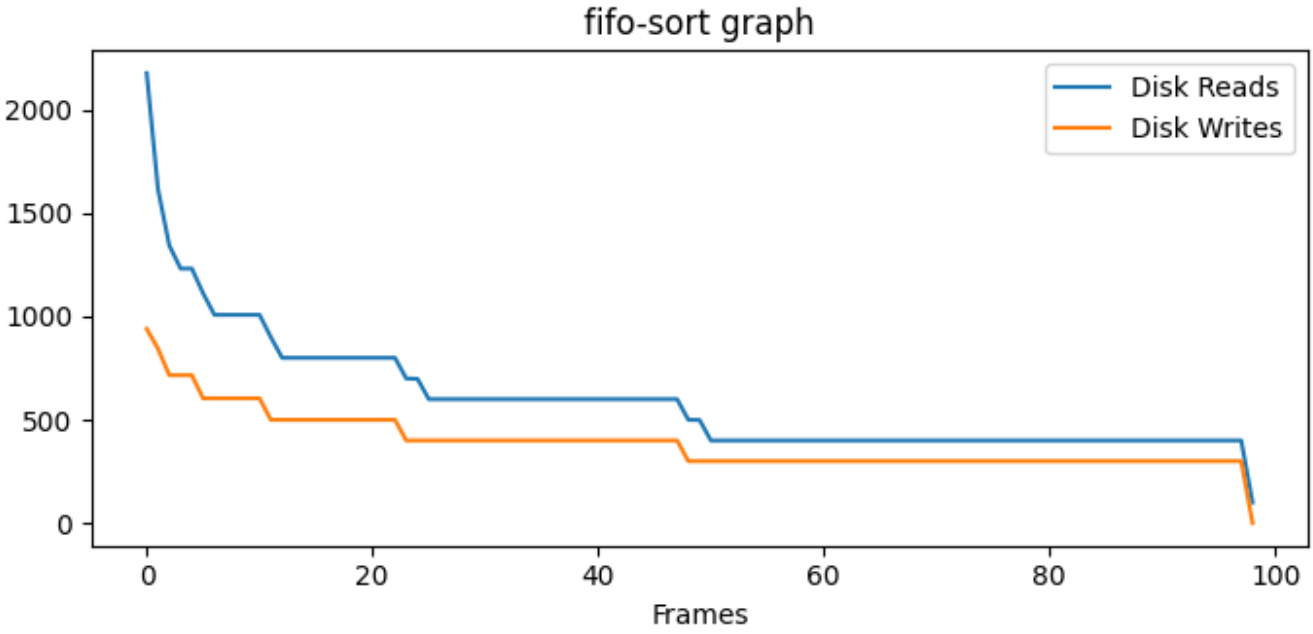
Graphs of the number of *page faults* (*disk reads*), and *disk writes* for each program and each page replacement algorithm

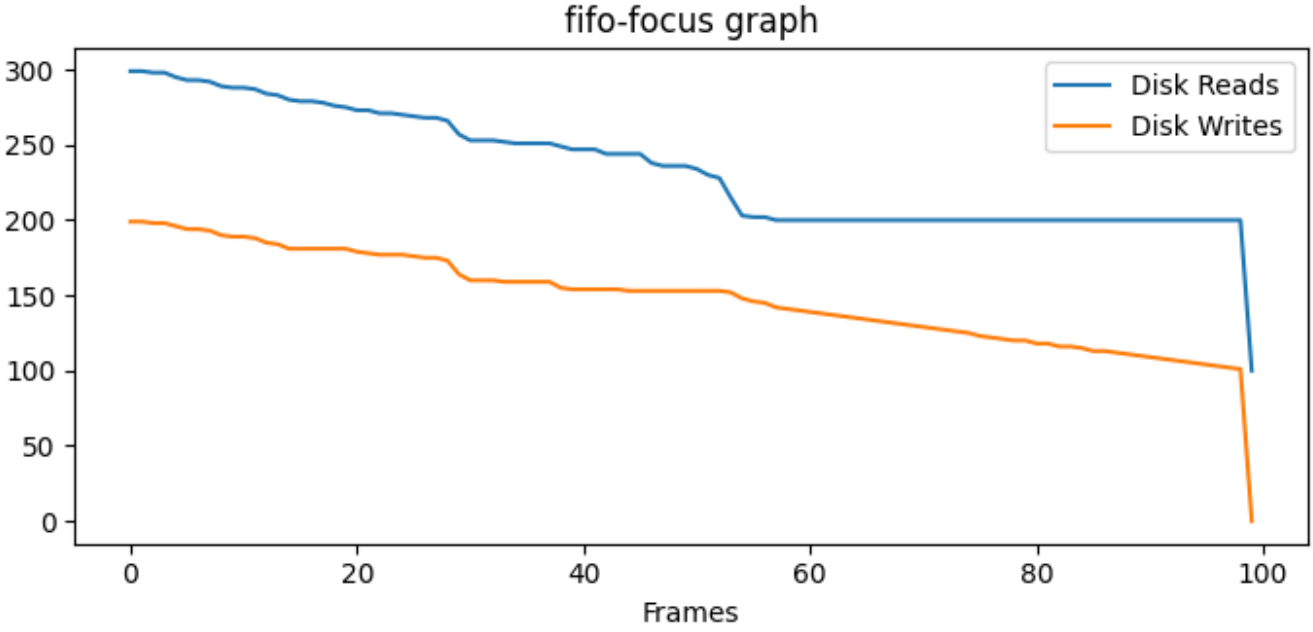


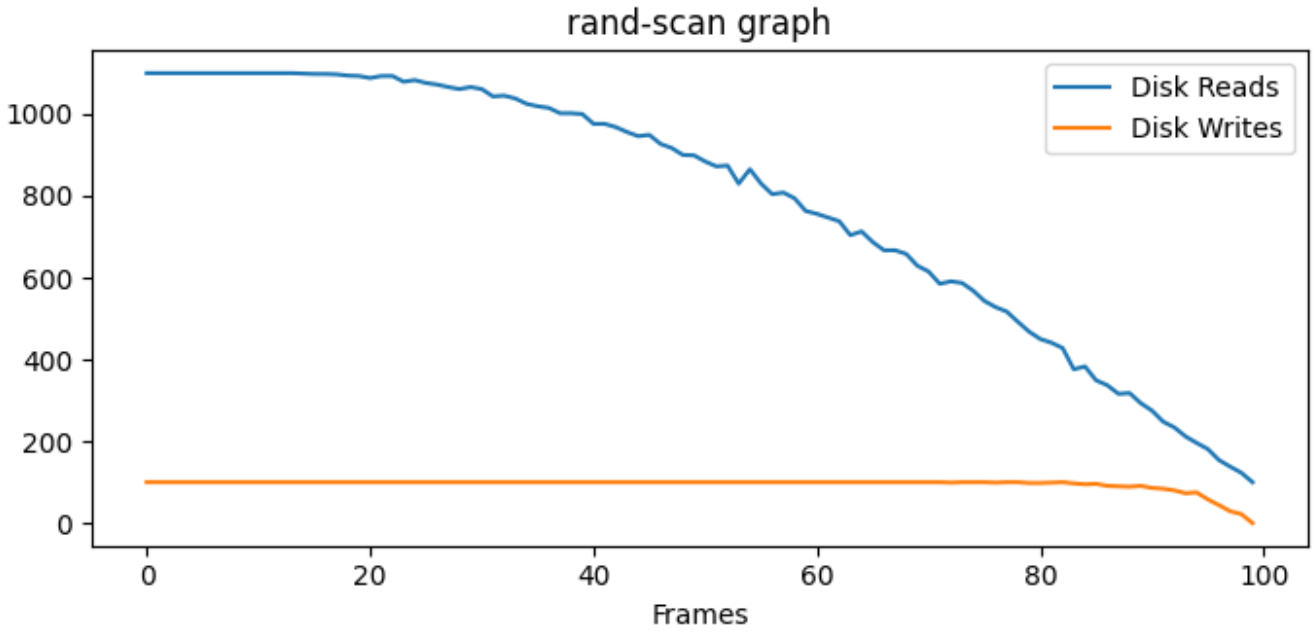


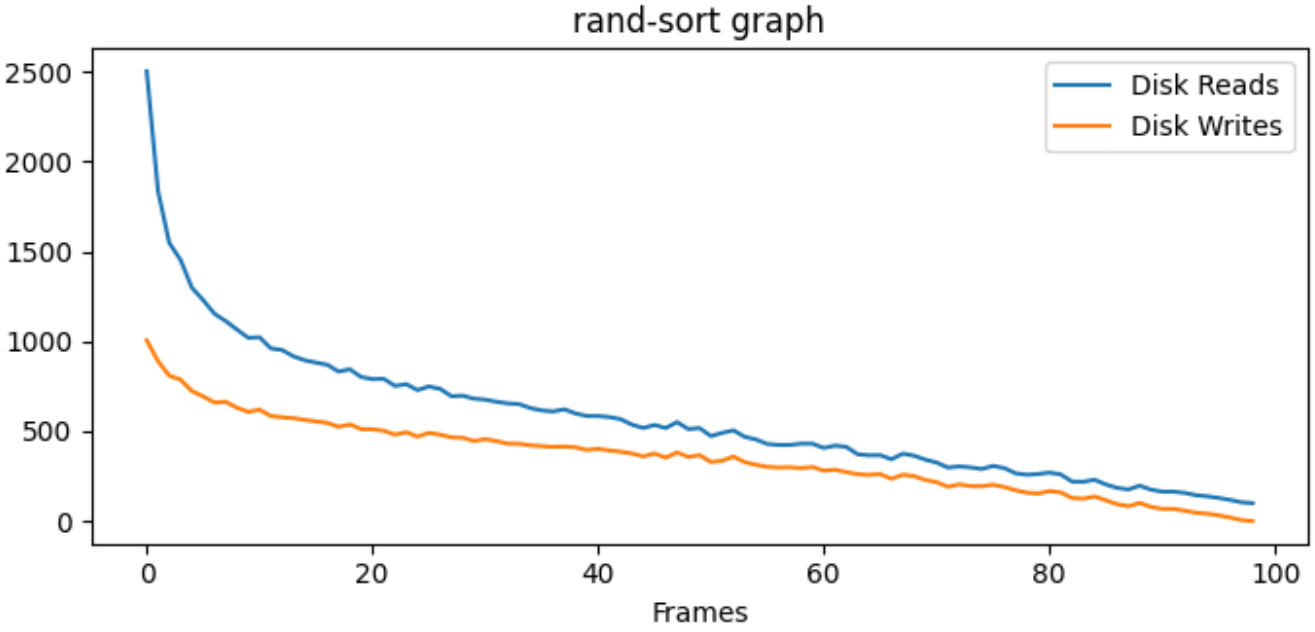

custom-focus graph
Disk Reads
Disk Writes
300
250
200
150
100
50
0
0
20
40
60
80
100
Frames

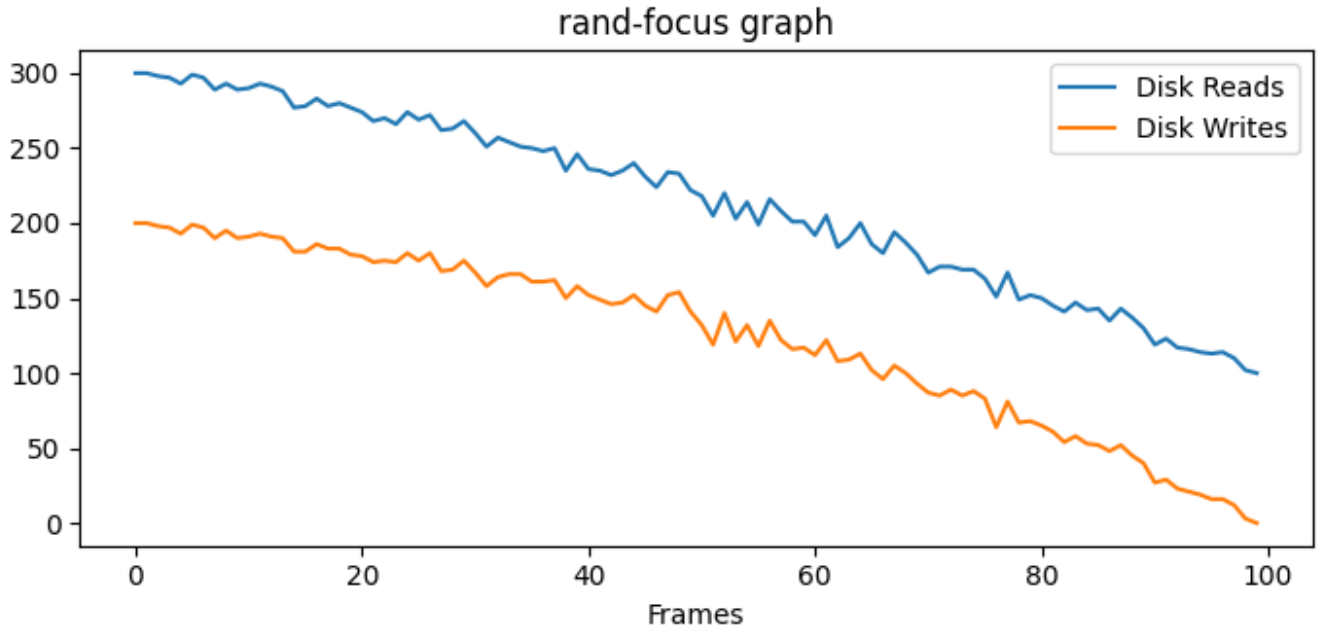

fifo-scan graph
Disk Reads
Disk Writes
1000
800
600
400
200
0
0
20
40
60
80
100
Frames

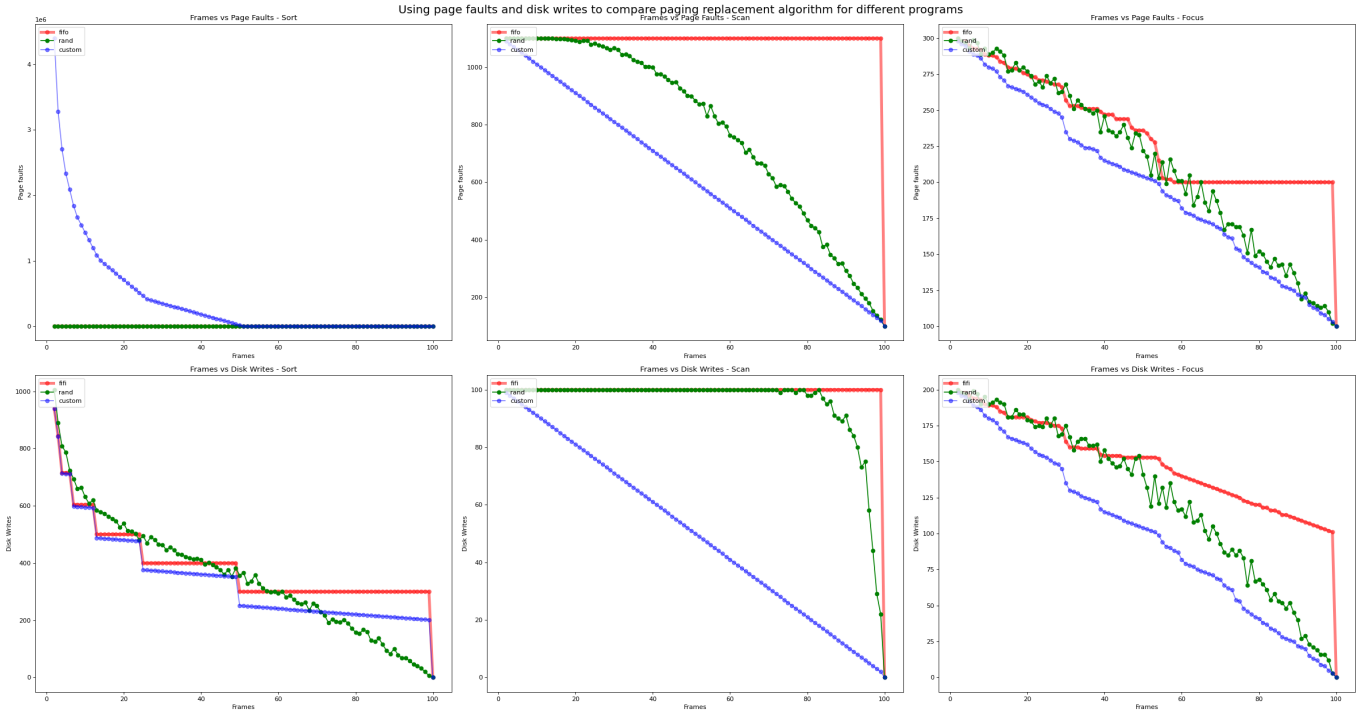

fifo-sort graph
Disk Reads
Disk Writes
2000
1500
1000
500
0
0
20
40
60
80
100
Frames


fifo-focus graph
Disk Reads
Disk Writes
300
250
200
150
100
50
0
0
20
40
60
80
100
Frames


rand-scan graph
Disk Reads
Disk Writes
1000
800
600
400
200
0
0
20
40
60
80
100
Frames


rand-sort graph
Disk Reads
Disk Writes
2500
2000
1500
1000
500
0
0
20
40
60
80
100
Frames





Analyze results and describe when one algorithm outperforms the others, and why.

`sort`

1. what does `sort_program` do?

   `sort_program` generates random value, put it into an array of `length=PAGE_SIZE*npage` and sorts it using `qsort()`.

2. Different page_fault_handlders performance?

When the cache is small (npages=100, nframes < 50), custom handler performs drastically worse than rand and fifo algorithms. It is so bad that, the graph for rand and fifo seems flat.

When the cache is big (npages=100, nframes >= 50), custom handler has the best performance, which means that the number of page faults and disk writes is smaller than the other two algos.

3. Why does one algorithm outperforms the others?

   The data in our `virtmem` is changing frequently, the locality does not hold for sort program.

   The principle of **locality** is an observation about programs and their behavior. What this principle says is that programs tend to access certain code sequences (e.g., in a loop) and data structures (e.g., an array accessed by the loop) quite frequently; we should thus try to use history to figure out which pages are important, and keep those pages in memory when it comes to eviction time.

   This **locality** does not hold for sort_program, especially when the cahe is small.

`scan`

1. What does `scan_program` do? It takes a pointer to a block of memory and the length of that memory. It then fills the block of memory with a pattern. It then scans the block of memory and sums the values. It then verifies the sum.

2. Different page_fault_handlders performance?

   custom's performance is the best, rand second, fifo worst.

3. Why does one algorithm outperforms the others?

   custom evict the least frequently used page in memory and the principle of locality property holds perfectly for scan program.

`focus`

1. what does `focus_program` do?

   The purpose of the focus_program is to test the cache It does this by writing to the array and then reading from the array It does this 100 times It then sums the array and compares the sum to the sum of the array It then prints out the sum and the expected sum

2. Different page_fault_handlders performance?

   custom's performance is the best, rand second, fifo worst.

3. Why does one algorithm outperforms the others?

   focus program has many same properties with the scan program.

Summary of three page replacement algorithms:

When to use Custom(LFU) cache?

- When nframes is small, and the data is not changing frequently.

- When nframes is large, and the data is changing frequently.
- When nframes is large, and the data is not changing frequently.

When to use FIFO cache?

- When nframes is small, and the data is changing frequently.
- When nframes is large, and the data is changing frequently.
- When nframes is large, and the data is not changing frequently.

When to use randon page replacement algorithms?

- When nframes is small, and the data is changing frequently.
- When nframes is large, and the data is changing frequently.
- When nframes is large, and the data is not changing frequently.

Customized page replacement algorithm

1. what additional implementations we have added? How the customized algorithm determines which page to replace?

   Our `page_fault_handler_custom`, handles a page fault using the custom policy.

   It can occur if the write permissions were not set or if the page doesnt exist in memory.

   The page replacement policy for our custom hander is is very similar to LFU cache replacement policy.

   To achieve this, we implement a new class called `CustomCache`.

   In `CustomCache`, we define two hash tables:

   ```
   unordered_map<int, list<Node>::iterator> key_table;
   unordered_map<int, list<Node>> freq_table;
   ```

   The first `freq_table` is indexed by frequency--`freq`. Each index stores a doubly linked list. This linked list stores all caches with frequency `freq`. There are three pieces of information stored in the cache, namely the `key` which is the page number, `val` which is the frame number, and the frequency--`freq`.

   The second `key_table` is indexed by the key value `key`, and each index stores the corresponding memory address cached in the linked list in `freq_table`, so that we can use two hash tables to make the time complexity of both `put()` and `get()` operations **O(1)O(1)**.

   At the same time, it is necessary to record a frequency `minFreq` that is the least used by the current cache, which is for evicting the leased used page operation.

   - For the `get(key)` operation, we can find the memory address of the linked list cached in `freq_table` in `key_table` through the index `key`. If it does not exist, return -1 directly,

otherwise we can get the relevant information of the corresponding cache, so that we can know The `key` value of the cache and the frequency of use can be directly returned to the value corresponding to the key.

But we noticed that the usage frequency of this cache increases by one after the `get()` operation, so we need to update the location of the cache in the hash table `freq_table`. Knowing the key `key`, value `val`, and usage frequency `freq` of this cache, then the cache should be stored in the linked list under the `freq + 1` index in `freq_table`. So we delete the node corresponding to the cache in **O(1)O(1)** in the current linked list, update the `minFreq` value according to the situation, and then insert it into the head of the linked list under the `freq + 1` index to complete the update.

We inserted into the head of the linked list when updating. This is actually to ensure that the insertion time cached in the current linked list from the head of the linked list to the end of the linked list is in order, serving the following evict operations.

- For the `put(key, value)` operation, we first check whether there is a corresponding cache in the `key_table` through the index `key`. If so, the operation is actually equivalent to the `get(key)` operation. The only difference is that we need to put the current cache The `val` in is updated to `value`. If not, it is equivalent to a newly added cache. If the cache has reached capacity, you need to delete the least recently used cache before inserting it.

  Consider insertion first. Since it is newly inserted, the frequency of use of the cache must be 1, so we insert the cached information into the head of the list under index `1` in `freq_table`, and at the same time, update the information of `key_table[key]` and `minFreq = 1`.

- For the `evict()` operation. Since we maintain `minFreq` in real time, we can know the index with the least frequency of use in `freq_table`. At the same time, because we ensure that the insertion time in the linked list from the head of the linked list to the end of the linked list is in order, the node at the end of the linked list of `freq_table[minFreq]` is the **node with the least frequency of use** and **the earliest insertion time**. We return a pair of ints `<key, val>`.

2. Difference between LFU cache and our custom cache.

   CustomCache cannot update the cache table upon each memory reference. For example, if a particular page's write permissions were set, memory reference cannot be detected by the `page_fault_handler_custom`.