CSN-232 Operating Systems

LRU & Clock Algorithm Implementation

Nikunj Gupta(117114052), Kanav Gupta(17114042), Jaynil Jaiswal(17114040) , Mahesh Kale(17114041), Pradhumna Rathore(17114058), Sameer Gupta(17114067), Ashish Singh(17114014), Azim Bil Se Yeswanth ()

GitHub Link: https://github.com/NK-Nikunj/Page-Replacement

Introduction

The LRU (least recently used) algorithm uses the recent past as an approximation of the near future ant it replaces the page that has not been used for the longest period of time.

There are mainly three methods of implementing a LRU page replacement algorithm as follows:

1. **Counters implementation :** Add to the CPU a logical clock and increment it for every memory reference. Associate with each page entry a time-of-use field and, whenever a page is referenced, copy the clock register to the time-of-use field of that page. To replace a page, search the entire page table to find the one with the smallest time value.
2. **Stack Implementation :** Keep a stack of page numbers in a doubly linked list with head and tail pointers and, whenever a page is referenced, move it to the top of the stack. To replace a page, the LRU page is always at the bottom. Comparing to counters implementation, each update is more expensive, but there is no search for replacement.
3. **Aging Register implementation :** The aging algorithm is a descendant of the NFU algorithm, with modifications to make it aware of the time span of use. Instead of just incrementing the counters of pages referenced, putting equal emphasis on page references regardless of the time, the reference counter on a page is first shifted right (divided by 2), before adding the referenced bit to the left of that binary number. For instance, if a page has referenced bits 1,0,0,1,1,0 in the past 6 clock ticks, its referenced counter will look like this:

10000000, 01000000, 00100000, 10010000, 11001000, 01100100. Page references closer to the present time have more impact than page references long ago. This ensures that pages referenced more recently, though less frequently referenced, will have higher priority over pages more frequently referenced in the past. Thus, when a page needs to be swapped out, the page with the lowest counter will be chosen.

Clock Algorithm:

- The clock algorithm is a more efficient implementation of the second chance algorithm, which arranges pages in a circular queue and a pointer indicates which page is to be checked next.
- When a victim page is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits.As it advances, it clears the reference bits.
- Once a victim page is found, the page is replaced and the new page is inserted in the circular queue in that position.
- In the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance and then replaces the initial page

Getting started

Pre-Requisites

The program is written in standard C++14 standards and requires a C++ compiler (`gcc`, `clang` etc.). Following are the pre-requisites to building:

```
* A C++ compiler

* Make

* CMake (Preferred)
```

Note: To run *graph.py* to generate graphs for the benchmarks, you require to install `matplotlib` and `tkinter`. For Ubuntu/Debian systems, the commands are as follows:

```
$ sudo apt install python-tk

$ pip install matplotlib
```

Building

The complete code for the LRU and clock are given as header files for easy and convenient use. The library can be utilized simply by copy-pasting into your personal project. Here are the instructions to run `benchmarks`

1. Create a build folder

```
$ cd benchmarks
```

```
$ mkdir build && cd build
```

1. Invoke cmake

```
$ cmake ..
```

1. Build files

```
$ make
```

If you have done the setup right, you should see 4 executables `lru_aging`, `lru_counter`, `lru_stack` and `clock`. To run, simply execute the executable:

```
$ ./lru_counter
```

Once all executables are in place, we can plot the graph from the given output. To generate graph, run graph.py:

```
$ python2 graph.py
```

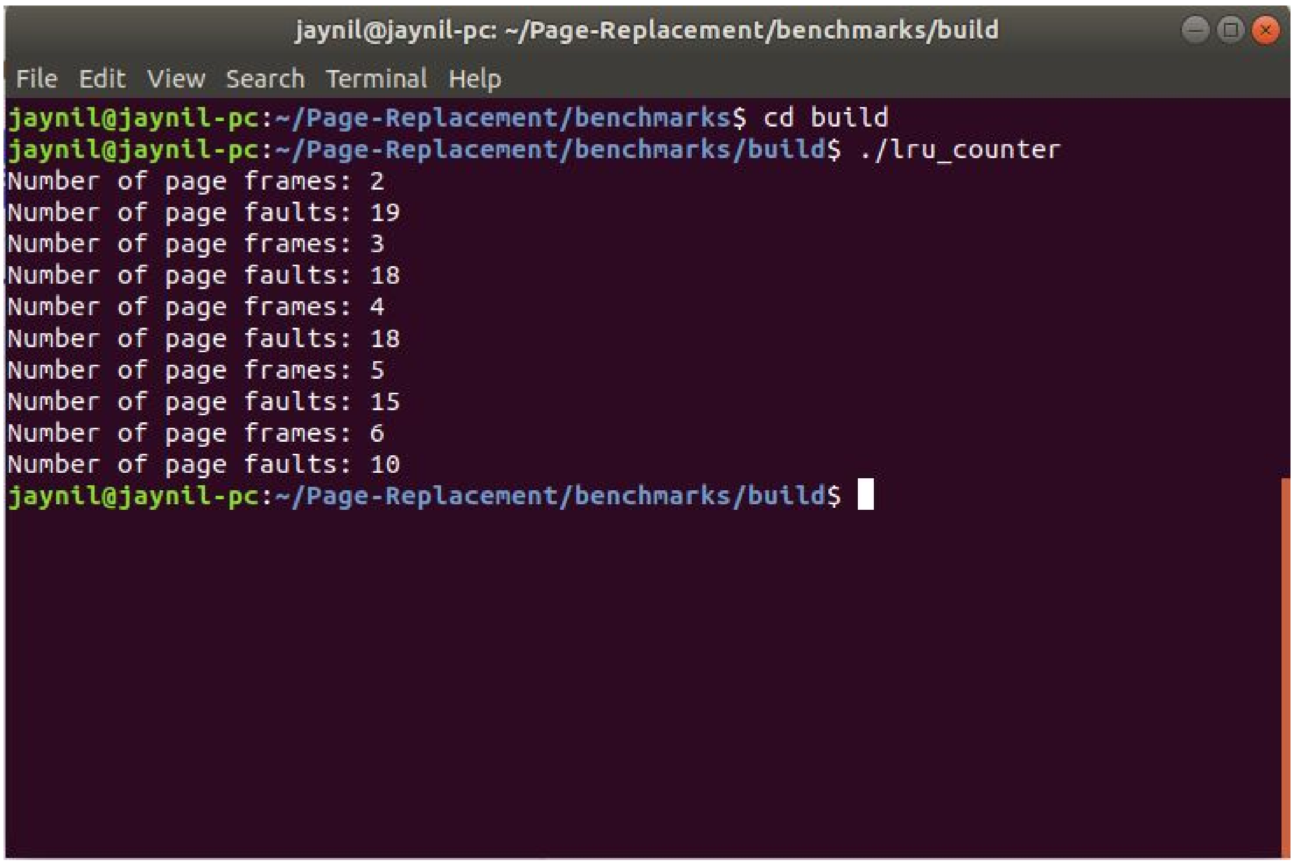
Note: In case you do not have Cmake installed, you can use a fallback makefile option. To get the above executables, simply run the *Makefile* provided in the *benchmarks* folder.

```
$ cd benchmarks
$ make
```

Results

Following are the benchmarks of the different implementations that we have done in this coding assignment. **All the complexities mentioned below will be for simulation of our implementation of algorithms.**

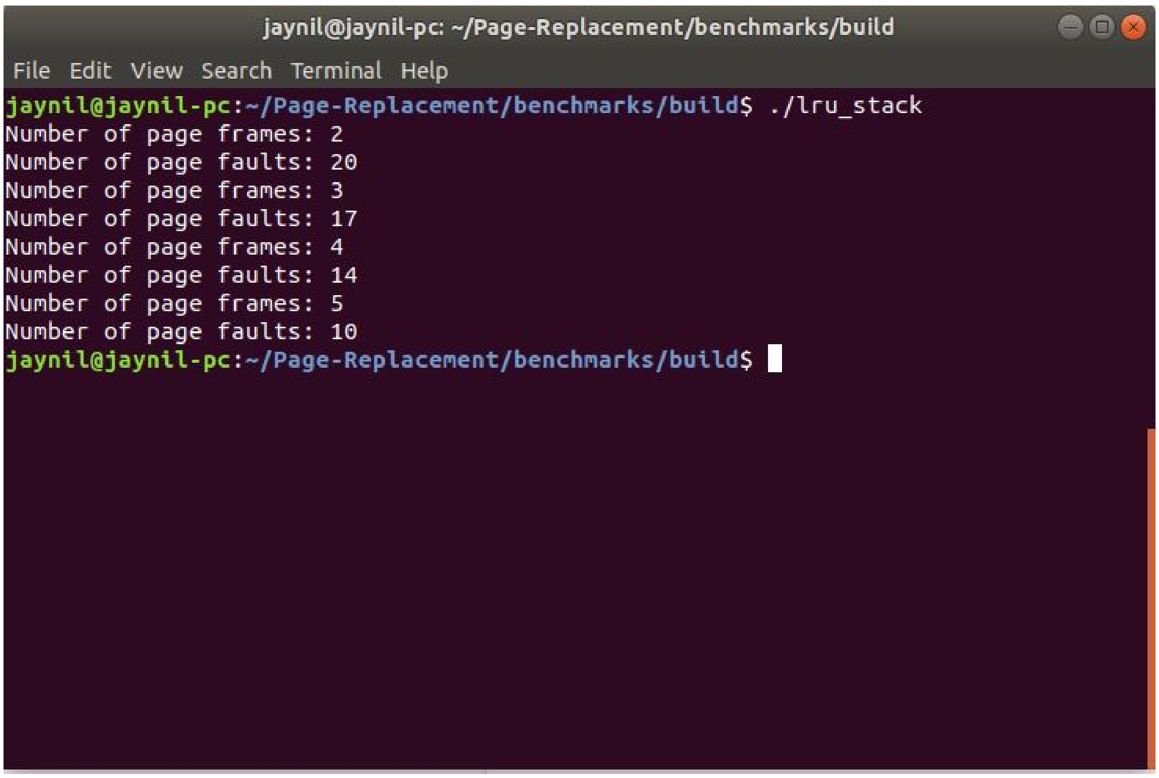
- LRU implementation using Counters:



Complexity :

Complexity of this algorithm is O(n*(mlog(m))) where ‘m’ is the no. of frames and ‘n’ is the no. of incoming processes.

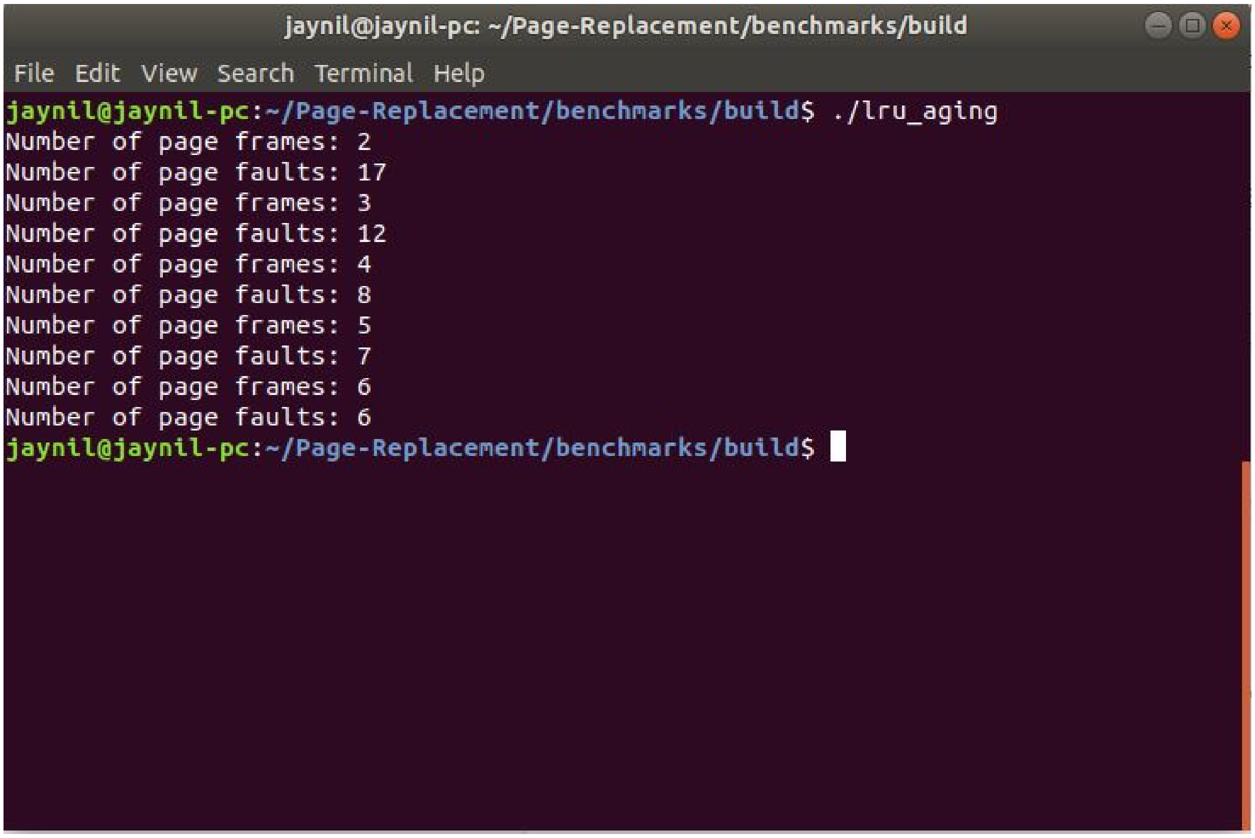
- LRU implementation using Stack:



Complexity:

Complexity of this algorithm is O(n*m) where ‘m’ is the no. of frames and ‘n’ is the no. of incoming processes.

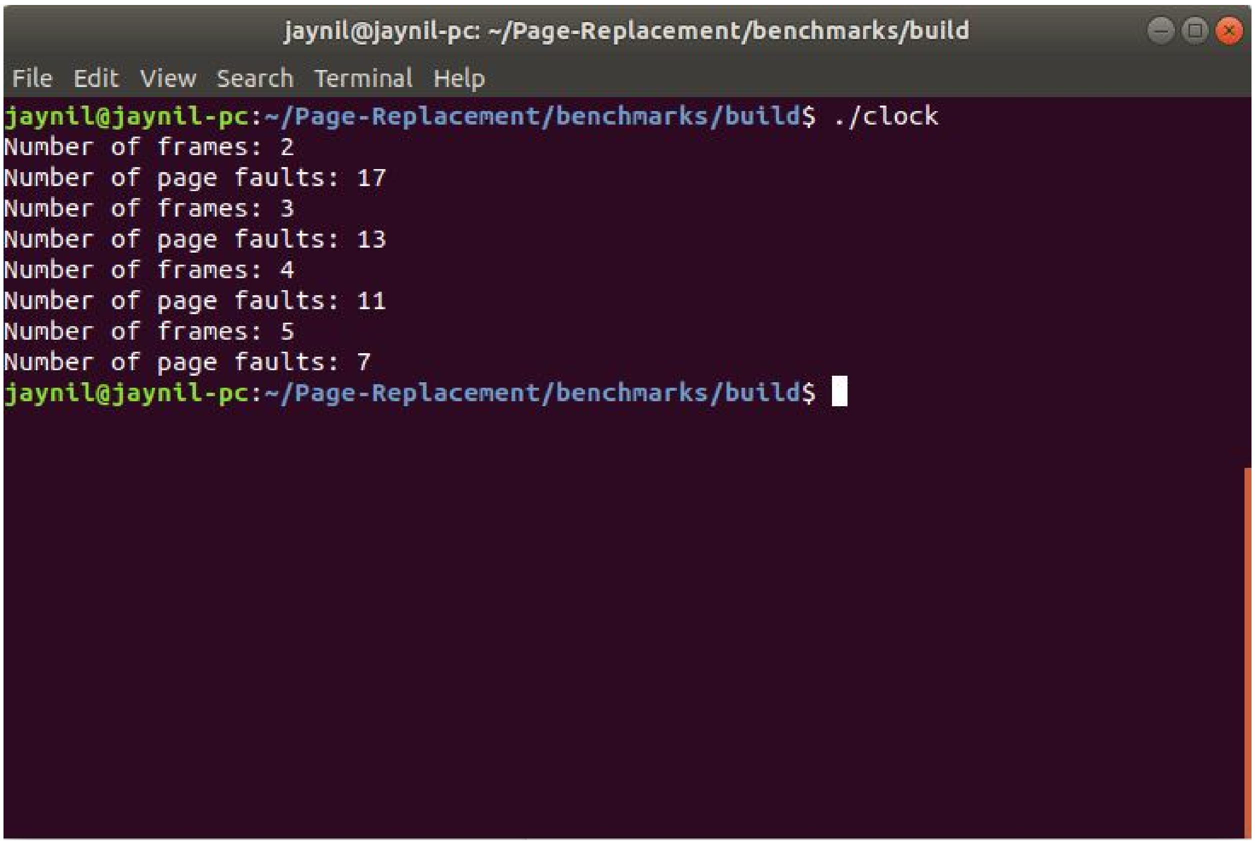
- LRU implementation using Aging Register:



Complexity:

Complexity of this algorithm is O(n*(mlog(m))) where ‘m’ is the no. of frames and ‘n’ is the no. of incoming processes.

- Clock algorithm implementation:



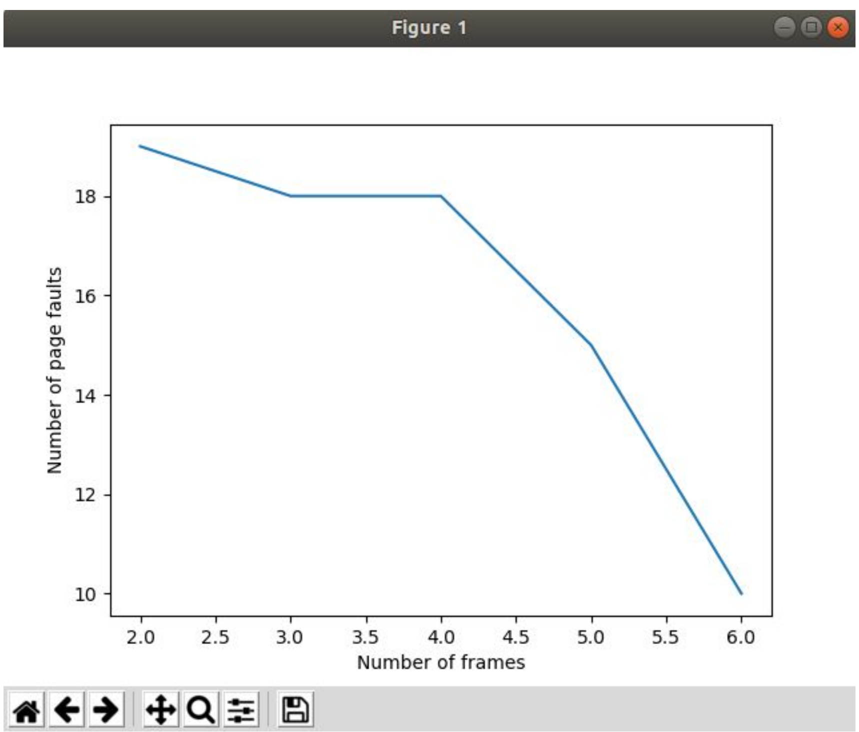
Complexity:

Complexity of this algorithm is $O(n*m^2)$ where ‘m’ is the no. of frames and ‘n’ is the no. of incoming processes.

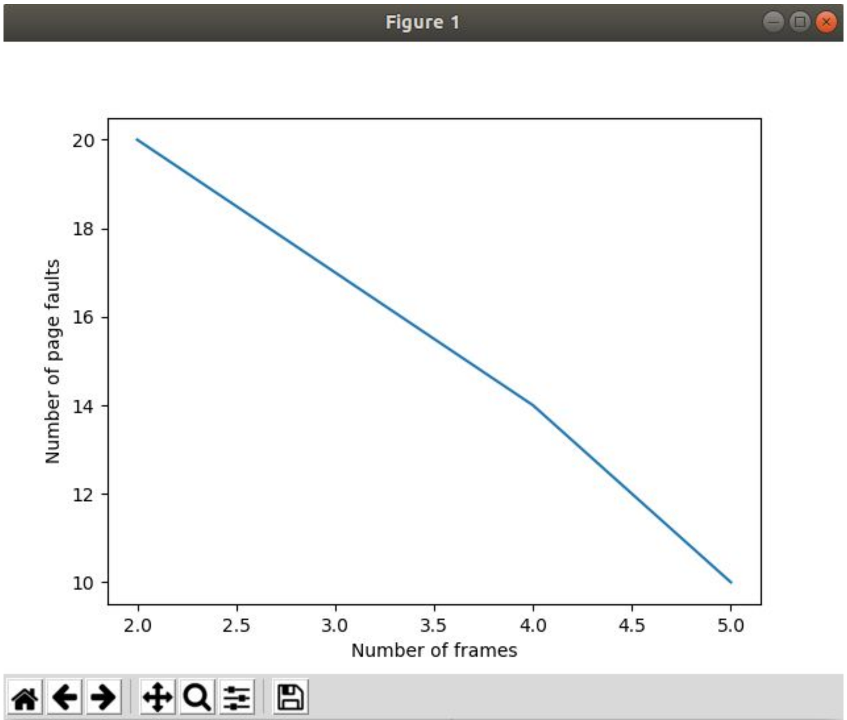
Following are the graph plotted between number of page faults against number of frames.

Also as we can see that as the no. of frames increases the no. of page faults decrease in any of the algorithm. This happens because as more no. of pages can be fit into page table there are lesser chances to have a missing page since most of them would be in the table already.

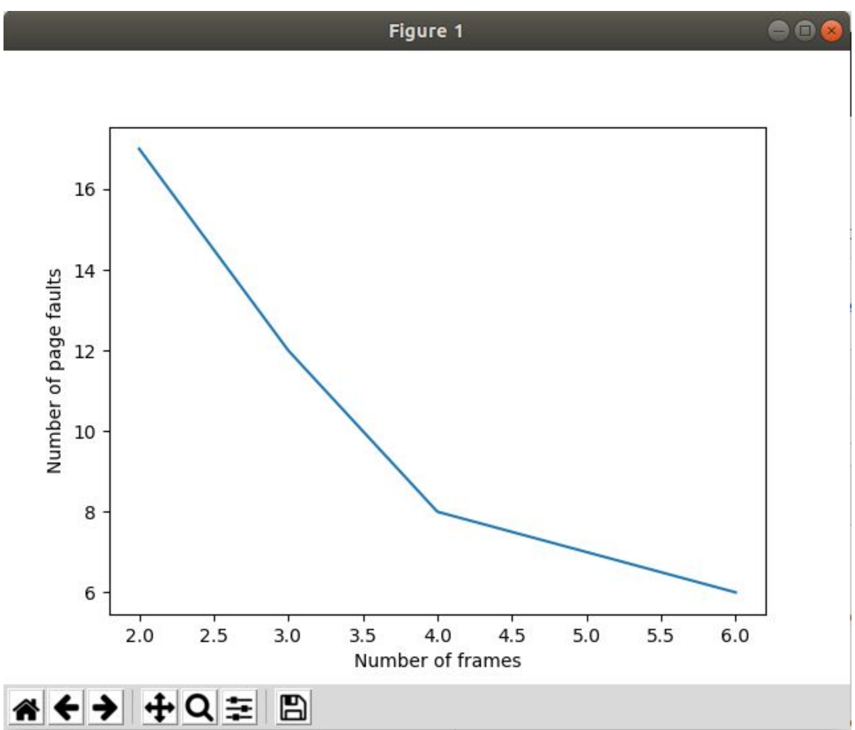
LRU using counter:



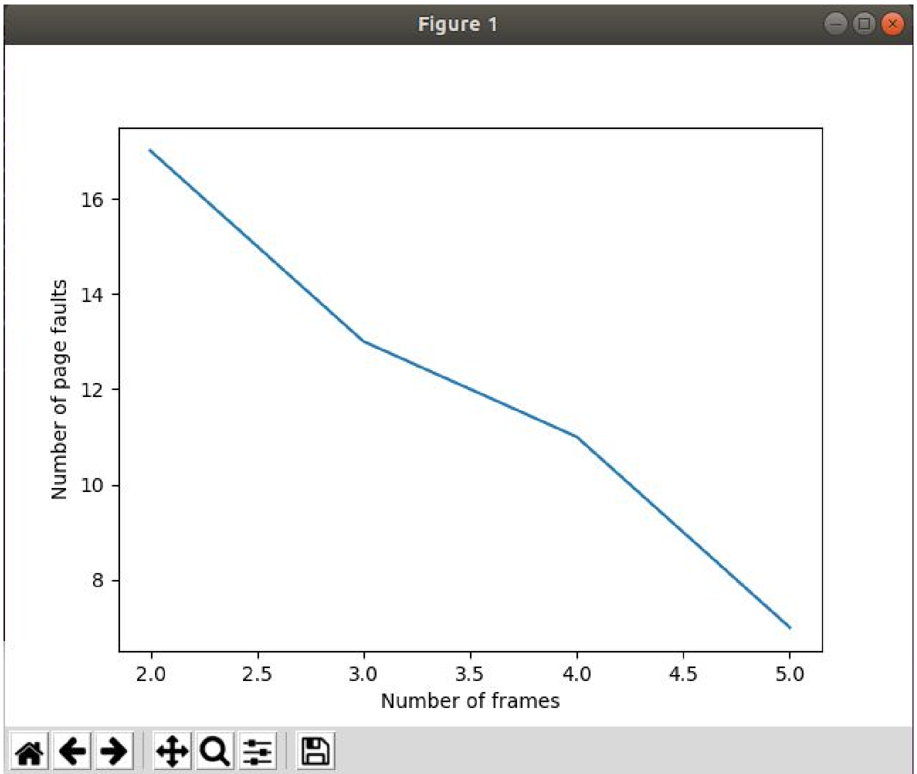
LRU using Stack:



LRU using Aging register:



Clock :



Conclusion

Hence we have clearly understood the LRU algorithm for page replacement and came up with our own implementations of LRU using different methods like using counters, stack and aging registers. We have also understood and implemented the clock algorithm which is similar to LRU.