Shashank P
200010048

CS314 Operating Systems Lab
Lab 8

1 Algorithms

1.1 FIFO

FIFO is also used as a page replacement algorithm for swap space. When a system runs out of physical memory, it swaps out the oldest inserted page to disk and brings a new page from the disk into physical memory. FIFO is a simple and efficient algorithm where the page that was brought into the swap space first is the one that is selected for replacement. While FIFO is a simple and easy-to-implement algorithm, it does not necessarily choose the best page for replacement as it doesn't take into account how often a page is accessed or how frequently it's likely to be used in the future.
The code snippet for the logic is shown below.

```
for (auto i : requests)
{
    if (memory_frames.find(i) != memory_frames.end())
        continue;

    pageFaults++;
    if (memory_frames.size() >= memory_size)
    {
        int swap = memory.front();
        memory_frames.erase(swap);
        memory.pop();
        swap_space.insert(swap);
        if (swap_space.size() > swap_size)
        {
            cout << "Error: Swap space is full\n";
            return 1;
        }
    }
    if (swap_space.find(i) != swap_space.end())
        swap_space.erase(i);
    memory_frames.insert(i);
    memory.push(i);
}
```

1.2 Random

The basic idea behind this algorithm is that when a page fault occurs and the operating system needs to replace a page in memory, it chooses a random page to be replaced.

The random page replacement algorithm is very simple to implement and does not require much overhead in terms of computational resources. However, it has some drawbacks. Since the

algorithm selects pages randomly, it may replace a frequently used page, causing the system to perform poorly. In addition, the algorithm does not take into account the access patterns of the pages, which can lead to poor performance.
The code snippet for the logic is shown below.

```
for (auto i : requests)
{

    if (memory_frames.find(i) != memory_frames.end())
        continue;

    pageFaults++;
    if (memory_frames.size() >= memory_size)
    {
        int index = rand() % memory_size;
        auto itr = memory_frames.begin();
        advance(itr, index);
        memory_frames.erase(itr);
        swap_space.insert(*itr);
        if (swap_space.size() > swap_size)
        {
            cout << "Error: Swap space is full\n";
            return 1;
        }
    }
    if (swap_space.find(i) != swap_space.end())
        swap_space.erase(i);
    memory_frames.insert(i);
}
```

1.3 LRU

In LRU algorithm, the page that has not been accessed for the longest time is chosen for replacement when a page fault occurs. The basic idea behind the LRU algorithm is to prioritize the pages that have been used most recently and to replace those that have not been used for a long time. This helps to ensure that the most frequently accessed pages remain in memory and the least frequently accessed pages are swapped out. While the LRU algorithm can be effective in improving memory performance, it requires additional overhead to maintain the page table and track access times.
The code snippet for the logic is shown below.

```
for (int i = 0; i < requests.size(); i++)
{
    if (memory_frames.find(requests[i]) != memory_frames.end())
    {
        memory_frames[requests[i]] = i;
        continue;
    }

    pageFaults++;
    if (memory_frames.size() >= memory_size)
    {
        int swap = get_lru_request(memory_frames);
        cout << swap << endl;
```

```
        memory_frames.erase(swap);
        swap_space.insert(swap);
        if (swap_space.size() > swap_size)
        {
            cout << "Error: Swap space is full\n";
            return 1;
        }
    }
    if (swap_space.find(requests[i]) != swap_space.end())
        swap_space.erase(requests[i]);
    memory_frames[requests[i]] = i;
}
```

2 Plots

For making the plots, I have used the following request sequences:

1. **req1.dat**: Default requests
2. **seg-loc.dat**: LRU better than FIFO (Segmented Locality)
3. **spac-temp.dat**: LRU does better (Spacial & Temporal Locality)
4. **random.dat**: Random does better (1-70 Loop)
5. **80-20.dat**: 80-20 workload requests. (Natural)

The request sequences are provided with the submission. The number of blocks used is given in the caption of the plots.

3 Analysis

From the plots, we can observe the following:

1. For the given requests (**req1.dat**), all algorithms are observed to be giving the same results; this is because the number of requests is small and randomly distributed.
2. For the second request file, LRU performs better as it keeps track of the least recently used files, while FIFO replaces them without checking the pattern of the request, i.e., segmented locality.
3. The third request file has a very high spatial and temporal locality, as they are accessed in a stepped random loop; in this case, LRU and FIFO, perform much better than random
4. The fourth request file has continuous looped requests. In this case, LRU and FIFO perform very poorly, as whenever the loop resets, the previous caches would have been reset. Therefore Random is much better.
5. The last request file has a natural 80-20 workload. In this case, LRU performs better than FIFO and Random. Therefore in standard caches, LRU is preferred over other algorithms.

Figure 1: Blocks = 60

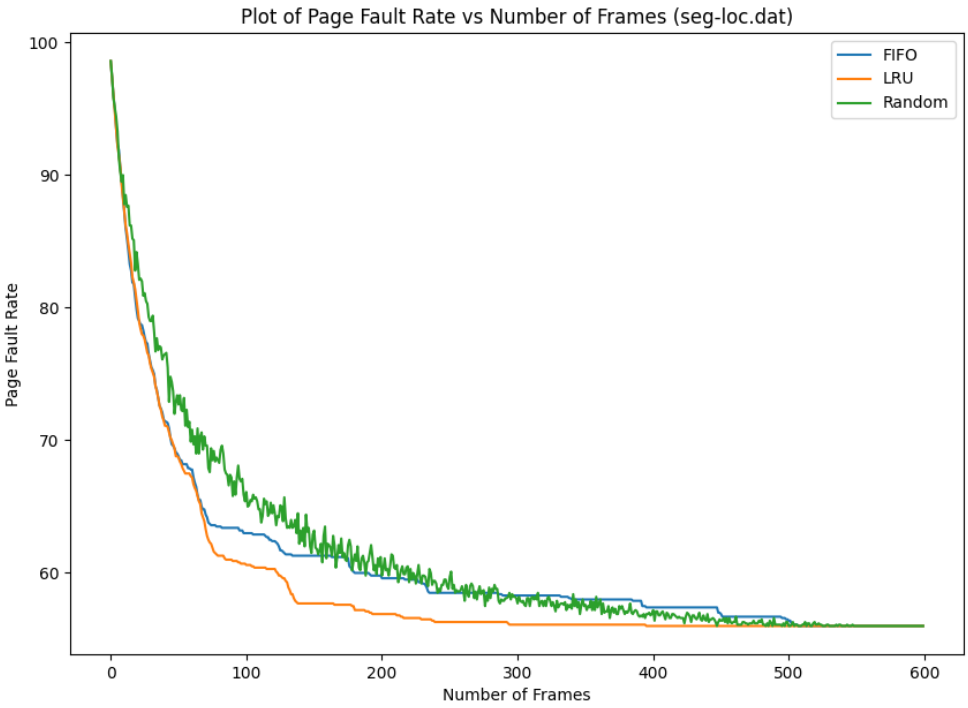

Figure 2: Blocks = 600

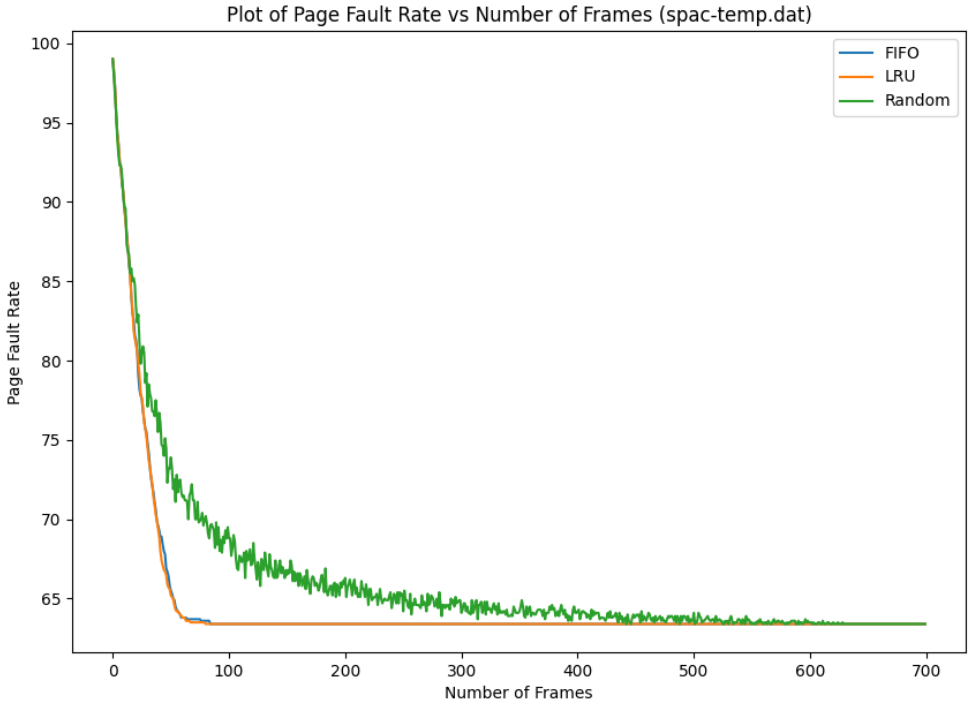

Figure 3: Blocks = 700

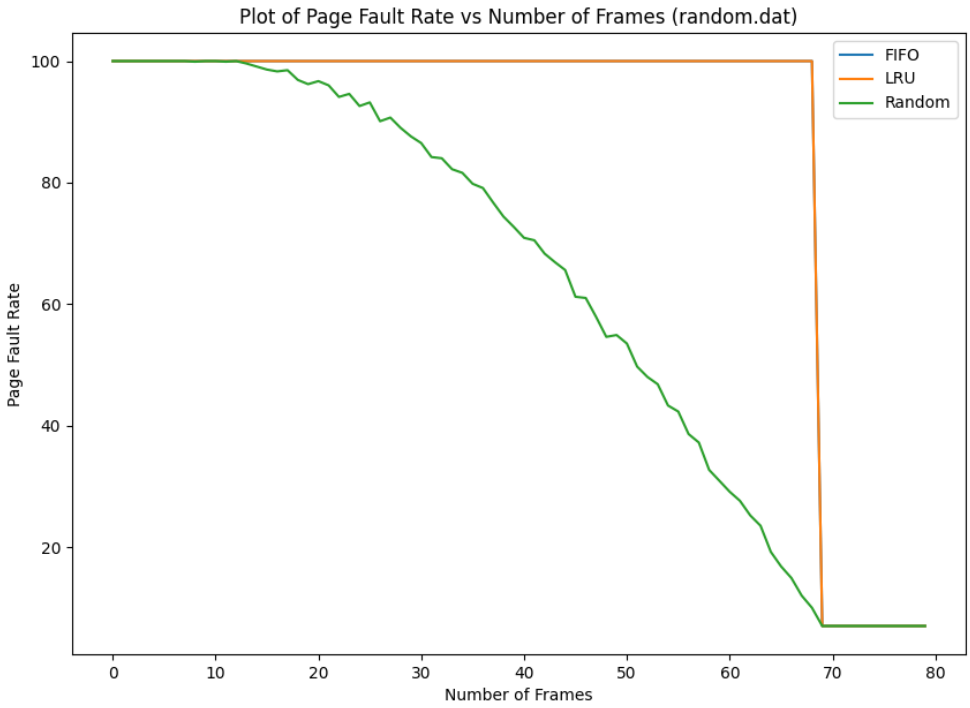

Figure 4: Blocks = 80

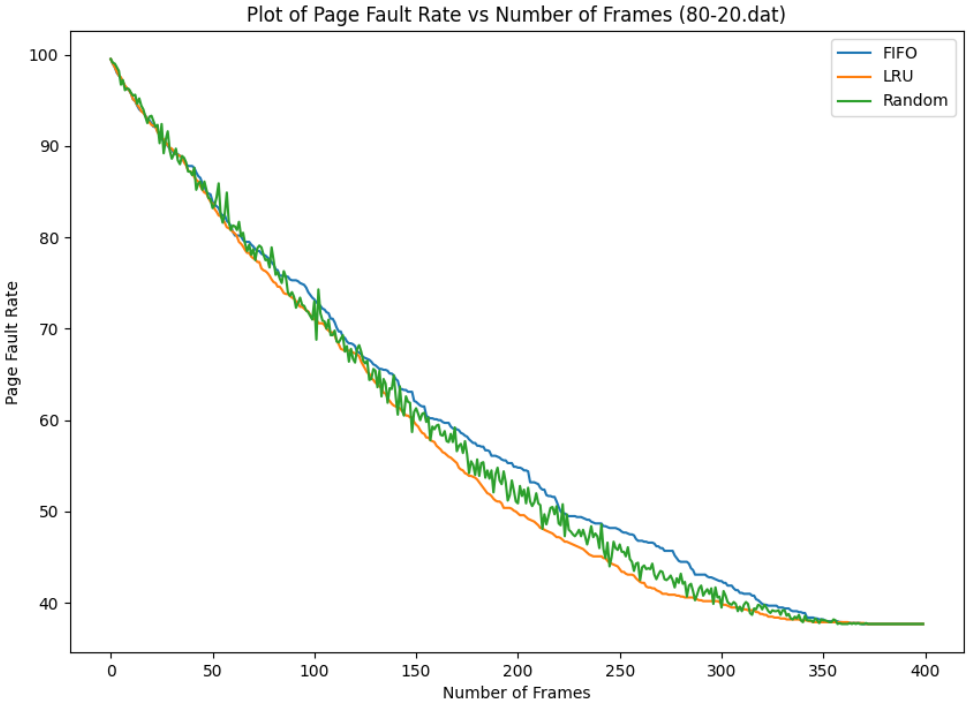

Figure 5: Blocks = 400