# Page Fault Handling and Management

**Operating Systems CSCI5103**

By

Loose Threads:

Subhash Sethumurugan - sethu018@umn.edu
Bhaargav Sriraman - srira048@umn.edu
Edwin Nellickal - nelli053@umn.edu

# Table of Contents

# 1. Introduction

Paging is a key factor that affects the performance of an application. How efficiently an application gets the data it wants to work on can affect its runtime. With ever-changing requirements of applications and the underlying hardware, custom paging is required to get better throughput out of applications. However, we acknowledge the importance of fundamental paging schemes, which are still handy in terms of understanding more complicated paging schemes.

As outlined, for this project, we have implemented paging schemes which evict pages in random or in FIFO fashion using the APIs provided for page creation, page mapping, disk read, disk write, etc. We have also implemented a custom paging policy, which is a simple approximation to LFU and low on computation to achieve the same.

The aforementioned schemes have been written to emphasise readability while incorporating metrics to collect information such as the numbers of page-faults, page writes, page evictions, disk reads, disk writes, etc. A few simple test cases have also been introduced to check if the implemented policies are working as expected by comparing the output of two sort algorithms, of which one does the sorting without having to compare and swap elements

# 2. Experiment Setup

The experiments are carried out in order to study the characteristics of different eviction policies. The graphs compare the performance of these policies which gives us a better explanation of how each policy behaves, what it does best and how it can be used efficiently.

**System used:** The data points were collected while running the code in cse lab systems.

Command used:

```
./virtmem 100 <frames> <policy> <program>
```

Where **`frames`** is the number of frames that are to be allocated while running virtmem. It lies between 1 and 100.

**`policy`** is the page eviction scheme that is to be used while the program is executed. Which can be either rand, fifo or custom.

Three policies are implemented in the experiments.

1. **`rand`**: Frame to be allocated to the new page is randomly chosen.
2. **`fifo`**: The page which has stayed the longest in memory is replaced by the new page.
3. **`custom`**: An approximated LFU is implemented as a frame allocation strategy.

**`program`** is the piece of code that operates on virtual memory to solve a simple computation. The program can either be **`sort`**, **`focus`** or **`scan`**.

**`test`** is another program that provides additional test cases that we have developed for testing our paging implementation. The test cases include:

1. **`mean_mode`**: the program randomly assigns a value between 0 and 127 to all elements in the data. While doing so the test program keeps track of the frequency of each number that is being stored in virtual memory and updates the total count of all numbers that are being generated. Once all the data is allocated, it displays the mean and mode of the data.

   Run command:

   ```
   ./virtmem 100 <frames> <policy> mean_mode
   ```

2. **`count_sort`:** similar to the computation above, the program keeps track of the frequency of all the elements that are being allocated. These frequencies will be used to sort the data in place without having to compare and swap elements. The same data is sorted using quicksort. The output for count sort and quick sort is compared element by element to ensure that data is not lost or corrupted when compare and swap is performed for sorting.

   Run Command:

   **`./virtmem 100 <frames> <policy> count_sort`**

# 3. Custom Page Replacement Algorithm

We implemented an approximation to LFU (Least Frequently Used) algorithm as our custom replacement policy. In LFU, a page accessed the least number of times within a time window is chosen for eviction when a page needs to be replaced in memory. Access could be either a page read or page write. To realize this algorithm, we need to maintain a count for each page and increment the count whenever the page has been accessed or modified.

## 3.1 Challenges in implementing traditional LFU

1. The handler needs to know a page is being accessed by the program every time to maintain an accurate count.
2. This implies that the pages in the frame must always have no access rights. As a result, an exception is raised every time the program tries to use a page.
3. The dirty bit is necessary as it decides which pages are written to the disk before evicting. In this implementation, it becomes complex to keep track of the dirty bit.
4. We then decided to implement an **approximation to LFU** rather than trying to implement something as complex as stated above.

## 3.2 Salient features

1. The custom algorithm maintains a data structure to count the number of times page faults occur for each page. The handler refers to this data structure in the case of evictions.
2. The handler is called whenever the program is trying to access a page that is not in memory or when it tries to write to a read-only page. When the handler is being called, it updates the count which is later used in the eviction policy.
3. At the time of eviction, the handler iterates through all the pages in memory to find the page which was accessed the least number of times. It then decides to evict this page to place the interested page in memory.

## 3.3 Behaviour of the algorithm

1. At the beginning of the program, the algorithm acts fairly unbiased. This is because the pages hardly have any count and they take turns to be evicted, just like FIFO.
2. But as count grows, the algorithm gives more priority to the page accessed frequently in the past.
3. As a consequence, it reduces the number of evictions in total. This also brings down the number of disk reads and writes. The same can be inferred from the plots.

# 4. Evaluation

We have evaluated programs - sort, scan and focus by running each program for 100 virtual pages and varying the physical frames from 1 - 100. From our simulations, we have collected the stats - page faults, disk reads and disk writes. To further understand the effects of the replacement policies, we have generated the below plots. The x-axis varies from 1 to 100 frame and y-axis denotes the collected stats - page faults, disk reads and disk writes respectively.

## 4.1 Plots for sort program

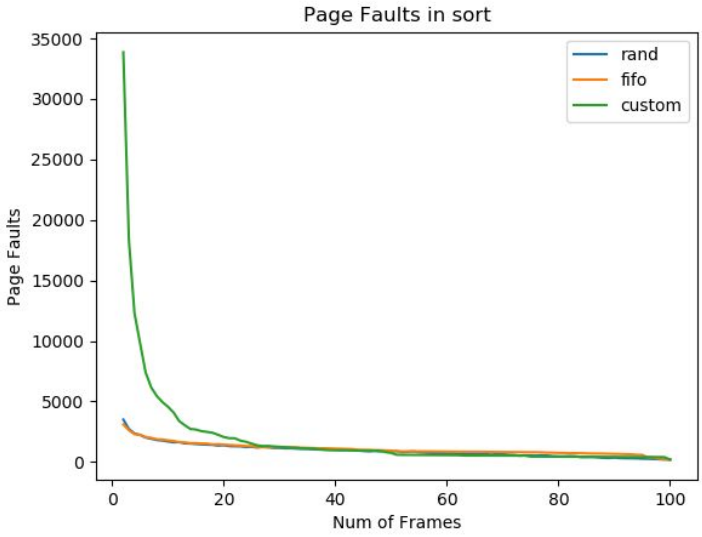


**Fig 1: Page faults for sort (all policies)**

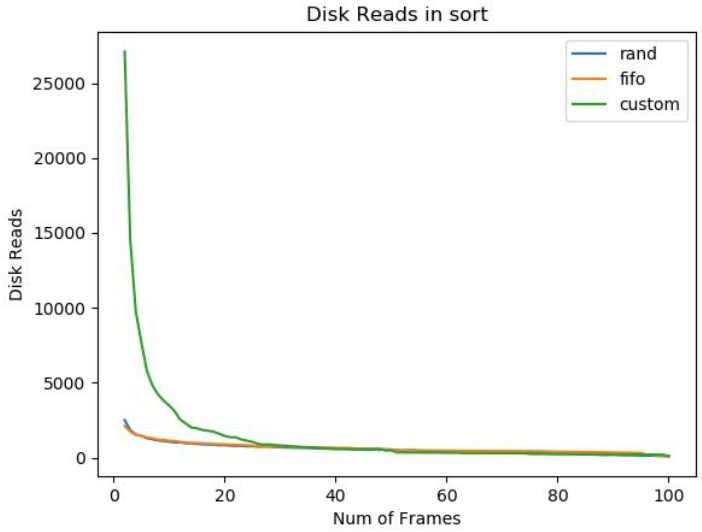


**Fig 2: Disk Reads for sort (all policies)**

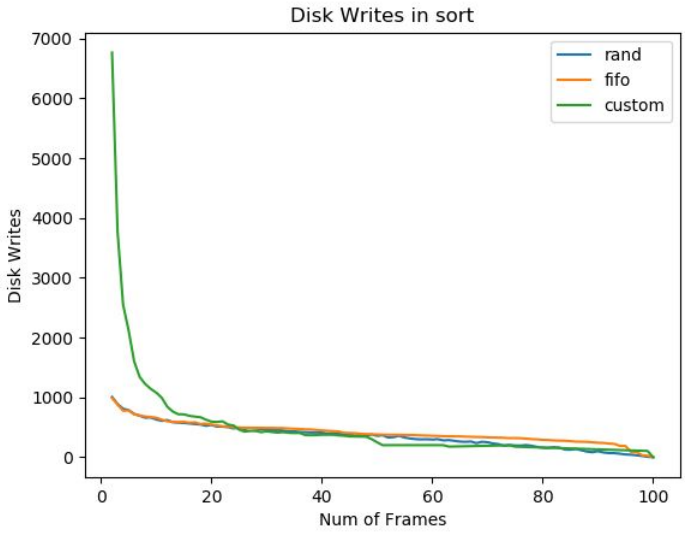


**Fig 3: Disk Writes for sort (all policies)**

### 4.1.1 A note on sort:

When sorting an array, the program compares two elements to figure out which is greater than the other. As a result, it needs two variables to make this decision, although both the variables need not be in the same page. This is why the sort program never terminates when it is executed with just one frame, as it would not be able to access the other variable. This would lead to page thrashing and the program would never proceed.

### 4.1.2 A note on rand paging policy:

The outputs of FIFO and custom always match whereas rand produces a different output in focus program. This happens as rand depends on the random number generated in the handler. The usage of rand in the eviction policy randomizes the number generated in the program. On the other hand, FIFO and custom always generate the same random number when it runs.

## 4.2 Analysis of sort program

### 4.2.1 Behavior of sort with custom policy

In the sort program, custom policy (approximate LFU) has the highest number of page faults when fewer frames are available in memory. To understand this, we need to review the qsort algorithm.

The algorithm does the following:

1. Pick a pivot element in the array
2. Re-arrange other elements by comparing against the pivot.
3. Follow the first two steps recursively in the elements lesser than and elements more than the pivot separately.

From the algorithm, we can realize that, in any iteration, the pivot element is the one that is accessed the most. However, once the pivot is placed, it is never accessed later.

This behaviour goes against the custom policy for the following reasons:

1. The page which contains the pivot element will have the highest access count after one iteration.
2. The algorithm tries to retain this page in memory because of high access count.
3. However, that page is never accessed again because the pivot is placed in its position and need not be referenced again.
4. When fewer frames are available, retaining the page that is never going to be accessed has detrimental effects.
5. As the number of frames increases, the extra frames allows us to retain the frames that are actually referenced in the future. Hence, the number of page faults decreases.

### 4.2.2 Behavior of sort with fifo policy

FIFO works well for qsort because the page accessing pivot element is accessed first. After it is used, it is evicted and never accessed again. The sort algorithm favours FIFO in a way. Hence, FIFO performs well with the sort program.

### 4.2.3 Behavior of sort with rand policy

Although FIFO does a good job in evicting the page accessing pivot element after the first iteration, it suffers a lot of page faults during the first iteration as the page accessing pivot element is used extensively. Rand policy randomly chooses frames for eviction. With fewer frames in memory, the chances of randomly evicting page accessing pivot element during the first iteration are more. Hence, with fewer frames, rand performs poorly as compared to FIFO. However, as the number of frames increases, rand outperforms FIFO as shown in Fig 4.

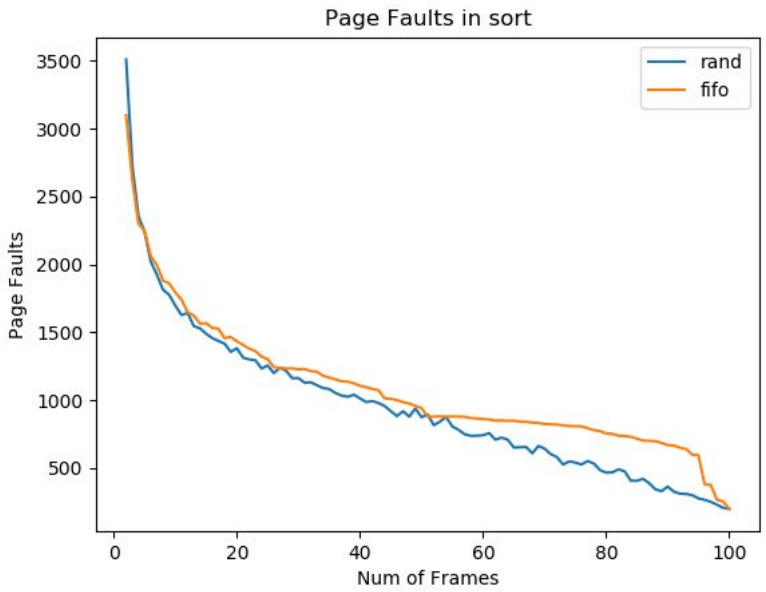


**Fig 4: Page faults for sort (rand, FIFO)**

## 4.3 Plots for scan program

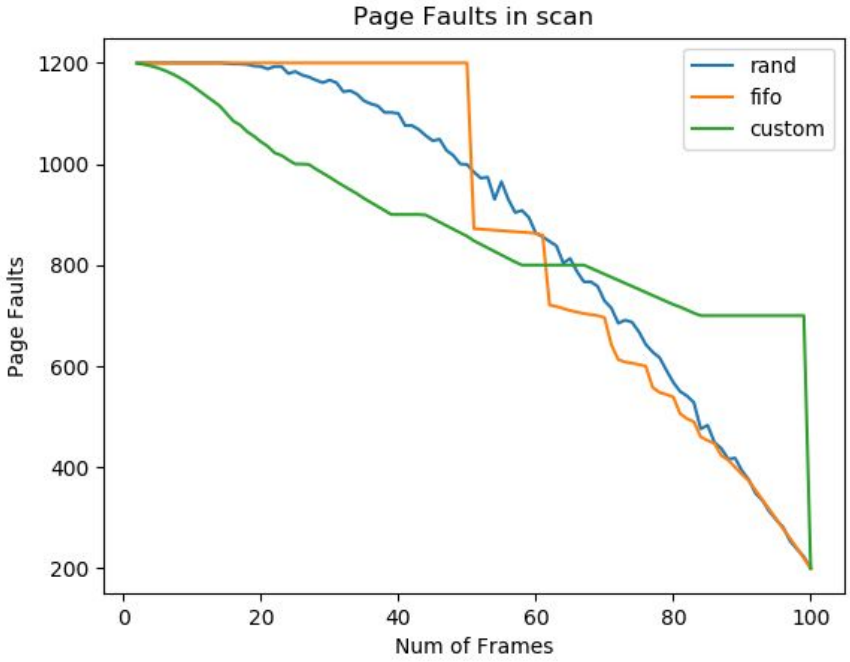


**Fig 5: Page faults for scan (all policies)**

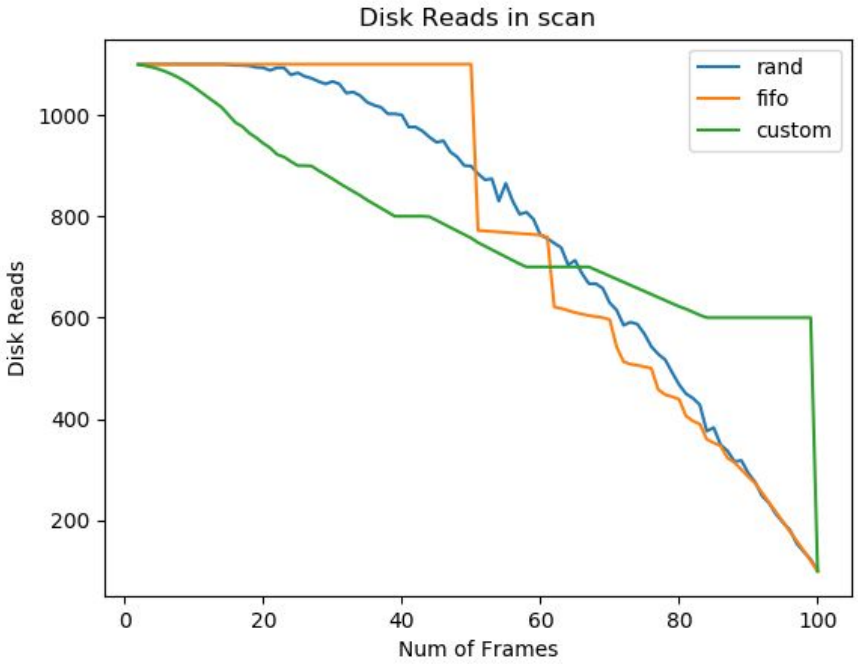


**Fig 6: Disk Reads for scan (all policies)**

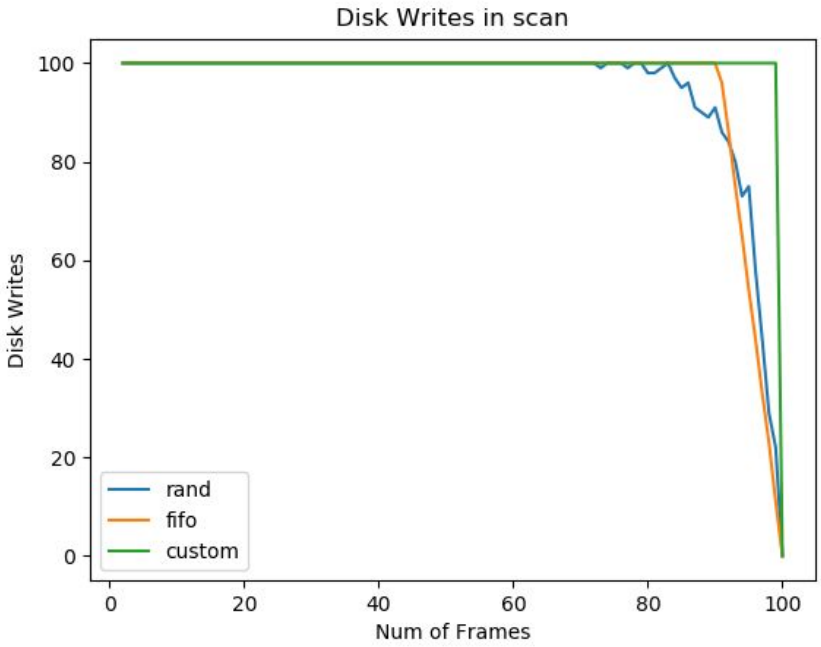


**Fig 7: Disk Writes for scan (all policies)**

## 4.4 Analysis of scan program

Custom policy performs better than rand and FIFO when fewer frames are available. As the number of frames increases, rand and FIFO outperform the custom policy. The reason being, in scan algorithm the number of accesses to all pages is the same. When fewer frames are available, FIFO tends to evict pages which are about to be accessed. As the number of frames increases, the effect of evicting pages about to be accessed deteriorates.

In custom policy, pages with more number of accesses are retained. When fewer frames are available, newly accessed pages are preferred for eviction. The policy tends to retain a fixed number of pages in memory while evicting newly accessed pages. The old pages keep receiving hits, while new pages are always missed. This is better than FIFO where all pages about to be accessed are evicted. However, when the number of frames increases, the difference in access count between pages become small and all pages are a candidate for eviction. Hence, the performance deteriorates.

## 4.5 Plots for focus program

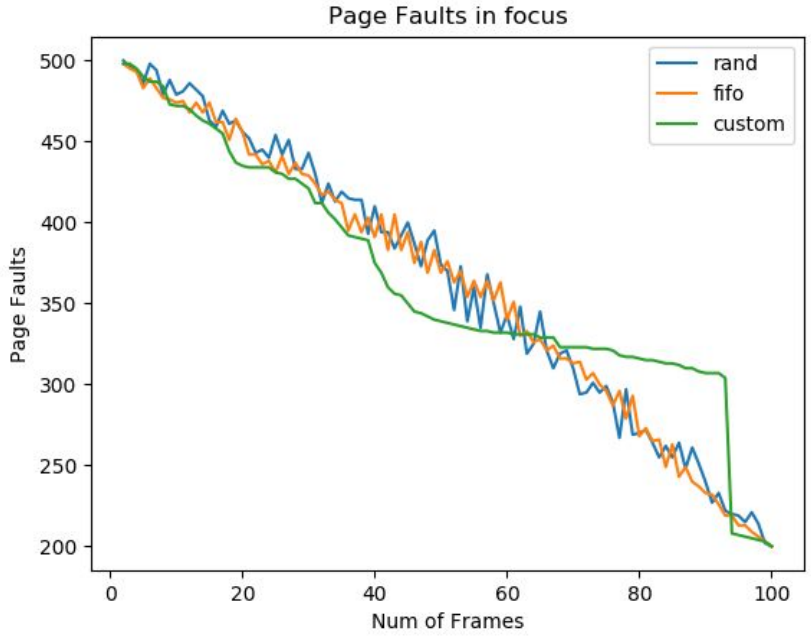


**Fig 8: Page Faults for Focus (all policies)**

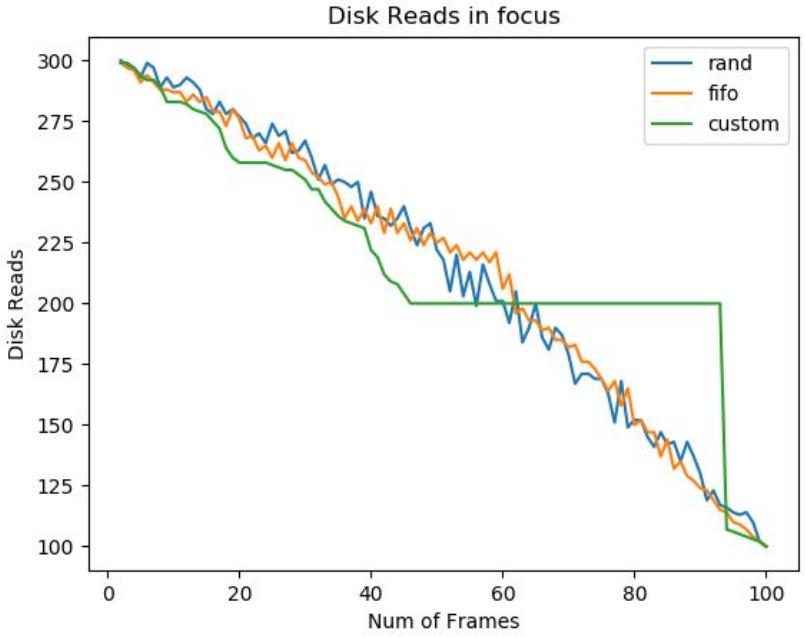


**Fig 9: Disk Reads for Focus (all policies)**

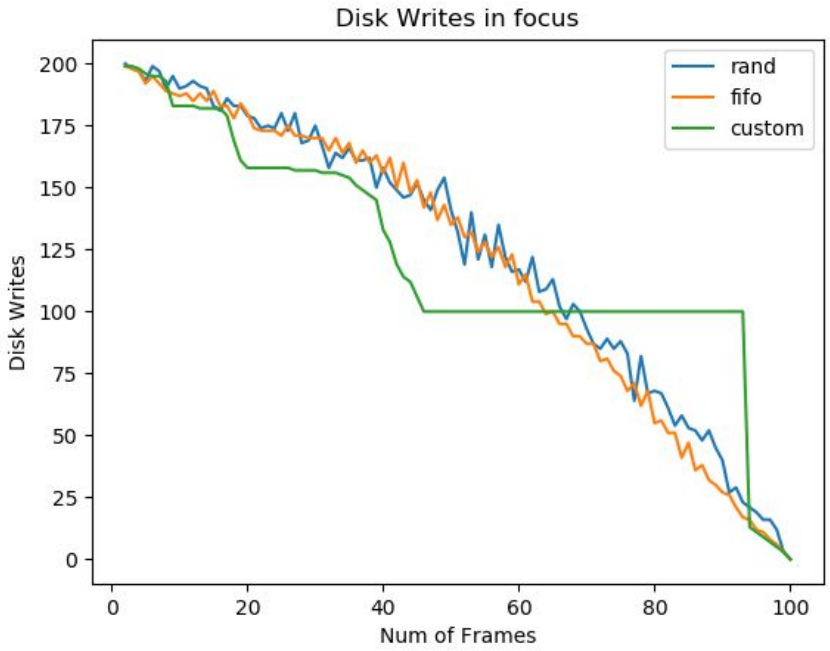


**Fig 10: Disk Writes for Focus (all policies)**

## 4.6 Analysis of focus program

Custom policy behaviour is similar to as explained for scan program.
Rand and FIFO perform very similarly because the pages are accessed are in random order.

## 4.7 Overall Analysis

From the plots we can make the following observations for all replacement policies:

1. Page Fault is inversely proportional to the number of frames.
   Reason: Fewer frames implies fewer pages in memory. Accessing pages that are not in memory causes page faults. By increasing the number of frames, we are accommodating more pages in memory, hence fewer page faults.

2. Disk reads and Disk writes are directly proportional to page faults
   Reason: Each page fault requires the OS to allocate a frame in memory (choosing an available frame or evicting the page in a used frame) and copy the accessed page from disk to memory. Copying page from memory causes disk reads. Evicting a page from memory causes a disk write (if the page is modified). Hence, disk reads and writes are directly proportional to page faults.

## 4.8 Comparison of Policies

**FIFO** performs poorly when pages are accessed in sequential order but the number of pages is more than the number of frames available. FIFO tends to kick out pages that are about to be accessed. FIFO performs well in cases where pages are accessed in a timely order, that is, when some pages are accessed for a time window and then never accessed again while other pages are accessed in a different time window. FIFO tends to retain pages accessed in the current window in memory while evicting old pages.

**Rand** performs better when there is no particular order of access among pages. In other cases, rand policy performance varies for each run. If pages in use are evicted, it performs poorly while it performs better when it retains pages in use.

**Custom**(approximate LFU) performs better in cases when few pages are accessed extensively while other pages are accessed sparsely. The policy would then retain the extensively used pages while evicting pages that were accessed relatively fewer number of times. The Custom policy performs poorly when all pages are accessed equally. As the number of accesses increases, it retains some pages in memory. In the meantime, the access count to other pages increases and will eventually replace the pages already existing in memory. In such scenarios, it is better to retain a portion of the working set in memory rather than switching between pages.

# 5. References

https://linux.die.net/man